CS 113 PROGRAMMING IN JAVA



Milan Samplewala
Los Angeles Mission College

LibreTexts™

CS 113 Programming in Java (Milan)

This text is disseminated via the Open Education Resource (OER) LibreTexts Project (https://LibreTexts.org) and like the thousands of other texts available within this powerful platform, it is freely available for reading, printing, and "consuming."

The LibreTexts mission is to bring together students, faculty, and scholars in a collaborative effort to provide an accessible, and comprehensive platform that empowers our community to develop, curate, adapt, and adopt openly licensed resources and technologies; through these efforts we can reduce the financial burden born from traditional educational resource costs, ensuring education is more accessible for students and communities worldwide.

Most, but not all, pages in the library have licenses that may allow individuals to make changes, save, and print this book. Carefully consult the applicable license(s) before pursuing such effects. Instructors can adopt existing LibreTexts texts or Remix them to quickly build course-specific resources to meet the needs of their students. Unlike traditional textbooks, LibreTexts' web based origins allow powerful integration of advanced features and new technologies to support learning.



LibreTexts is the adaptable, user-friendly non-profit open education resource platform that educators trust for creating, customizing, and sharing accessible, interactive textbooks, adaptive homework, and ancillary materials. We collaborate with individuals and organizations to champion open education initiatives, support institutional publishing programs, drive curriculum development projects, and more.

The LibreTexts libraries are Powered by NICE CXone Expert and was supported by the Department of Education Open Textbook Pilot Project, the California Education Learning Lab, the UC Davis Office of the Provost, the UC Davis Library, the California State University Affordable Learning Solutions Program, and Merlot. This material is based upon work supported by the National Science Foundation under Grant No. 1246120, 1525057, and 1413739.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation nor the US Department of Education.

Have questions or comments? For information about adoptions or adaptions contact info@LibreTexts.org or visit our main website at https://LibreTexts.org.

This text was compiled on 06/06/2026

TABLE OF CONTENTS

Licensing

1: Java Program Design and Development

- 1.1: Introduction
- 1.2: Designing Good Programs
- 1.3: Designing a Riddle Program
- 1.4: Java Language Elements
- 1.5: Editing, Compiling, and Running a Java Program
- 1.6: From the Java Library- System and PrintStream

2: Objects- Defining, Creating, and Using

- 2.1: Introduction
- 2.2: Using String Objects
- 2.3: Drawing Shapes with the Graphics Object (Optional)
- 2.4: Class Definition
- 2.5: Case Study- Simulating a Two-Person Game
- 2.6: From the Java Library- java.util.Scanner

3: Methods- Communicating with Objects

- 3.1: Introduction
- 3.2: Passing Information to an Object
- 3.3: Constructor Methods
- 3.4: Retrieving Information from an Object
- 3.5: Passing a Value and Passing a Reference
- 3.6: Flow of Control- Control Structures
- 3.7: Testing an Improved OneRowNim
- 3.8: From the Java Library- java.lang.Object
- 3.9: Object-Oriented Design- Inheritance and Polymorphism
- 3.10: Drawing Lines and Defining Graphical Methods (Optional)

4: Input/Output- Designing the User Interface

- 4.1: Introduction
- 4.2: The User Interface
- 4.3: A Command-line Interface
- 4.4: A Graphical User Interface (GUI)
- 4.5: Case Study- The One Row Nim Game
- 4.6: From the Java Library- java.io.File and file input (Optional)

5: Java Data and Operators

- 5.1: Introduction
- 5.2: Boolean Data and Operators
- 5.3: Numeric Data and Operators
- 5.4: From the Java Library java.lang.Math
- 5.5: Numeric Processing Examples
- 5.6: From the Java Library- java.text.NumberFormat
- 5.7: Character Data and Operators

- 5.8: Example- Character Conversions
- 5.9: Problem Solving = Representation + Action

6: Control Structures

- 6.1: Introduction
- 6.2: Flow of Control- Repetition Structures
- 6.3: Counting Loops
- 6.4: Example- Car Loan
- 6.5: Graphics Example- Drawing a Checkerboard
- 6.6: Conditional Loops
- 6.7: Example- Computing Averages
- 6.8: Example- Data Validation
- 6.9: Principles of Loop Design
- 6.10: The switch Multiway Selection Structure
- 6.11: Object-Oriented Design- Structured Programming
- 6.12: Special Topic- What Can Be Computed?

7: Strings and String Processing

- 7.1: Introduction
- 7.2: String Basics
- 7.3: Finding Things Within a String
- 7.4: Example- Keyword Search
- 7.5: From the Java Library- StringBuffer
- 7.6: Retrieving Parts of Strings
- 7.7: Example- Processing Names and Passwords
- 7.8: Processing Each Character in a String
- 7.9: Comparing Strings
- 7.10: From the Java Library- StringTokenizer
- 7.11: Handling Text in a Graphics Context (Optional)

8: Arrays and Array Processing

- 8.1: Introduction
- 8.2: One-Dimensional Arrays
- 8.3: Simple Array Examples
- 8.4: Example- Counting Frequencies of Letters
- 8.5: Array Algorithms- Sorting
- 8.6: Array Algorithms- Searching
- 8.7: Two-Dimensional Arrays
- 8.8: Multidimensional Arrays (Optional)
- 8.9: Object-Oriented Design- Polymorphic Sorting (Optional)
- 8.10: From the Java Library- java.util.Vector
- 8.11: Case Study- An N-Player Computer Game
- 8.12: A GUI-Based Game (Optional Graphics)

9: Exceptions- When Things Go Wrong

- 9.1: Introduction
- 9.2: Handling Exceptional Conditions
- 9.3: Java's Exception Hierarchy
- 9.4: Handling Exceptions Within a Program
- 9.5: Error Handling and Robust Program Design

- 9.6: Creating and Throwing Your Own Exceptions
- 9.7: From the Java Library- javax.swing.JOptionPane

10: Recursive Problem Solving

- 10.1: Introduction
- 10.2: Recursive Definition
- 10.3: Recursive String Methods
- 10.4: Recursive Array Processing
- 10.5: Example- Drawing (Recursive) Fractals
- 10.6: Object-Oriented Design- Tail Recursion
- 10.7: Object-Oriented Design- Recursion or Iteration?
- 10.8: Special Topic- Exploring the Mandelbrot Set
- 10.9: From the Java Library- javax.swing.JComboBox

Index

Glossary

Detailed Licensing

Detailed Licensing

Licensing

*A detailed breakdown of this resource's licensing can be found in **Back Matter/Detailed Licensing**.*

CHAPTER OVERVIEW

1: Java Program Design and Development

Learning Objectives

After studying this chapter, you will

- Know the basic steps involved in program development.
- Understand some of the basic elements of the Java language.
- Know how to use simple output operations in a Java program.
- Be able to distinguish between different types of errors in a
- Be able to distinguish between different types of errors in a program.
- Understand how a Java program is translated into machine language.
- Understand the difference between a Java console application and a Java Swing application.
- Know how to edit, compile, and run Java programs.

1.1: Introduction
1.2: Designing Good Programs
1.3: Designing a Riddle Program
1.4: Java Language Elements
1.5: Editing, Compiling, and Running a Java Program
1.6: From the Java Library- System and PrintStream

This page titled 1: Java Program Design and Development is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

1.1: Introduction

This chapter introduces some of the basic concepts and techniques involved in Java program design and development. We begin by identifying the main steps in designing an object-oriented program. The steps are illustrated by designing a program that “asks” and “answers” riddles. As an example of a riddle, consider the question “What is black and white and read all over?” The answer, of course, is a newspaper. Following the design phase, we then focus on the steps involved in coding a Java program, including the process of editing, compiling, and running a program. Because Java programs can be text based applications or window based graphical applications, we describe how the coding process differs for these two varieties.

Next we begin to familiarize ourselves with Java’s extensive class library by studying its PrintStream and System classes. These classes contain objects and methods that enable us to print output from a program. By the end of the chapter you will be able to design and write a Java application that “sings” your favorite song.

1.1: Introduction is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by LibreTexts.

- **1: Java Program Design and Development** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

1.2: Designing Good Programs

Programming is not simply a question of typing Java code. Rather, it involves a considerable amount of planning and careful designing. Badly designed programs rarely work correctly. Even though it is tempting for novice programmers to start entering code almost immediately, one of the first rules of programming is

> **JAVA PROGRAMMING TIP**
>
> The sooner you begin to type code, the longer the program will take to finish, because careful design of the program must precede coding. This is particularly true of object-oriented programs.

In other words, the more thought and care you put into designing a program, the more likely you are to end up with one that works correctly. The following subsections provide a brief overview of the program development process.

The Software Engineering Life Cycle

Software engineering is the process of designing and writing software. The *software life cycle* refers to the different phases involved in the design and development of a computer program. Our presentation of examples in the book will focus on four phases of the overall life cycle. In the *specification* phase we provide a statement of the problem and a detailed description of what the program will do. In the *design* phase we describe the details of the various classes, methods, and data that will be used in the program. The *implementation* phase refers to the actual coding of the program into Java. In the *testing* phase we test the program's performance to make sure it is correct, recoding it or redesigning it as necessary.

Figure 1.1 gives a more detailed overview of the program development process, focusing most of the attention on the design phase of the software life cycle. It shows that designing an object-oriented program is a matter of asking the right questions about the classes, data, and methods that make up the program.

Overall, the program development process can be viewed as one that repeatedly applies the divide-and-conquer principle. That is, most programming problems can be repeatedly divided until you have a collection of relatively easy-to-solve subproblems, each of which can be handled by an object. In this way the program is divided into a collection of interacting objects. For each object we design a class. During class design, each object is divided further into its variables and methods.

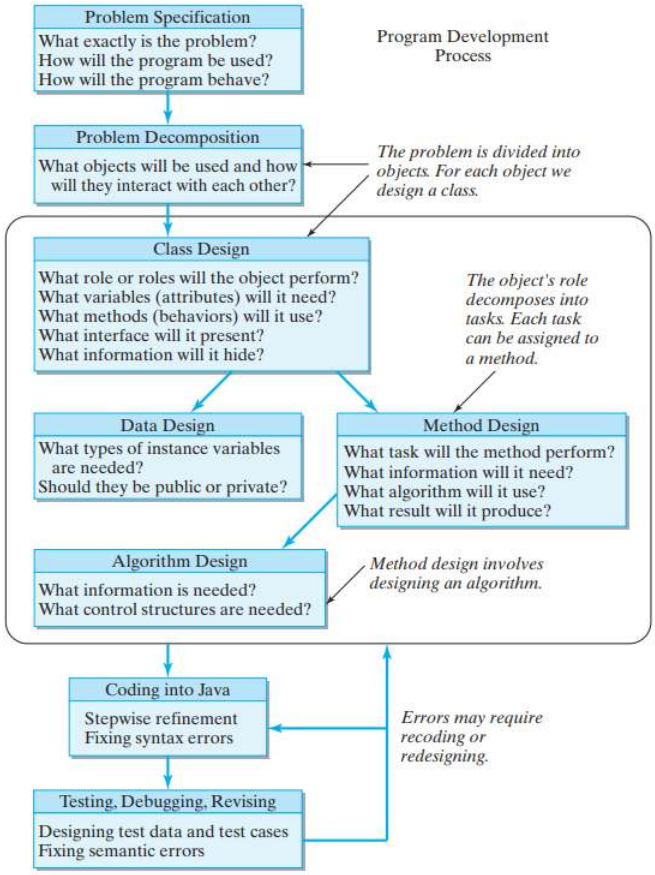


Figure 1.1: An overview of the program development process.

When should we stop subdividing? How much of a task should be assigned to a single object or a single method? The answers to these and similar questions are not easy. Good answers require the kind of judgment that comes through experience, and frequently there is more than one good way to design a solution. Here again, as we learn more about object-oriented programming, we'll learn more about how to make these design decisions.

This page titled 1.2: Designing Good Programs is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **1.2: Designing Good Programs** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

1.3: Designing a Riddle Program

The first step in the program-development process is making sure you understand the problem (Fig. [fig:progdev]). Thus, we begin by developing a detailed specification, which should address three basic questions:

- What exactly is the problem to be solved?
- How will the program be used?
- How should the program behave?

In the real world, the problem specification is often arrived at through an extensive discussion between the customer and the developer. In an introductory programming course, the specification is usually assigned by the instructor.

To help make these ideas a little clearer, let's design an object-oriented solution to a simple problem.

Problem Specification

Design a class that will represent a riddle with a given question and answer. The definition of this class should make it possible to store different riddles and to retrieve a riddle's question and answer independently

Problem Decomposition

Most problems are too big and too complex to be tackled all at once. So the next step in the design process is to divide the problem into parts that make the solution more manageable. In the object-oriented approach, a problem is divided into objects, where each object will handle one specific aspect of the program's overall job. In effect, each object will become an expert or specialist in some aspect of the program's overall behavior.

Note that there is some ambiguity here about how far we should go in decomposing a given program. This ambiguity is part of the design process. How much we should decompose the program before its parts become "simple to solve" depends on the problem we're trying to solve and on the problem solver.

One useful design guideline for trying to decide what objects are needed is the following:

JAVA EFFECTIVE DESIGN

Looking for Nouns. Choosing a program's objects is often a matter of looking for nouns in the problem specification.

Again, there's some ambiguity involved in this guideline. For example, the key noun in our current problem is *riddle*, so our solution will involve an object that serves as a model for a riddle. The main task of this Java object will be simply to represent a riddle. Two other nouns in the specification are *question* and *answer*. Fortunately, Java has built-in String objects that represent strings of characters such as words or sentences. We can use two String objects for the riddle's question and answer. Thus, for this simple problem, we need only design one new type of object—a riddle—whose primary role will be to represent a riddle's question and answer.

Don't worry too much if our design decisions seem somewhat mysterious at this stage. A good understanding of object-oriented design can come only after much design experience, but this is a good place to start.

Object Design

Once we have divided a problem into a set of cooperating objects, designing a Java program is primarily a matter of designing and creating the objects themselves. In our example, this means we must now design the features of our riddle object. For each object, we must answer the following basic design questions:

- What role will the object perform in the program?
- What data or information will it need?
- What actions will it take?
- What interface will it present to other objects?
- What information will it hide from other objects?

For our riddle object, the answers to these questions are shown in Figure 1.2. Note that although we talk about "designing an object," we are really talking about designing the object's class. A class defines the collection of objects that belong to it. The class can be considered the object's *type*. This is the same as for real-world objects. Thus, Seabiscuit is a horse—that is, Seabiscuit is an object of type horse. Similarly, an individual riddle, such as the newspaper riddle, is a riddle. That is, it is an object of type Riddle.

The following discussion shows how we arrived at the decisions for the design specifications for the Riddle class, illustrated in Figure 1.2.

- Class Name: Riddle
- Role: To store and retrieve a question and answer
- Attributes (Information)
  - question: A variable to store a riddle's question (private)
  - answer: A variable to store a riddle's answer (private)
- Behaviors
  - Riddle(): A method to set a riddle's question and answer
  - getQuestion(): A method to return a riddle's question
  - getAnswer(): A method to return a riddle's answer

Figure 1.2: Design specification for the Riddle class.

The role of the Riddle object is to model an ordinary riddle. Because a riddle is defined in terms of its question and answer, our Riddle object will need some way to store these two pieces of information. As we learned in Chapter [chapter-intro], an instance variable is a named memory location that belongs to an object. The fact that the memory location is named, makes it easy to retrieve the data stored there by invoking the variable's name. For example, to print a riddle's question we would say something like "print question," and whatever is stored in *question* would be retrieved and printed.

In general, instance variables are used to store the information that an object needs to perform its role. They correspond to what we have been calling the object's attributes. Deciding on these variables provides the answer to the question, "What information does the object need?"

Next we decide what actions a Riddle object will take. A useful design guideline for actions of objects is the following:

JAVA EFFECTIVE DESIGN

Looking for Verbs. Choosing the behavior of an object is often a matter of looking for verbs in the problem specification.

For this problem, the key verbs are *set* and *retrieve*. As specified in Figure 1.2, each Riddle object should provide some means of setting the values of its question and answer variables and a means of retrieving each value separately.

Each of the actions we have identified will be encapsulated in a Java method. As you recall from Chapter [chapter-intro], a method is a named section of code that can be *invoked*, or called upon, to perform a particular action. In the object-oriented approach, calling a method (method invocation) is the means by which interaction occurs among objects. Calling a method is like sending a message between objects. For example, when we want to get a riddle's answer, we would invoke the getAnswer() method. This is like sending the message "Give me your answer." One special method, known as a constructor, is invoked when an object is first created. We will use the Riddle() constructor to give specific values to riddle's question and answer variables.

In designing an object, we must decide which methods should be made available to other objects. This determines what interface the object should present and what information it should hide from other objects. In general, those methods that will be used to communicate with an object are designated as part of the object's interface. Except for its interface, all other information maintained by each riddle should be kept "hidden" from other objects. For example, it is not necessary for other objects to know where a riddle object stores its question and answer. The fact that they are stored in variables named question and answer, rather than variables named ques and ans, is irrelevant to other objects.

JAVA EFFECTIVE DESIGN

Object Interface. An object's interface should consist of just those methods needed to communicate with or to use the object.

JAVA EFFECTIVE DESIGN

Information Hiding. An object should hide most of the details of its implementation.

Taken together, these various design decisions lead to the A UML class diagram representing the Riddle class. fig:rumlspecification shown in Figure 1.3. As our discussion has illustrated, we arrived at the decisions by asking and answering the right questions. In most classes the attributes (variables) are private. This is represented by a minus sign (−). In this example, the operations (methods) are public, which is represented by the plus sign (+). The figure shows that the Riddle class has two hidden (or private) variables for storing data and three visible (or public) methods that represent the operations that it can perform.

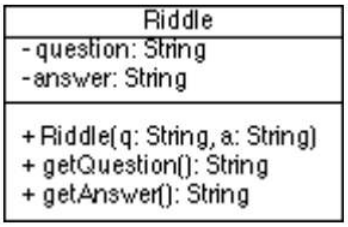


Figure 1.3: A UML class diagram representing the Riddle class.

Data, Methods, and Algorithms

Among the details that must be worked out in designing a riddle object is deciding what type of data, methods, and algorithms we need. There are two basic questions involved:

- What type of data will be used to represent the information needed by the riddle?
- How will each method carry out its task?

Like other programming languages, Java supports a wide range of different types of data, some simple and some complex. Obviously a riddle's question and answer should be represented by text. As we noted earlier, Java has a Stringtype, which is designed to store text, which can be considered a string of characters.

In designing a method, you have to decide what the method will do. In order to carry out its task, a method will need certain information, which it may store in variables. Plus, it will have to carry out a sequence of individual actions to perform the task. This is called its **algorithm**, which is a step-by-step description of the solution to a problem. And, finally, you must decide what result the method will produce. Thus, as in designing objects, it is important to ask the right questions:

- What specific task will the method perform?
- What information will it need to perform its task?
- What algorithm will the method use?
- What result will the method produce?

Methods can be thought of as using an algorithm to complete a required action. The algorithm required for the Riddle()constructor is very simple but also typical of constructors for many classes. It takes two strings and assigns the first to the question instance variable and then assigns the second to the answer instance variable. The algorithms for the other two methods for the Riddle class are even simpler. They are referred to as *get* methods that merely *return* or produce the value that is currently stored in an instance variable.

Not all methods are so simple to design, and not all algorithms are so simple. Even when programming a simple arithmetic problem, the steps involved in the algorithm will not always be as obvious as they are when doing the calculation by hand. For example, suppose the problem were to calculate the sum of a list of numbers. If we were telling our classmate how to do this problem, we might just say, "add up all the numbers and report their total." But this description is far too vague to be used in a program. By contrast, here's an algorithm that a program could use:

- Set the initial value of the sum to 0.
- If there are no more numbers to total, go to step 5.
- Add the next number to the sum.
- Go to step 2.
- Report the sum.

Note that each step in this algorithm is simple and easy to follow. It would be relatively easy to translate it into Java. Because English is somewhat imprecise as an algorithmic language, programmers frequently write algorithms in the programming language itself or in **pseudocode**, a hybrid language that combines English and programming language structures without being too fussy about programming language syntax. For example, the preceding algorithm might be expressed in pseudocode as follows:

```
sum = 0
while (more numbers remain)
    add next number to sum
print the sum
```

Of course, it is unlikely that an experienced programmer would take the trouble to write out pseudocode for such a simple algorithm. But many programming problems are quite complex and require careful design to minimize the number of errors that the program contains. In such situations, pseudocode could be useful.

Another important part of designing an algorithm is to *trace* it—that is, to step through it line by line—on some sample data. For example, we might test the list-summing algorithm by tracing it on the list of numbers shown in the margin.

Initially, the sum starts out at 0 and the list of numbers contains 54, 30, and 20. On each iteration through the algorithm, the sum increases by the amount of the next number, and the list diminishes in size. The algorithm stops with the correct total left under the sum column. While this trace didn't turn up any errors, it is frequently possible to find flaws in an algorithm by tracing it in this way.

Coding into Java

Once a sufficiently detailed design has been developed, it is time to start generating Java code. The wrong way to do this would be to type the entire program and then compile and run it. This generally leads to dozens of errors that can be both demoralizing and difficult to fix.

The right way to code is to use the principle of **stepwise refinement**. The program is coded in small stages, and after each stage the code is compiled and tested. For example, you could write the code for a single method and test that method before moving on to another part of the program. In this way, small errors are caught before moving on to the next stage.

The code for the Riddle class is shown in Figure [fig:riddleclass]. Even though we have not yet begun learning the details of the Java language, you can easily pick out the key parts in this program: the instance variables question and answer of type String, which are used to store the riddle's data; the Riddle() constructor and the getQuestion() and getAnswer() methods make up the interface. The specific language details needed to understand each of these elements will be covered in this and the following chapter.

```
/*
 * File: Riddle.java
 * Author: Java, Java, Java
 * Description: Defines a simple riddle.
 */
public class Riddle extends Object  // Class header
{                                   // Begin class body
   private String question;       // Instance variables
   private String answer;

   public Riddle(String q, String a) // Constructor method
   {
     question = q;
     answer = a;
   } // Riddle()

   public String getQuestion()   // Instance method
   {
     return question;
   } // getQuestion()

   public String getAnswer()     // Instance method
   {
     return answer;
   } //getAnswer()
} // Riddle class                  // End class body
```

Syntax and Semantics

Writing Java code requires that you know its syntax and semantics. A language's **syntax** is the set of rules that determines whether a particular statement is correctly formulated. As an example of a syntax rule, consider the following two English statements:

```
The rain in Spain falls mainly on the plain. // Valid
Spain rain the mainly in on the falls plain. // Invalid
```

The first sentence follows the rules of English syntax (grammar), and it means that it rains a lot on the Spanish plain. The second sentence does not follow English syntax, and, as a result, it is rendered meaningless. An example of a Java syntax rule is that a Java statement must end with a semicolon.

However, unlike in English, where one can still be understood even when one breaks a syntax rule, in a programming language the syntax rules are very strict. If you break even the slightest syntax rule—for example, if you forget just a single semicolon—the program won't work at all.

Similarly, the programmer must know the of the language—that is, the meaning of each statement. In a programming language, a statement's meaning is determined by what effect it will have on the program. For example, to set the sumto 0 in the preceding algorithm, an assignment statement is used to store the value 0 into the memory location named sum. Thus, we say that the statement

```
sum = 0;
```

assigns 0 to the memory location sum, where it will be stored until some other part of the program needs it.

Learning Java's syntax and semantics is a major part of learning to program. This aspect of learning to program is a lot like learning a foreign language. The more quickly you become fluent in the new language (Java), the better you will be at expressing

solutions to interesting programming problems. The longer you struggle with Java's rules and conventions, the more difficult it will be to talk about problems in a common language. Also, computers are a lot fussier about correct language than humans, and even the smallest syntax or semantic error can cause tremendous frustration. So, try to be very precise in learning Java's syntax and semantics.

Testing, Debugging, and Revising

Coding, testing, and revising a program is an repetitive process, one that may require you to repeat the different program-development stages shown in (Fig. 1.1). According to the stepwise-refinement principle, the process of developing a program should proceed in small, incremental steps, where the solution becomes more refined at each step. However, no matter how much care you take, things can still go wrong during the coding process.

A *syntax error* is an error that breaks one of Java's syntax rules. Such errors will be detected by the Java compiler. Syntax errors are relatively easy to fix once you understand the error messages provided by the compiler. As long as a program contains syntax errors, the programmer must correct them and recompile the program. Once all the syntax errors are corrected, the compiler will produce an executable version of the program, which can then be run.

When a program is run, the computer carries out the steps specified in the program and produces results. However, just because a program runs does not mean that its actions and results are correct. A running program can contain *semantic errors*, also called . A semantic error is caused by an error in the logical design of the program causing it to behave incorrectly, producing incorrect results.

Unlike syntax errors, semantic errors cannot be detected automatically. For example, suppose that a program contains the following statement for calculating the area of a rectangle:

```
return length + width;
```

Because we are adding length and width instead of multiplying them, the area calculation will be incorrect. Because there is nothing syntactically wrong with the expression length + width, the compiler won't detect an error in this statement. Thus, the computer will still execute this statement and compute the incorrect area.

Semantic errors can only be discovered by testing the program and they are sometimes very hard to detect. Just because a program appears to run correctly on one test doesn't guarantee that it contains no semantic errors. It might just mean that it has not been adequately tested.

Fixing semantic errors is known as *debugging* a program, and when subtle errors occur it can be the most frustrating part of the whole program development process. The various examples presented will occasionally provide hints and suggestions on how to track down *bugs*, or errors, in your code. One point to remember when you are trying to find a very subtle bug is that no matter how convinced you are that your code is correct and that the bug must be caused by some kind of error in the computer, the error is almost certainly caused by your code!

Writing Readable Programs

Becoming a proficient programmer goes beyond simply writing a program that produces correct output. It also involves developing good *programming style*, which includes how readable and understandable your code is. Our goal is to help you develop a programming style that satisfies the following principles:

- **Readability.** Programs should be easy to read and understand. Comments should be used to document and explain the program's code.
- **Clarity.** Programs should employ well-known constructs and standard conventions and should avoid programming tricks and unnecessarily obscure or complex code.
- **Flexibility.** Programs should be designed and written so that they are easy to modify.

Special Topic: Grace Hopper and the First Computer Bug

Rear Admiral Grace Murray Hopper (1906–1992) was a pioneer computer programmer and one of the original developers of the COBOL programming language, which stands for *CO*mmon *B*usiness-*O*riented *L*anguage. Among her many achievements and distinctions, Admiral Hopper also had a role in coining the term *computer bug*.

In August 1945, she and a group of other programmers were working on the Mark I, an electro-mechanical computer developed at Harvard that was one of the ancestors of today's electronic computers. After several hours of trying to figure out why the machine was malfunctioning, someone located and removed a two-inch moth from one of the computer's circuits. From then on whenever anything went wrong with a computer, Admiral Hopper and others would say "it had bugs in it." The first bug itself is still taped to Admiral Hopper's 1945 log book, which is now in the collection of the Naval Surface Weapons Center.

In 1991, Admiral Hopper was awarded the National Medal of Technology by President George Bush. To commemorate and honor Admiral Hopper's many contributions, the U.S. Navy recently named a warship after her. For more information on Admiral Hopper, see the Web site at

```
http://www.chips.navy.mil/
```

This page titled 1.3: Designing a Riddle Program is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **1.3: Designing a Riddle Program** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

1.4: Java Language Elements

In this section we will introduce some of the key elements of the Java language by describing the details of a small program. We will look at how a program is organized and what the various parts do. Our intent is to introduce important language elements, many of which will be explained in greater detail in later sections.

The program we will study is a Java version of the traditional HelloWorld program—"traditional" because practically every introductory programming text begins with it. When it is run, the HelloWorld program (Fig. 1.5) just displays the greeting "Hello, World!" on the console.

```
 /*
  * File: HelloWorld.java
  * Author: Java Java Java
  * Description: Prints Hello, World! greeting.
  */
public class HelloWorld extends Object // Class header
{                                      // Start class body
  private String greeting = "Hello, World!";
  public void greet()                  // Method definition
  {                                    // Start method body
      System.out.println(greeting); //  Output statement
  } // greet()                         // End method body
 public static void main(String args[])// Method header
 {
   HelloWorld helloworld;          // declare
   helloworld = new HelloWorld(); // create
   helloworld.greet();             // Method call
  }  //  main()
}  // HelloWorld                    // End class body
```

Figure 1.5: The HelloWorld application program.

Comments

The first thing to notice about the HelloWorld program is the use of comments. A **comment** is a non-executable portion of a program that is used to document the program. Because comments are not executable instructions they are just ignored by the compiler. Their sole purpose is to make the program easier for the programmer to read and understand.

The HelloWorld program contains examples of two types of Java comments. Any text contained within /* and */ is considered a comment. As you can see in HelloWorld, this kind of comment can extend over several lines and is sometimes called a *multiline* comment. A second type of comment is any text that follows double slashes (//) on a line. This is known as a *single-line comment* because it cannot extend beyond a single line.

When the compiler encounters the beginning marker (/*) of a multiline comment, it skips over everything until it finds a matching end marker (*/). One implication of this is that it is not possible to put one multiline comment inside of another. That is, one comment cannot be *nested*, or contained, within another comment. The following code segment illustrates the rules that govern the use of /* and */:

```
/* This first comment begins and ends on the same line. */
/* A second comment starts on this line ...
   and goes on ...
   and this is the last line of the second comment.
*/
/* A third comment starts on this line ...
    /* This is NOT a fourth comment. It is just
       part of the third comment.
   And this is the last line of the third comment.
*/
*/  This is an error because it is an unmatched end marker.
```

As you can see from this example, it is impossible to begin a new comment inside an already-started comment because all text inside the first comment, including /*, is ignored by the compiler.

JAVA LANGUAGE RULE

Comments. Any text contained within /* and */, which may span several lines, is considered a comment and is ignored by the compiler. Inserting double slashes (//) into a line turns the rest of the line into a comment.

Multiline comments are often used to create a *comment block* that provides useful documentation for the program. In HelloWorld, the program begins with a comment block that identifies the name of file that contains the program and its author and provides a brief description of what the program does.

For single-line comments, double slashes (//) can be inserted anywhere on a line of code. The result is that the rest of the line is ignored by the compiler. We use single-line comments throughout the HelloWorld program to provide a running commentary of its language elements.

JAVA PROGRAMMING TIP

A well-written program should begin with a comment block that provides the name of the program, its author, and a description of what the program does.

Program Layout

Another thing to notice about the program is how neatly it is arranged on the page. This is done deliberately so that the program is easy to read and understand. In Java, program expressions and statements may be arranged any way the programmer likes. They may occur one per line, several per line, or one per several lines. But the fact that the rules governing the layout of the program are so lax makes it all the more important that we adopt a good programming style, one that will help make programs easy to read.

So look at how things are presented in HelloWorld. Notice how beginning and ending braces, and , are aligned, and note how we use single-line comments to annotate ending braces. Braces are used to mark the beginning and end of different blocks of code in a Java program and it can sometimes be difficult to know which beginning and end braces are matched up. Proper indentation and the use of single-line comments make it easier to determine how the braces are matched up.

Similarly, notice how indentation is used to show when one element of the program is contained within another element. Thus, the elements of the HelloWorld class are indented inside of the braces that mark the beginning and end of the class. And the statements in the main() method are indented to indicate that they belong to that method. Use of indentation in this way, to identify the program's structure, makes the program easier to read and understand.

JAVA PROGRAMMING TIP

Use of Indentation. Indent the code within a block and align the block's opening and closing braces. Use a comment to mark the end of a block of code.

Keywords and Identifiers

The Java language contains 48 predefined *keywords* (Table 1.1). These are words that have special meaning in the language and whose use is reserved for special purposes. For example, the keywords used in the HelloWorldprogram (Fig. 1.5) are: class, extends, private, public, static, and void.

Table 1.1: Java keywords

```
abstract   default   goto          package      this
boolean    do        if            private      throw
break      double    implements    protected    throws
byte       enum      import        public       transient
case       elses     instanceof    return       try
catch      extend    int           short        void
char       final     interface     static       volatile
class      finally   long          super        while
const      float     native        switch
continue   for       new           synchronized
```

Because their use is restricted, keywords cannot be used as the names of methods, variables, or classes. However, the programmer can make up his or her own names for the classes, methods, and variables that occur in the program, provided that certain rules and conventions are followed.

The names for classes, methods, and variables are called identifiers, which follow certain syntax rules:

JAVA LANGUAGE RULE

Identifier. An identifier must begin with a capital or lowercase letter and may be followed by any number of letters, digits, underscores ( ), or dollar signs ($). An identifier may not be identical to a Java keyword.

Names in Java are *case sensitive*, which means that two different identifiers may contain the same letters in the same order. For example, thisVar and ThisVar are two different identifiers.

In addition to the syntax rule that governs identifiers, Java programmers follow certain style conventions in making up names for classes, variables, and methods. By convention, class names in Java begin with a capital letter and use capital letters to distinguish the individual words in the name—for example, HelloWorld and TextField. Variable and method names begin with a lowercase letter but also use capital letters to distinguish the words in the name—for example, main(), greeting, greet(), getQuestion(), and getAnswer(). The advantage of this convention is that it is easy to distinguish the different elements in a program—classes, methods, variables—just by how they are written. (For more on Java style conventions, see Appendix A.).

Another important style convention followed by Java programmers is to choose descriptive identifiers when naming classes, variables, and methods. This helps to make the program more readable.

JAVA PROGRAMMING TIP

To make your program more readable, choose names that describe the purpose of the class, variable, or method.

Data Types and Variables

A computer program wouldn't be very useful if it couldn't manipulate different kinds of data, such as numbers and strings. The operations that one can do on a piece of data depend on the data's type. For example, you can divide and multiply numbers, but you cannot do this with strings. Thus, every piece of data in a Java program is classified according to its **data type**.

Broadly speaking, there are two categories of data in Java: various types of objects and eight different types of built-in **primitive data types**. In addition to new types of objects that are created by programmers, Java has many different types of built-in objects. Two types that we will encounter in this chapter are the String and PrintStream objects. Java's primitive types include three integer

types, three real number types, a character type, and a boolean type with values true and false. The names of the primitive types are keywords like int for one integer type, double for one real number type, and boolean.

As we noted in Chapter 0[chapter-intro], a variable is a named storage location that can store a value of a particular type. Practically speaking, you can think of a variable as a special container into which you can place values, but only values of a certain type (Fig. 1.6). For example, an int variable can store values like 5 or -100. A String variable can store values like “Hello”. (Actually, this is not the full story, which is a little more complicated, but we will get to that in Chapter 2.)

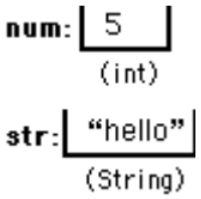


Figure 1.6: Variables are like typed containers.

In the HelloWorld class, the instance variable greeting (line 8) stores a value of type String. In the main() method, the variable helloworld is assigned a HelloWorld object (line 16).

A **literal value** is an actual value of some type that occurs in a program. For example, a string enclosed in double quotes, such as "Hello, World!", is known as a String literal. A number such as 45.2 would be an example of a literal of type double, and -72 would be an example of a literal of type int. Our HelloWorld program contains just a single literal value, the "HelloWorld!" String.

Statements

A Java program is a collection of statements. A **statement** is a segment of code that takes some action in the program. As a program runs, we say it *executes* statements, meaning it carries out the actions specified by those statements. In our HelloWorld program, statements of various types occur on lines 8, 11, 15, 16, and 17. Notice that all of these lines end with a semicolon. The rule in Java is that statements must end with a semicolon. Forgetting to do so would cause a syntax error.

A **declaration statement** is a statement that declares a variable of a particular type. In Java, a variable must be declared before it can be used in a program. Failure to do so would cause a syntax error. In its simplest form, a declaration statement begins with the variable’s type, which is followed by the variable’s name, and ends with a semicolon:

Type VariableName ;

A variable’s type is either one of the primitive types we mentioned, such as int, double, or boolean, or for objects, it is the name of the object’s class, such as String or HelloWorld. A variable’s name may be any legal identifier, as defined earlier, although the convention in Java is to begin variable names with a lowercase letter. In our HelloWorld program, an example a simple declaration statement occurs on line 15:

```
HelloWorld helloworld;
```

This example declares a variable for an object. The variable’s name is helloworld and its type is HelloWorld, the name of the class that is being defined in our example. To take another example the following statements declare two intvariables, named int1 and int2:

```
int int1;
int int2;
```

As we noted, an int is one of Java’s primitive types and the word *int* is a Java keyword.

Without going into too much detail at this point, declaring a variable causes the program to set aside enough memory for the type of data that will be stored in that variable. So in this example, Java would reserve enough space to store an int.

An **assignment statement** is a statement that stores (assigns) a value in a variable. An assignment statement uses the equal sign (=) as an assignment operator. In its simplest form, an assignment statement has a variable on the left hand side of the equals sign and

some type of value on the right hand side. Like other statements, an assignment statement ends with a semicolon:

VariableName = *Value* ;

When it executes an assignment statement, Java will first determine what value is given on the right hand side and then assign (store) that value to (in) the variable on the left hand side. Here are some simple examples:

```
greeting = "Hello, World";
num1 = 50;         // (a) Assign 50 to num1
num2 = 10 + 15;    // (b) Assign 25 to num2
num1 = num2;       // (c) Copy num2's value (25) into num1
```

In the first case, the value on the right hand side is the string literal "Hello, World!", which gets stored in greeting. Of course, greeting has to be the right type of container–in this case, a String variable. In the next case, the value on the right hand side is 50. So that is the value that gets stored in num1, assuming that num1 is an int variable. The situation after this assignment is shown in the top drawing in Figure 1.7. In the third case, the value on the right hand side is 25, which is determined by adding 10 and 15. So the value that gets assigned to num2 is 25. After this assignment we have the situation shown in the middle drawing in the figure. Of course, this assumes that num2 is an intvariable. In the last case, the value on the right hand side is 25, the value that we just stored in the variable num2. So, 25 gets stored in num1. This is the bottom drawing in the accompanying figure.

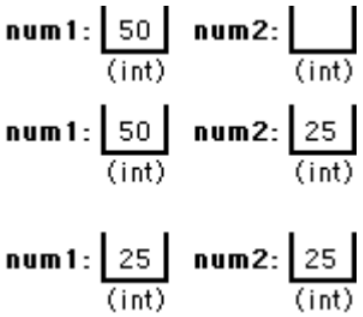


Figure 1.7: This illustrates how the state of the variables num1 and num2 changes over the course of the three assignments, (a), (b), (c), given in the text.

The last of these examples

```
num1 = num2;   // Copy num2's value into num1
```

can be confusing to beginning programmers, so it is worth some additional comment. In this case, there are variables on both the left and right of the assignment operator. But they have very different meaning. The variable on the right is treated as a value. If that variable is storing 25, then that is its value. In fact, whatever occurs on the right hand side of an assignment operator is treated as a value. The variable on the left hand side is treated as a memory location. It is where the value 25 will be stored as a result of executing this statement. The effect of this statement is to copy the value stored in *num2* into *num1*, as illustrated in Figure 1.8.

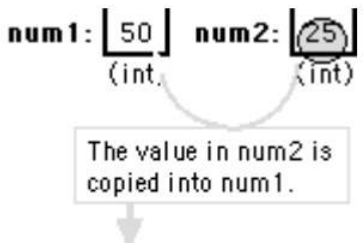


Figure 1.8

Java has many other kinds of statements and we will be learning about these in subsequent examples. The following examples from the HelloWorld program are examples of statements in which a method is called:

```
System.out.println(greeting);// Call println() method
helloworld.greet();          // Call greet() method
```

We will discuss these kinds of statements in greater detail as we go along. One final type of statement that should be mentioned at this point is the **compound statement** (or **block**), which is a sequence of statements contained within braces (). We see three examples of this in the HelloWorld program. The body of a class definition extends from lines 7 through 19. The body of the greet() method is a block that extends from lines 10 through 12. The body of the main()method is a block that extends from lines 14 to 19.

Expressions and Operators

The manipulation of data in a program is done by using some kind of *expression* that specifies the action. An **expression** is Java code that specifies or produces a value in the program. For example, if you want to add two numbers, you would use an arithmetic expression, such as $num1 + num2$. If you want to compare two numbers, you would use a relation expression such as $num1 < num2$. As you can see, these and many other expressions in Java involve the use of special symbols called **operators**. Here we see the addition operator ($+$) and the less-than operator ($<$). We have already talked about the assignment operator ($=$).

Java expressions and operators have a type that depends on the type of data that is being manipulated. For example, when adding two int values, such as $5 + 10$, the expression itself produces an int result. When comparing two numbers with the less than operator, $num1 < num2$, the expression itself produces a boolean type, either true or false.

It is important to note that expressions cannot occur on their own. Rather they occur as part of the program's statements. Here are some additional examples of expressions:

```
num = 7      // An assignment expression of type int
num = square(7) // An method call expression of type int
num == 7     // An equality expression of type boolean
```

The first of these is an assignment expression. It has a value of 7, because it is assigning 7 to num. The second example is also an assignment expression, but this one has a method call, square(7), on its right hand side. (We can assume that a method named square() has been appropriately defined in the program.) A method call is just another kind of expression. In this case, it has the value 49. Note that an assignment expression can be turned into a stand-alone assignment statement by placing a semicolon after it.

The third expression is an equality expression, which has the value true, assuming that the variable on its left is storing the value 7. It is important to note the difference between the assignment operator ($=$) and the equality operator ($==$).

JAVA LANGUAGE RULE

Equality and Assignment. Be careful not to confuse = and ==. The symbol = is the assignment operator. It assigns the value on its right-hand side to the variable on its left-hand side. The symbol == is the equality operator. It evaluates whether the expressions on its left- and right-hand sides have the same value and returns either true or false.

Exercise 1.4.1

What is stored in the variable num after the following two statements are executed?

```
int num = 11;
num = 23 - num;
```

Exercise 1.4.2

Write a statement that will declare a variable of type int called num2, and store in it the sum of 711 and 712.

Class Definition

A Java program consists of one or more class definitions. In the HelloWorld example, we are defining the HelloWorldclass, but there are also three predefined classes involved in the program. These are the Object, String, and Systemclasses all of which are defined in the Java class library. Predefined classes, such as these, can be used in any program.

As the HelloWorld program's comments indicate, a class definition has two parts: a *class header* and a *class body*. In general, a class header takes the following form, some parts of which are optional (*opt*):

$$ClassModifiers_{opt} \quad \texttt{class} \quad ClassName \quad Pedigree_{opt}$$

The class header for the HelloWorld class is:

```
public class HelloWorld extends Object
```

The purpose of the header is to give the class its name (HelloWorld), identify its accessibility (public as opposed to private), and describe where it fits into the Java class hierarchy (as an extension of the Object class). In this case, the header begins with the optional access modifier, public, which declares that this class can be accessed by any other classes. The next part of the declaration identifies the name of the class, HelloWorld. And the last part declares that HelloWorld is a subclass of the Object class. We call this part of the definition the class's pedigree.

As you recall from Chapter 0 [chapter-intro], the Object class is the top class of the entire Java hierarchy. By declaring that HelloWorld extends Object, we are saying that HelloWorld is a direct *subclass* of Object. In fact, it is not necessary to declare explicitly that HelloWorld extends Object because that is Java's default assumption. That is, if you omit the extends clause in the class header, Java will automatically assume that the class is a subclass of Object.

The class's body, which is enclosed within curly brackets (), contains the declaration and definition of the elements that make up the objects of the class. This is where the object's attributes and actions are defined.

Declaring an Instance Variable

There are generally two kinds of elements declared and defined in the class body: variables and methods. As we described in Chapter [chapter-intro], an instance variable is a variable that belongs to each object, or instance, of the class. That is, each instance of a class has its own copies of the class's instance variables. The HelloWorld class has a single instance variable, (greeting), which is declared as follows:

```
private String greeting = "Hello, World!";
```

In general, an instance variable declaration has the following syntax, some parts of which are optional:

$$Modifiers_{opt} \quad Type \quad VariableName \quad InitializerExpression_{opt}$$

Thus, a variable declaration begins with optional modifiers. In declaring the greeting variable, we use the access modifier, private, to declare that greeting, which belongs to the HelloWorld class, cannot be directly accessed by other objects. The next part of the declaration is the variable's type. In this case, the greeting variable is a String, which means that it can store a string object. The type is followed by the name of the variable, in this case (greeting). This is the name that is used to refer to this memory location throughout the class. For example, notice that the variable is referred to on line 11 where it is used in a println() statement.

The last part of the declaration is an optional initializer expression. In this example, we use it to assign an initial value, "Hello, World!," to the greeting variable.

Defining an Instance Method

Recall that a method is a named section of code that can be called or invoked to carry out an action or operation. In a Java class, the methods correspond to the object's behaviors or actions. The HelloWorld program has two method definitions: the greet() method and the main() method.

A method definition consists of two parts: the method header and the method body. In general, a method header takes the following form, including some parts which are optional:

$$Modifiers \quad ReturnType \quad MethodName \quad (\quad ParameterList \quad)$$

$Modifiers_{opt}$ $ReturnType$ $MethodName$ ($ParameterList_{opt}$)

As with a variable declaration, a method definition begins with optional modifiers. For example, the definition of the greet() method on line 9 uses the access modifier, public, to declare that this method can be accessed or referred to by other classes. The main() method, whose definition begins on line 13, is a special method, and is explained in the next section.

The next part of the method header is the method's return type. This is the type of value, if any, that the method returns. Both of the methods in HelloWorld have a return type of void. This means that they don't return any kind of value. Void methods just execute the sequence of statements given in their bodies. For an example of a method that does return a value, take a look again at the declaration of the getQuestion() method in the Riddle class, which returns a String(Fig. 1.4).

The method's name follows the method's return type. This is the name that is used when the method is called. For example, the greet() method is called on line 17.

Following the method's name is the method's parameter list. A **parameter** is a variable that temporarily stores data values that are being passed to the method when the method is called. Some methods, such as the greet() method, do not have parameters, because they are not passed any information. For an example of a method that does have parameters, see the Riddle() constructor, which contains parameters for the riddle's question and answer (Fig. 1.4).

The last part of method definition is its body, which contains a sequence of executable statements. An **executable statement** is a Java statement that takes some kind of action when the program is run. For example, the statement in the greet() method,

```
System.out.println(greeting);   //  Output statement
```

prints a greeting on the console.

Java Application Programs

The HelloWorld program is an example of a Java **application program**, or a Java application, for short. An application program is a stand-alone program, "stand-alone" in the sense that it does not depend on any other program, like a Web browser, for its execution. Every Java application program must contain a main() method, which is where the program begins execution when it is run. For a program that contains several classes, it is up to the programmer to decide which class should contain the main() method. We don't have to worry about that decision for the HelloWorld, because it contains just a single class.

Because of its unique role as the starting point for every Java application program, it is very important that the header for the main method be declared exactly as shown in the HelloWorld class:

```
public static void main(String args[])
```

It must be declared public so it can be accessed from outside the class that contains it. The static modifier is used to designate main() as a class method. As you might recall from Chapter 0, a class method is a method that is associated directly with the class that contains it rather than with the objects of the class. A class method is not part of the class's objects. Unlike instance methods, which are invoked through a class's objects, a class method is called through the class itself. Thus, a class method can be called even before the program has created objects of that class. Because of main()'s special role as the program's starting point, it is necessary for main() to be a class method because it is called, by the Java runtime system, before the program has created any objects.

The main() method has a void return type, which means it does not return any kind of value. Finally, notice that main()'s parameter list contains a declaration of some kind of String parameter named *args*. This is actually an array that can be used to pass string arguments to the program when it is started up. We won't worry about this feature until our chapter on arrays.

Creating and Using Objects

The body of the main() method is where the HelloWorld program creates its one and only object. Recall that when it is run the HelloWorld program just prints the "Hello World!" greeting. As we noted earlier, this action happens in the greet() method. So in order to make this action happen, we need to call the greet() method. However, because the greet() method is an instance method that belongs to a HelloWorld object, we first need to create a HelloWorld instance. This is what happens in the body of the main() method (Fig. 1.5).

The main() method contains three statements:

```
HelloWorld helloworld;          // Variable declaration
helloworld = new HelloWorld();  // Object instantiation
helloworld.greet();             // Method invocation
```

The first statement declares a variable of type HelloWorld, which is then assigned a HelloWorld object. The second statement creates a HelloWorld object. This is done by invoking the HelloWorld() constructor method. Creating an object is called **object instantiation** because you are creating an instance of the object. Once a HelloWorld instance is created, we can use one of its instance methods to perform some task or operation. Thus, in the third statement, we call the greet() method, which will print "Hello World!" on the console.

If you look back at the HelloWorld program in Figure [fig:helloworld] you won't find a definition of a constructor method. This is not an error because Java will provide a default constructor if a class does not contain a constructor definition. The **default constructor** is a trivial constructor method, "trivial" because its body contains no statements. Here is what the default HelloWorld() constructor would look like:

```
public HelloWorld() {  }  // Default constructor
```

For most of the classes we design, we will design our own constructors, just as we did in the Riddle class (Fig. [fig:riddleclass]). We will use constructors to assign initial values to an object's instance variables or to perform other kinds of tasks that are needed when an object is created. Because the HelloWorld object doesn't require any startup tasks, we can make do with the default constructor.

The HelloWorld program illustrates the idea that an object-oriented program is a collection of interacting objects. Although we create just a single HelloWorld object in the main() method, there are two other objects used in the program. One is the greeting, which is a String object consisting of the string "Hello, World!". The other is the System.out object, which is a special Java system object used for printing.

Java JFrames

Java cann run a program in a **JFrame** so that the output and interaction occurs in a Window (or Frame). Figure 1.9 shows a Java program named HelloWorldSwing. This program does more or less the same thing as the HelloWorld application—it displays the "Hello, World!" greeting.

```
 /** File: HelloWorldSwing program */

import javax.swing.JFrame; // Import class names
import java.awt.Graphics;
import java.awt.Canvas;

public class HelloWorldCanvas extends Canvas // Class header
{
    // Start of body
    public void paint(Graphics g)
        // The paint method
    {
        g.drawString("Hello, World!", 10, 10);
    }  // End of paint

    public static void main(String[] args){
        HelloWorldCanvas c = new HelloWorldCanvas();
        JFrame f = new JFrame();
        f.add(c);
        f.setSize(150,50);
        f.setVisible(true);
    }
}  // End of HelloWorldCanvas
```

Figure 1.9: HelloWorldCanvas program.

The difference is that it displays the greeting within a Window rather than directly on the console.

As in the case of the HelloWorld console application program, HelloWorldCanvas consists of a class definition. It contains a single method definition, the paint() method, which contains a single executable statement:

```
g.drawString("Hello, World!",10,10);
```

This statement displays the “Hello, World!” message directly in a Window. The drawString() method is one of the many drawing and painting methods defined in the Graphics class. Every Java Canvas comes with its own Graphicsobject, which is referred to here simply as g. Thus, we are using that object’s drawString() method to draw on the window. Don’t worry if this seems a bit mysterious now. We’ll explain it more fully when we take up graphics examples again.

The HelloWorldSwing also contains some elements, such as the import statements, that we did not find in the HelloWorld application. We will now discuss those features.

Java Library Packages

Recall that the HelloWorld application program used two pre-defined classes, the String and the System classes. Both of these classes are basic language classes in Java. The HelloWorldSwing program also uses pre-defined classes, such as JFrame and Graphics. However, these two classes are not part of Java’s basic language classes. To understand the difference between these classes, it will be necessary to talk briefly about how the Java class library is organized.

A **package** is a collection a inter-related classes in the Java class library. For example, the java.lang package contains classes, such as Object, String, and System, that are central to the Java language. Just about all Java programs use classes in this package. The java.awt package provides classes, such as Button, TextField, and Graphics, that are used in graphical user interfaces (GUIs). The java.net package provides classes used for networking tasks, and the java.iopackage provides classes used for input and output operations.

All Java classes belong to some package, including those that are programmer defined. To assign a class to a package, you would provide a package statement as the first statement in the file that contains the class definition. For example, the files containing the

definitions of the classes in the java.lang package all begin with the following statement.

```
package java.lang;
```

If you omit package statement, as we do for the programs in this book, Java places such classes into an unnamed default package.

Thus, for any Java class, its full name includes the name of the package that contains it. For example, the full name for the System class is java.lang.System and the full name for the String class is java.lang.String. Similarly, the full name for the Graphics class is java.awt.Graphics. In short, the full name for a Java class takes the following form:

$$package.class$$

In other words, the full name of any class provides its package name as a prefix.

Of all the packages in the Java library, the java.lang package is the only one whose classes are available by their shorthand names to all Java programs. This means that when a program uses a class from the java.lang package, it can refer to it simply by its class name. For example, in the HelloWorld program we referred directly to the String class rather than to java.lang.String.

The import Statement

The import statement makes Java classes available to programs under their abbreviated names. Any public class in the Java class library is available to a program by its fully qualified name. Thus, if a program was using the Graphics class, it could always refer to it as java.awt.Graphics. However, being able to refer to Graphics by its shorthand name, makes the program a bit shorter and more readable.

The import statement doesn't actually load classes into the program. It just makes their abbreviated names available. For example, the import statements in HelloWorldSwing allow us to refer to the JFrame, Canvas, and Graphics classes by their abbreviated names (Fig. 1.9).

The import statement takes two possible forms:

$$\texttt{import} \quad package.class$$

$$\texttt{import} \quad package.*$$

The first form allows a specific class to be known by its abbreviated name. The second form, which uses the asterisk as a wildcard characters ('*'), allows all the classes in the specified package to be known by their short names. The importstatements in HelloWorldSwing are examples of the first form. The following example,

```
import java.lang.*;
```

allows all classes in the java.lang package to be referred to by their class names alone. In fact, this particular importstatement is implicit in every Java program.

Qualified Names in Java

In the previous subsections we have seen several examples of names in Java programs that used *dot notation*. A **qualified name** is a name that is separated into parts using Java's dot notation. Examples include package names, such as java.awt, class names, such as javax.swing.JFrame, and even method names, such as helloworld.greet().

Just as in our natural language, the meaning of a name within a Java program depends on the context. For example, the expression helloworld.greet() refers to the greet() method, which belongs to the HelloWorld class. If we were using this expression from within that class, you wouldn't need to qualify the name in this way. You could just refer to greet() and it would be clear from the context which method you meant.

This is no different than using someone's first name ("Kim") when there's only one Kim around, but using a full name ("Kim Smith") when the first name alone would be too vague or ambiguous.

One thing that complicates the use of qualified names is that they are used to refer to different kinds of things within a Java program. But this is no different, really, than in our natural language, where names ("George Washington") can refer to people,

bridges, universities, and so on. Here again, just as in our natural language, Java uses the context to understand the meaning of the name. For example, the expression java.lang.System refers to the System class in the java.lang package, whereas the expression System.out.print() refers to a method in the System.out object.

How can you tell these apart? Java can tell them apart because the first one occurs as part of an import statement, so it must be referring to something that belongs to a package. The second expression would only be valid in a context where a method invocation is allowed. You will have to learn a bit more about the Java language before you'll be able to completely understand these names, but the following provide some naming rules to get you started.

JAVA LANGUAGE RULE

Library Class Names. By convention, class names in Java begin with an uppercase letter. When referenced as part of a package, the class name is the last part of the name. For example, java.lang.System refers to the System class in the java.lang package.

JAVA LANGUAGE RULE

Dot Notation. Names expressed in Java's dot notation depend for their meaning on the context in which they are used. In qualified names—that is, names of the form X.Y.Z—the last item in the name (Z) is the referent—that is, the element being referred to. The items that precede it (X.Y.) are used to qualify or clarify the referent.

The fact that names are context dependent in this way certainly complicates the task of learning what's what in a Java program. Part of learning to use Java's built-in classes is learning where a particular object or method is defined. It is a syntax error if the Java compiler can't find the object or method that you are referencing.

JAVA DEBUGGING TIP

Not Found Error. If Java cannot find the item you are referring to, it will report an "X not found" error, where X is the class, method, variable, or package being referred to.

This page titled 1.4: Java Language Elements is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **1.4: Java Language Elements** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

1.5: Editing, Compiling, and Running a Java Program

In this section we discuss the nuts and bolts of how to compile and run a Java program. Because we are exploring two different varieties of Java programs, console applications and Swing applications, the process differs slightly for each variety. We have already discussed some of the main language features of console and Swing applications, so in this section we focus more on features of the programming environment itself. Because we do not assume any particular programming environment in this book, our discussion will be somewhat generic. However, we do begin with a brief overview of the types of programming environments one might encounter.

Java Development Environments

A Java programming environment typically consists of several programs that perform different tasks required to edit, compile, and run a Java program. The following description will be based on the software development environment provided by Oracle, the company that owns and maintains Java. It is currently known as the *Java Platform, Standard Edition 8.0 (Java SE 8)*. Versions of Java SE are available for various platforms, including Linux, Windows, and macOS computers. Free downloads are available at Sun's Web site at http://www.oracle.com/technetwork/java/. (For more details about the Java SE, see Appendix B.)

In some cases, the individual programs that make up the Java SE are available in a single program development environment, known as an *integrated development environment (IDE)*. Some examples include Eclipse, jGrasp, and Oracle's own NetBeans IDE. Each of these provides a complete development package for editing, compiling, and running Java applications on a variety of platforms, including Linux, macOS, and Windows.

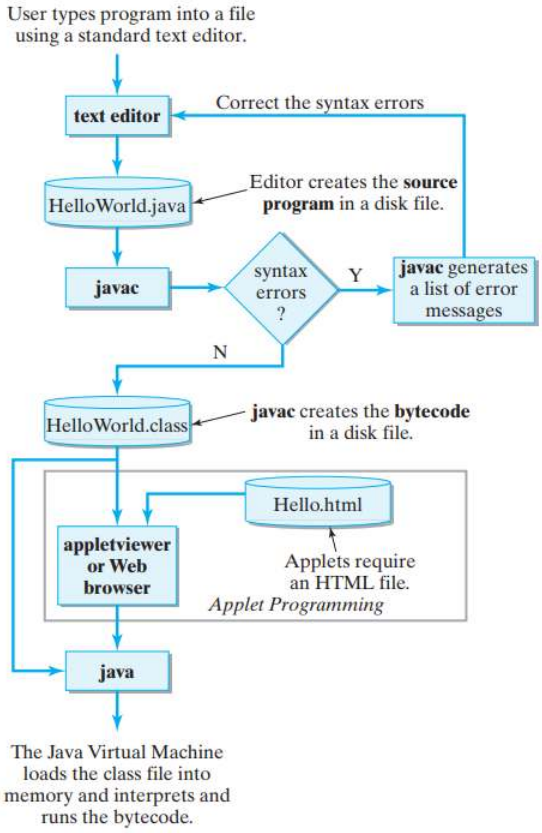


Figure 1.10: Editing, Compiling, and Running HellloWorld.java.

Figure 1.10 illustrates the process involved in creating and running a Java program. The discussion that follows here assumes that you are using the Java SE as your development environment to edit, compile and run the example program. If you are using some other environment, you will need to read the documentation provided with the software to determine exactly how to edit, compile, and run Java programs in that environment.

Editing a Program

Any text editor may be used to edit the program by merely typing the program and making corrections as needed. Popular Unix and Linux editors include vim and emacs. These editors are also available on macOS and Windows. However, free macOS editors include TextMate and TextWrangler, and Windows has Notepad++ for free.

As we have seen, a Java program consists of one or more class definitions. We will follow the convention of placing each class definition in its own file. (The rule in Java is that a source file may contain only one public class definition.) The files containing these classes' definitions must be named *ClassName.java* where *ClassName* is the name of the public Java class contained in the file.

JAVA LANGUAGE RULE

File Names. A file that defines a public Java class named ClassName must be saved in a text file named ClassName.java. Otherwise an error will result.

For example, in the case of our HelloWorld application program, the file must be named HelloWorld.java, and for HelloWorldSwing, it must be named HelloWorldSwing.java. Because Java is *case sensitive*, which means that Java pays attention to whether a letter is typed uppercase or lowercase, it would be an error if the file containing the HelloWorld class were named helloworld.java or Helloworld.java. The error in this case would be a semantic error. Java would not be able to find the HelloWorld class because it will be looking for a file named HelloWorld.java.

JAVA LANGUAGE RULE

Case Sensitivity. Java is case sensitive, which means that it treats helloWorld and Helloworld as different names.

Compiling a Program

Recall that before you can run a Java source program you have to compile it into the Java bytecode, the intermediate code understood by the Java Virtual Machine (JVM). Source code for both applets and applications must be compiled. To run a Java program, whether an applet or an application, the JVM is then used to interpret and execute the bytecode.

The Java SE comes in two parts, a runtime program, called the *Java Runtime Environment (JRE)* and a development package, called the *Software Development Kit (SDK)*. If you are just going to run Java programs, you need only install the JRE on your computer. In order to run Java applets, browsers, such as Internet Explorer and Netscape Navigator, must contain a plugin version of the JRE. On the other hand, if you are going to be developing Java programs, you will need to install the SDK as well.

The Java SDK compiler is named javac. In some environments—such as within Linux or at the Windows command prompt — HelloWorld.java would be compiled by typing the following command at the system prompt:

```
javac HelloWorld.java
```

As Figure 1.10 illustrates, if the HelloWorld.java program does not contain errors, the result of this command is the creation of a Java bytecode file named HelloWorld.class—a file that has the same prefix as the source file but with the suffix .class rather than .java. By default, the bytecode file will be placed in the same directory as the source file. If javac detects errors in the Java code, a list of error messages will be printed.

Running a Java Application Program

In order to run (or execute) a program on any computer, the program's *executable code* must be loaded into the computer's main memory. For Java environments, this means that the program's .class file must be loaded into the computer's memory, where it is then interpreted by the Java Virtual Machine. To run a Java program on Linux systems or at the Windows command prompt, type

```
java HelloWorld
```

on the command line. This command loads the JVM, which will then load and interpret the application's bytecode(HelloWorld.class). The "HelloWorld" string will be displayed on the command line.

On Macintosh systems, or within an IDE, which do not typically have a command line interface, you would select the compile and run commands from a menu. Once the code is compiled, the run command will cause the JVM to be loaded and the bytecode to be interpreted. The “Hello, World!” output would appear in a text-based window that automatically pops up on your computer screen. In any case, regardless of the system you use, running the HelloWorldconsole application program will cause the “Hello, World!” message to be displayed on some kind of standard output device (Fig. 1.11).



Figure 1.11: Compiling and Running the HelloWorld.java console application program.

Running a Java Swing Program

When you run a Java Swing Program, there is typically no console output. You only see your output in the Window (JFrame) that your Graphics are displayed in. This makes automated testing more difficult since you need to visually inspect that the program is working correctly.

When you run

```
java HelloWorldSwing
```

A window will open, and you won’t be able to type in the console until you close the window, quit the program, or type ctl-c to send a kill signal to the Swing program. The result of running, as shown in Figure 1.12, is that the “Hello, World!” message will be displayed within it’s own window.



Figure 1.12: Running HelloWorldSwing.java graphical program.

This page titled 1.5: Editing, Compiling, and Running a Java Program is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **1.5: Editing, Compiling, and Running a Java Program** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

1.6: From the Java Library- System and PrintStream

Java comes with a library of classes that can be used to perform common tasks. The Java class library is organized into a set of packages, where each package contains a collection of related classes. Throughout the book we will identify library classes and explain how to use them. In this section we introduce the System and PrintStream classes, which are used for printing a program's output.

Java programs need to be able to accept input and to display output. Deciding how a program will handle input and output (I/O) is part of designing its *user interface*, a topic we take up in detail in Chapter 4. The simplest type of user interface is a *command-line interface*, in which input is taken from the command line through the keyboard, and output is displayed on the console. Some Java applications use this type of interface. Another type of user interface is a *Graphical User Interface (GUI)*, which uses buttons, text fields, and other graphical components for input and output. Java applets use GUIs as do many Java applications. Because we want to be able to write programs that generate output, this section describes how Java handles simple console output.

In Java, any source or destination for I/O is considered a *stream* of bytes or characters. To perform output, we insert bytes or characters into the stream. To perform input, we extract bytes or characters from the stream. Even characters entered at a keyboard, if considered as a sequence of keystrokes, can be represented as a stream.

There are no I/O statements in the Java language. Instead, I/O is handled through methods that belong to classes contained in the java.io package. We have already seen how the output method println() is used to output a string to the console. For example, the following println() statement

```
System.out.println("Hello, World");
```

prints the message "Hello, World" on the Java console. Let's now examine this statement more carefully to see how it makes use of the Java I/O classes.

The java.io.PrintStream class is Java's printing expert, so to speak. It contains a variety of print() and println() methods that can be used to print all of the various types of data we find in a Java program. A partial definition of PrintStream is shown in Figure 1.13. Note A UML class diagram of the PrintStream class. fig:printstreamUML that in this case the PrintStream class has no attributes, just operations or methods.

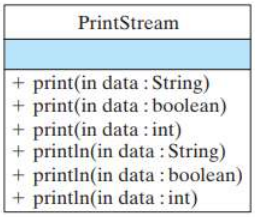


Figure 1.13: A UML class diagram of the PrintStream class.

Because the various print() and println() methods are instance methods of a PrintStream object, we can only use them by finding a PrintStream object and "telling" it to print data for us. As shown in Figure 1.15, Java's java.lang.Systemclass contains three predefined streams, including two PrintStream objects. This class has public (+) attributes. None of its public methods are shown here.

Both the System.out and System.err objects can be used to write output to the console. As its name suggests, the errstream is used primarily for error messages, whereas the out stream is used for other printed output. Similarly, as its name suggests, the System.in object can be used to handle input, which will be covered in Chapter 2.

The only difference between the print() and println() methods is that println() will also print a carriage return and line feed after printing its data, thereby allowing subsequent output to be printed on a new line. For example, the following statements

```
System.out.print("hello");
System.out.println("hello again");
System.out.println("goodbye");
```

would produce the following output:

```
hellohello again
goodbye
```

Now that we know how to use Java's printing expert, let's use it to "sing" a version of "Old MacDonald Had a Farm." As you might guess, this program will simply consist of a sequence of System.out.println() statements each of which prints a line of the verse. The complete Java application program is shown in Figure 1.15.

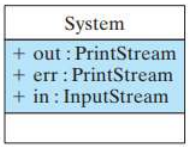


Figure 1.14: The System class.

```
public class OldMacDonald
{
   public static void main(String args[])   // Main method
   {
     System.out.println("Old MacDonald had a farm");
     System.out.println("E I E I O.");
     System.out.println("And on his farm he had a duck.");
     System.out.println("E I E I O.");
     System.out.println("With a quack quack here.");
     System.out.println("And a quack quack there.");
     System.out.println("Here a quack, there a quack,");
     System.out.println("Everywhere a quack quack.");
     System.out.println("Old MacDonald had a farm");
     System.out.println("E I E I O.");
   }  // End of main
}  // End of OldMacDonald
```

Figure 1.15: The OldMacDonald.java class.

This example illustrates the importance of using the Java class library. If there's a particular task we want to perform, one of the first things we should ask is whether there is already an "expert" in Java's class library that performs that task. If so, we can use methods provided by the expert to perform that particular task.

JAVA EFFECTIVE DESIGN

Using the Java Library. Learning how to use classes and objects from the Java class library is an important part of object-oriented programming in Java.

Exercise 1.6.1

One good way to learn how to write programs is to modify existing programs. Modify the OldMacDonald class to "sing" one more verse of the song.

Exercise 1.6.2

Write a Java class that prints the design shown below.



Chapter summary

algorithm

applet

application program

assignment statement

comment

compound statement (block)

data type

declaration statement

default constructor

executable statement

expression

identifier

literal value

object instantiation

operator

package

parameter

primitive data type

pseudocode

qualified name

semantics

statement

stepwise refinement

syntax

Good program design requires that each object and method have a well-defined role and clear definition of what information is needed for the task and what results will be produced.

Good program design is important; the sooner you start coding, the longer the program will take to finish. Good program design strives for readability, clarity, and flexibility.

Testing a program is very important and must be done with care, but it can only reveal the presence of bugs, not their absence.

An algorithm is a step-by-step process that solves some problem. Algorithms are often described in pseudocode, a hybrid language that combines English and programming language constructs.

A syntax error occurs when a statement breaks a Java syntax rules. Syntax errors are detected by the compiler. A semantic error is an error in the program's design and cannot be detected by the compiler.

Writing Java code should follow the stepwise refinement process.

Double slashes (//) are used to make a single-line comment. Comments that extend over several lines must begin with /* and end with */.

An *identifier* must begin with a letter of the alphabet and may consist of any number of letters, digits, and the special characters _ and $. An identifier cannot be identical to a Java keyword. Identifiers are case sensitive.

A *keyword* is a term that has special meaning in the Java language (Table 1.1).

Examples of Java's *primitive data types* include the int, boolean, and double types.

A variable is a named storage location. In Java, a variable must be declared before it can be used.

A literal value is an actual value of some type, such as a String ("Hello") or an int (5).

A declaration statement has the form: ;

An assignment statement has the form: = ; When it is executed it determines the value of the *Expression* on the right of the assignment operator (=) and stores the value in the variable named on the left.

Java's operators are type dependent, where the type is dependent on the data being manipulated. When adding two intvalues (7 + 8), the + operation produces an int result.

A class definition has two parts: a class header and a class body. A class header takes the form of optional modifiers followed by the word class followed by an identifier naming the class followed, optionally, by the keyword extends and the name of the class's superclass.

There are generally two kinds of elements declared and defined in the class body: variables and methods.

Object instantiation is the process of creating an instance of a class using the new operator in conjunction with one of the class's constructors.

Dot notation takes the form *qualifiers.elementName*. The expression System.out.print("hello") uses Java dot notation to invoke the print() method of the System.out object.

A Java application program runs in stand-alone mode. A Java applet is a program that runs within the context of a Java-enabled browser. Java applets are identified in HTML documents by using the tag.

A Java source program must be stored in a file that has a .java extension. A Java bytecode file has the same name as the source file but a .class extension. It is an error in Java if the name of the source file is not identical to the name of the public Java class defined within the file.

Java programs are first compiled into bytecode and then interpreted by the Java Virtual Machine (JVM).

The value 12 is stored in num.

int num2 = 711 + 712;

The definition of the OldMacDonald class is:

```
public class OldMacDonald
{
   public static void main(String args[])   // Main method
   {
     System.out.println("Old MacDonald had a farm");
     System.out.println("E I E I O.");
     System.out.println("And on his farm he had a duck.");
     System.out.println("E I E I O.");
     System.out.println("With a quack quack here.");
     System.out.println("And a quack quack there.");
     System.out.println("Here a quack, there a quack,");
     System.out.println("Everywhere a quack quack.");
     System.out.println("Old MacDonald had a farm");
     System.out.println("E I E I O.");

     System.out.println("Old MacDonald had a farm");
     System.out.println("E I E I O.");
     System.out.println("And on his farm he had a pig.");
     System.out.println("E I E I O.");
     System.out.println("With an oink oink here.");
     System.out.println("And an oink oink  there.");
     System.out.println("Here an oink, there an oink,");
     System.out.println("Everywhere an oink oink.");
     System.out.println("Old MacDonald had a farm");
     System.out.println("E I E I O.");
   }  // End of main
}  // End of OldMacDonald
```

The definition of the Pattern class is:

```
public class Pattern
{
     public static void main(String args[])// Main method
     {
         System.out.println("**********");
         System.out.println("* **  ** *");
         System.out.println("*   **   *");
         System.out.println("* *    * *");
         System.out.println("*  ****  *");
         System.out.println("**********");
     }  // End of main
}  // End of Pattern
```

Fill in the blanks in each of the following statements.

=14pt

A Java class definition contains an object’s

_____

and

_____

.

A method definition contains two parts, a

_____

and a

.

=11pt

Explain the difference between each of the following pairs of concepts.

Application and *applet*.

Single-line and *multiline* comment.

Compiling and *running* a program.

Source code file and *bytecode* file.

Syntax and *semantics*.

Syntax error and *semantic error*.

Data and *methods*.

Variable and *method*.

Algorithm and *method*.

Pseudocode and *Java code*.

Method definition and *method invocation*.

For each of the following, identify it as either a syntax error or a semantic error. Justify your answers.

Write a class header as public Class MyClass.

Define the init() header as public vid init().

Print a string of five asterisks by System.out.println("***");.

Forget the semicolon at the end of a println() statement.

Calculate the sum of two numbers as N – M.

Suppose you have a Java program stored in a file named Test.java. Describe the compilation and execution process for this program, naming any other files that would be created.

Suppose *N* is 15. What numbers would be output by the following pseudocode algorithm? Suppose *N* is 6. What would be output by the algorithm in that case?

```
0. Print N.
1. If N equals 1, stop.
2. If N is even, divide it by 2.
3. If N is odd, triple it and add 1.
4. Go to step 0.
```

Suppose *N* is 5 and *M* is 3. What value would be reported by the following pseudocode algorithm? In general, what quantity does this algorithm calculate?

```
0. Write 0 on a piece of paper.
1. If M equals 0, report what's on the paper and stop.
2. Add N to the quantity written on the paper.
3. Subtract 1 from M.
4. Go to step 1.
```

Puzzle Problem: You are given two different length ropes that have the characteristic that they both take exactly one hour to burn. However, neither rope burns at a constant rate. Some sections of the ropes burn very fast; other sections burn very slowly. All you

have to work with is a box of matches and the two ropes. Describe an algorithm that uses the ropes and the matches to calculate when exactly 45 minutes have elapsed.

Puzzle Problem: A polar bear that lives right at the North Pole can walk due south for one hour, due east for one hour, and due north for one hour, and end up right back where it started. Is it possible to do this anywhere else on earth? Explain.

Puzzle Problem: Lewis Carroll, the author of *Alice in Wonderland*, used the following puzzle to entertain his guests: A captive queen weighing 195 pounds, her son weighing 90 pounds, and her daughter weighing 165 pounds, were trapped in a very high tower. Outside their window was a pulley and rope with a basket fastened on each end. They managed to escape by using the baskets and a 75-pound weight they found in the tower. How did they do it? The problem is that anytime the difference in weight between the two baskets is more than 15 pounds, someone might get hurt. Describe an algorithm that gets them down safely.

Puzzle Problem: Here's another Carroll favorite: A farmer needs to cross a river with his fox, goose, and a bag of corn. There's a rowboat that will hold the farmer and one other passenger. The problem is that the fox will eat the goose if they are left alone on the river bank, and the goose will eat the corn if they are left alone on the river bank. Write an algorithm that describes how he got across without losing any of his possessions.

Puzzle Problem: Have you heard this one? A farmer lent the mechanic next door a 40-pound weight. Unfortunately, the mechanic dropped the weight and it broke into four pieces. The good news is that, according to the mechanic, it is still possible to use the four pieces to weigh any quantity between one and 40 pounds on a balance scale. How much did each of the four pieces weigh? (*Hint*: You can weigh a 4-pound object on a balance by putting a 5-pound weight on one side and a 1-pound weight on the other.)

Suppose your little sister asks you to show her how to use a pocket calculator so that she can calculate her homework average in her science course. Describe an algorithm that she can use to find the average of 10 homework grades.

A Caesar cipher is a secret code in which each letter of the alphabet is shifted by *N* letters to the right, with the letters at the end of the alphabet wrapping around to the beginning. For example, if *N* is 1, when we shift each letter to the right, the word *daze* would be written as *ebaf*. Note that the *z* has wrapped around to the beginning of the alphabet. Describe an algorithm that can be used to create a Caesar encoded message with a shift of 5.

Suppose you received the message, "sxccohv duh ixq," which you know to be a Caesar cipher. Figure out what it says and then describe an algorithm that will always find what the message said regardless of the size of the shift that was used.

Suppose you're talking to your little brother on the phone and he wants you to calculate his homework average. All you have to work with is a piece of chalk and a very small chalkboard—big enough to write one four-digit number. What's more, although your little brother knows how to read numbers, he doesn't know how to count very well so he can't tell you how many grades there are. All he can do is read the numbers to you. Describe an algorithm that will calculate the correct average under these conditions.

Write a *header* for a public applet named SampleApplet.

Write a *header* for a public method named getName.

Design a class to represent a geometric rectangle with a given length and width, such that it is capable of calculating the area and the perimeter of the rectangle.

Modify the OldMacDonald class to "sing" either "Mary Had a Little Lamb" or your favorite nursery rhyme.

Define a Java class, called Patterns, modeled after OldMacDonald, that will print the following patterns of asterisks, one after the other heading down the page:

```
*****      *****    *****
 ****      *   *    * * *
  ***      *   *     * *
   **      *   *    * * *
    *      *****    *****
```

Write a Java class that prints your initials as block letters, as shown in the example in the margin.

Challenge: Define a class that represents a Temperature object. It should store the current temperature in an instance variable of type double, and it should have two public methods, setTemp(double t), which assigns t to the instance variable, and getTemp(), which returns the value of the instance variable. Use the Riddle class as a model.

Challenge: Define a class named TaxWhiz that computes the sales tax for a purchase. It should store the current tax rate as an instance variable. Following the model of the Riddle class, you can initialize the rate using a TaxWhiz()method. This class should have one public method, calcTax(double purchase), which returns a double, whose value is purchases times the tax rate. For example, if the tax rate is 4 percent, 0.04, and the purchase is $100, then calcTax()should return 4.0.

What is stored in the variables num1 and num2 after the following statements are executed?

```
int num1 = 5;
int num2 = 8;
num1 = num1 + num2;
num2 = nmm1 + num2;
```

Write a series of statements that will declare a variable of type int called num and store in it the difference between 61 and 51.

Modify the UML diagram of the Riddle class to contain a method named getRiddle() that would return both the riddle's question and answer.

Draw a UML class diagram representing the following class: The name of the class is Circle. It has one attribute, a radius that is represented by a double value. It has one operation, calculateArea(), which returns a double. Its attributes should be designated as private and its method as public.

To represent a triangle we need attributes for each of its three sides and operations to create a triangle, calculate its area, and calculate its perimeter. Draw a UML diagram to represent this triangle.

Try to give the Java class definition for the class described in The Person class. fig:person

the UML diagram shown in Figure 1.17.

This page titled 1.6: From the Java Library- System and PrintStream is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **1.6: From the Java Library- System and PrintStream** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

CHAPTER OVERVIEW

2: Objects- Defining, Creating, and Using

Learning Objectives

After studying this chapter, you will

- Be familiar with using variables to store and manipulate simple data.
- Be familiar with creating and using objects.
- Understand the relationship between classes and objects.
- Understand the difference between objects and data of primitive type.
- Understand the difference between static and and instance elements of a class.
- Be able to understand and design a simple class in Java.
- Understand some of the basic principles of object-oriented programming.

2.1: Introduction
2.2: Using String Objects
2.3: Drawing Shapes with the Graphics Object (Optional)
2.4: Class Definition
2.5: Case Study- Simulating a Two-Person Game
2.6: From the Java Library- java.util.Scanner

This page titled 2: Objects- Defining, Creating, and Using is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

2.1: Introduction

This chapter introduces some more of the basic principles of object-oriented programming. We begin by looking at some examples of creating and using objects of type String and Graphics. Then, we examine how user defined classes are used by doing a detailed walk-through of the Riddle class we saw in Chapter 1. We focus on the basic Java language elements involved. By the end of these sections, you should know how to identify the key elements that make up a Java program.

We then present a detailed example of the programming development process by designing a class that models a certain two person game and implements the class. The design is represented using UML notation.

2.1: Introduction is shared under a CC BY license and was authored, remixed, and/or curated by LibreTexts.

- **2: Objects- Defining, Creating, and Using** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

2.2: Using String Objects

As we know, a Java program is a collection of interacting objects, where each object is a module that encapsulates a portion of the program's attributes and actions. Objects belong to classes, which serve as templates or blueprints for creating objects. Think again of the cookie cutter analogy. A class is like a cookie cutter. Just as a cookie cutter is used to shape and create individual cookies, a class definition is used to shape and create individual objects.

Programming in Java is primarily a matter of designing and defining class definitions, which are then used to construct objects. The objects perform the program's desired actions. To push the cookie cutter analogy a little further, designing and defining a class is like building the cookie cutter. Obviously, very few of us would bake cookies if we first had to design and build the cookie cutters. We'd be better off using a pre-built cookie cutter. By the same token, rather than designing our own classes, it will be easier to get into "baking" programs if we begin by using some predefined Java classes.

The Java library contains many pre-defined classes that we will use in our programs. So let's begin our study of programming by using two of these classes, the String and Graphics classes.

Creating and Combining Strings

Strings are very useful objects in Java and in all computer programs. They are used for inputting and outputting all types of data. Therefore, it essential that we learn how to create and use String objects.

Figure 2.1 provides an overview of a very small part of Java's String class. In addition to the two String() constructor methods, which are used to create strings, it lists several useful instance methods that can be used to manipulate strings. The String class also has two instance variables. One stores the String's *value*, which is a string of characters such as "Hello98", and the other stores the String's *count*, which is the number of characters in its string value.

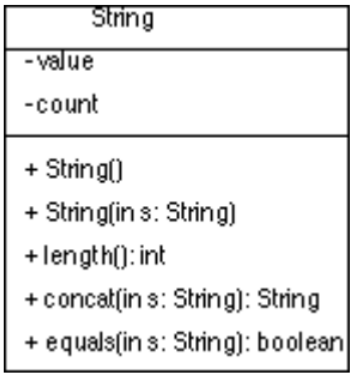


Figure 2.1: A partial representation of the String class.

Recall from Chapter 0 that in order to get things done in a program we send messages to objects. The messages must correspond to the object's instance methods. Sending a message to an object is a matter of calling one of its instance methods. In effect, we use an object's methods to get the object to perform certain actions for us. For example, if we have a String, named str and we want to find out how many characters it contains, we can call its length() method, using the expression str.length(). If we want to print str's length, we can embed this expression in a print statement:

```
System.out.println(str.length()); // Print str's length
```

In general, to use an object's instance method, we refer to the method in dot notation by first naming the object and then the method:

objectName.methodName() ;

The *objectName* refers to a particular object, and the methodName() refers to one of its instance methods.

As this example makes clear, instance methods belong to objects, and in order to use a method, you must first have an object that has that method. So, to use one of the String methods in a program, we must first create a String object.

To create a String object in a program, we first declare a String variable.

```
String str; // Declare a String variable named str
```

We then create a String object by using the new keyword in conjunction with one of the String() constructors. We assign the new object to the variable we declared:

```
str = new String("Hello");// Create a String object
```

This example will create a String that contains, as its value, the word "Hello" that is passed in by the constructor. The String object that this creates is shown in Figure 2.2.

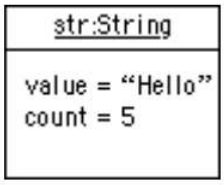


Figure 2.2: A String object stores a sequence of characters and a count giving the number of characters.

We can also use a constructor with an empty parameter list. Note that in this case we combine the *variable declaration* and the *object creation* into one statement:

```
String str2 = new String(); // Create a String
```

This example will create a String object that contains the empty string as its value. The **empty string** has the literal value "" – that is, a pair of double quotes that contain no characters. Because the empty string has no characters, the count variable stores a zero (Fig. 2.3).

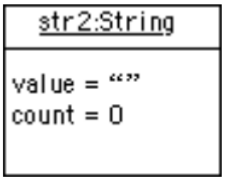


Figure 2.3: The empty string has a value of "" and a its length is 0.

Note that we use a constructor to assign an initial value to a variable of type String (or of a type equal to any other class). This differs from how we assign an initial value to variables of primitive type, for which we use a simple assignment operator. This difference is related to an important difference in the way Java treats these two types of variables. Variables of primitive type are names for memory locations where values of primitive type are stored. As soon as they are declared they are assigned a **default value** of that primitive type. The default value for int is 0 and the default value for boolean is false. On the other hand, variables that are declared to be of a type equal to a class name are designed to store a **reference** to an object of that type. (A reference is also called a **pointer** because it points to the memory address where the object itself is stored.) A constructor creates an object somewhere in memory and supplies a reference to it that is stored in the variable. For that reason, variables that are declared as a type equal to a class name are said to be variables of reference type or **reference variables**. Reference variables have a special default value called null after they are declared and before they are assigned a reference. It is possible to check whether or not a reference variable contains a reference to an actual object by checking whether or not it contains this **null pointer**.

Once you have constructed a String object, you can use any of the methods shown in Figure 2.1 on it. As we already saw, we use dot notation to call one of the methods. Thus, we first mention the name of the object followed by a period (dot), followed by the name of the method. For example, the following statements print the lengths of our two strings:

```
System.out.println(str.length());
System.out.println(str2.length());
```

Another useful String method is the concat(String) method, which can be used to *concatenate* two strings. This method takes a String argument. It returns a String that combines the String argument to the String that the method is called on. Consider this example:

```
String s1 = new String("George ");
String s2 = new String("Washington");
System.out.println(s1.concat(s2));
```

In this case, the concat() method adds the String *s2* to the end of the String *s1*. The result, which gets printed, will be the String "George Washington".

Because strings are so important, Java allows a number of shortcuts to be used when creating and concatenating strings. For example, you don't have to use new String() when creating a new string object. The following code will also work:

```
String s1 = "George ";
String s2 = "Washington";
```

Similarly, an easier way to concatenate two String objects is to use the plus sign (+), which serves as a *concatenation operator* in Java:

```
System.out.println(s1 + s2);
```

Another useful String method is the equals() method. This is a boolean method, which is used to compare two Strings. If both Strings have the same characters, in the same order, it will return true. Otherwise it will return false. For example, consider the following code segment:

```
String s1 = "Hello";
String s2 = "Hello";
String s3 = "hello";
```

In this case, the expression s1.equals(s2) will be true, but s1.equals(s3) will be false.

It is important to note that the empty string is not the same as a String variable that contains null. Executing the statements:

```
String s1;
String s2 = "";
System.out.println(s1.equals(s2));
```

will not only not print out true; it will cause the the program to terminate abnormally. It is an error to use the method of a String variable, or any other variable whose type is a class, before it has been assigned an object. When the above code is executed, it will report a **null pointer exception**, one of the most common runtime errors. When you see that error message, it means that some method was executed on a variable that does not refer to an object. On the other hand, the empty string is a perfectly good String object which just happens to contain zero characters.

Figure 2.4 shows a program that uses string concatenation to create some silly sentences. The programs declares a number of string variables, named s, s1, and so on, and it instantiates a String object for each variable to refer to. It then prints out a top-five list using the concatenation operator to combine strings. Can you figure out what it prints without running it?

```
public class StringPuns
{
  public static void main(String args[])
  { String s = new String("string");
    String s1 = s.concat(" puns.");
    System.out.println("Here are the top 5 " + s1);
    String s2 = "5. Hey baby, wanna ";
    String s3 = s + " along with me.";
    System.out.println(s2 + s3);
    System.out.println("4. I've got the world on a " +
                                              s + ".");
    String s4 = new String("two");
    String s5 = ". You have more class than a ";
    System.out.print(s4.length());
    System.out.println(s5 + s + " of pearls.");
    System.out.print("2. It is ");
    System.out.print(s.equals("string"));
    System.out.println(" that I am no " + s + " bean.");
    String s6 = " quintet.";
    System.out.println("1. These puns form a " + s + s6);
  } // main()
} //  StringPuns class
```

Figure 2.4: A program that prints silly string puns.

? Exercise 2.2.1

What is the output to the console window when the following Java code fragment is executed:

```
String s = "ing";
System.out.println("The s" + s + s + " k" + s + ".");
```

This page titled 2.2: Using String Objects is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **2.2: Using String Objects** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

2.3: Drawing Shapes with the Graphics Object (Optional)

All of the instance methods of the String class that we examined return values. The length() method return an int value, and the concat() method returned a String. It is also very common for classes to define instance methods that perform actions but do not return a value. The Graphics object, g, that appears in Chapter 1's HelloWorldSwing is one example. The program is reproduced in Figure 2.5.

```
 /** File: HelloWorldSwing program */

import javax.swing.JFrame; // Import class names
import java.awt.Graphics;
import java.awt.Canvas;

public class HelloWorldCanvas extends Canvas // Class header
{
    // Start of body
    public void paint(Graphics g)
        // The paint method
    {
        g.drawString("Hello, World!", 10, 10);
    }  // End of paint

    public static void main(String[] args){
        HelloWorldCanvas c = new HelloWorldCanvas();
        JFrame f = new JFrame();
        f.add(c);
        f.setSize(150,50);
        f.setVisible(true);
    }
}  // End of HelloWorldCanvas
```

Figure 2.5: HelloWorldCanvas program source code.

At this point we will not worry about the language features that enable the paint() method to draw on the Java Swing window. We will focus instead on the information needed to make good use of the g.drawString() method. The first thing you should know is that, when the paint() method is executed, its parameter, g, refers to an instance of the Graphics class. Unlike our other examples involving variables that refer to objects, in this case there is no need to use a constructor to create an object of type Graphics. We can assume g already refers to such an object.

We already know that the statement

```
g.drawString("Hello, World!",10,10);
```

displays the String "Hello, World!" in the program window. More generally, if str is a literal String value or a reference to a String object and x and y are literal int values or int variables then

```
g.drawString(str,x,y)
```

displays the String str from left to right in the program window beginning at a point which is x pixels from the left edge of the window and y pixels down from the top edge of the window. In a graphics window, the point with coordinates (0,0) is at the top-left corner. The horizontal axis grows positively from left to right. The vertical axis grows positively from top to bottom (Fig. 2.6).

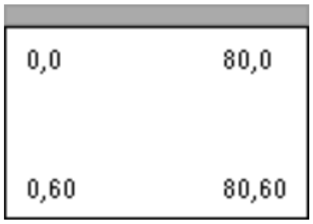


Figure 2.6: Coordinate system of a Java window.

(A *pixel* is a dot on the console window that can be set to a certain color.) Notice that increasing the value of y will cause str to be displayed lower. This is the opposite of the usual x and y coordinate system used in mathematics where increasing the y value designates a higher point.

With this information about g.drawString(), we can calculate where to display any message in the program window. For example, if we wish to display the message “Welcome to Java” 25 pixels below where “Hello, World!” is displayed we could use the statements

```
g.drawString("Hello, World!",10,10);
g.drawString("Welcome to Java",10,35);
```

in the body of HelloWorldCanvas’s paint() method. The result of these statements would appear as shown in Figure 2.7.



Figure 2.7

Graphics Drawing Methods

The Graphics class discussed in the previous section also has methods that can be used to draw geometric shapes in different colors. These methods can be used to create graphical user interfaces that are more interesting or to give a visual representation of data, such as a pie chart or a bar graph.

There are two Graphics methods for drawing rectangles, fillRect() and drawRect() (Fig. 2.8). The first draws a rectangle and fills it with the current drawing color and the second just draws the outline of the rectangle. Using the Graphicsobject, g, each of these is called in the same way as the drawString() method from the previous example. Each of these methods takes four int arguments, which specify the rectangle’s location and size. Thus, a call to fillRect() would take the form

```
g.fillRect(x,y,width,height);
```

where x and y arguments specify the location of the upper left corner of the rectangle as being x pixels from the left edge of the window and y pixels down from the top edge of the window. The width and height arguments specify the width and height of the rectangle in pixels. The drawRect() method also takes the same four arguments.

A Graphics object stores a single color for use in drawing shapes or displaying strings with drawString(). If we wish to draw an interesting scene in the JFrame, we need to understand how to use colors.

For a given Graphics object, such as g, the setColor() method will set its color for all subsequent drawing commands. The setColor() method takes, as an argument, an object of type Color. All we need to know about the Color class is that it is contained in the java.awt package and that it contains 13 constant Color objects corresponding to 13 common colors. Table 2.1 lists the 13 Color constants. Each name corresponds to the color it will represent in the program.

Table 2.1: Predefined color constants in the Color class.

Color.black	Color.green	Color.red
Color.blue	Color.lightGreen	Color.white
Color.cyan	Color.magenta	Color.yellow
Color.darkGray	Color.orange	
Color.gray	Color.pink	

To demonstrate how the new Graphics methods can be used for creating more interesting graphical programs, let's develop a plan for displaying the two messages, "Hello, World!" and "Welcome to Java.", on an JFrame, but this time we will draw the first inside a colored rectangle and the second inside a colored oval. For the rectangle, let's use the drawRect() method to create its border. We can choose some arbitrary colors, say, cyan for filling the rectangle, blue for its border, and black for the string itself. In order to have the message visible we should fill a rectangle with the color cyan first, then draw the border of the rectangle in blue and, finally, display the message in black.

Drawing and filling a Graphics oval is very similar to drawing and filling a rectangle. Notice in Figure 2.8 that the fillOval() and drawOval() methods take the same four arguments as the corresponding rectangle methods. An oval is inscribed within an enclosing rectangle. The x and y arguments give the coordinates of the enclosing rectangle's top left point. And the width and height arguments give the enclosing rectangles dimensions.

All that remains is to choose the location and dimensions of the rectangles. We could specify one rectangle as having its upper left corner 25 pixels to the right of the left edge of the JFrame and 25 pixels down from the top edge. A medium sized rectangle could have a width of 140 pixels and a height of 40 pixels. The statement

```
g.fillRect(25, 25, 140, 40);
```

will fill this rectangle with whatever color happens to be g's current color. A location 25 pixels to the right of the left edge of the rectangle and 25 pixels down from the top edge of the rectangle would have coordinates and . Thus, the statement

```
g.drawString("Hello, World!", 50, 50);
```

will display "Hello, World!" inside the rectangle. We can use similar planning to locate the oval and its enclosed message.

Thus, we now have sufficient information to finish the paint() method for accomplishing our plan. The completed program is displayed in Figure 2.9. Note how we repeatedly use the g.setColor() method to change g's current color before drawing each element of our picture.

```
import java.awt.*;
import javax.swing.JFrame;

public class HelloWorldGraphic  extends Canvas
{
    // called after setVisible(true)
    public void paint(Graphics g) {
        g.setColor(Color.cyan);      // Set color
        g.fillRect(25, 25, 140, 40); // Fill rectangle
        g.setColor(Color.blue);      // Set color
        g.drawRect(25, 25, 140, 40); // Outline rectangle
        g.setColor(Color.black);     // Set color
        g.drawString("Hello, World!", 50, 50); // Display string
        g.setColor(Color.yellow);
        g.fillOval(25, 75, 140, 40); // Fill oval
        g.setColor(Color.red);
        g.drawOval(25, 75, 140, 40); // Outline oval
        g.setColor(Color.black);
        g.drawString("Welcome to Java", 50, 100);
    }//paint()

   // the program
    public static void main(String[] args){
        HelloWorldCanvas c = new HelloWorldCanvas();
        JFrame f = new JFrame();
        f.add(c);
        f.setSize(150,50);
        f.setVisible(true);
    }
} //HelloWorldGraphic
```

Figure 2.9: The HelloWorldGraphic class is a Java Swing program that shows how to use color and drawing methods.

Figure 2.10 shows what this program looks like. To experiment with this Java Swing application, download its sourcecode from the book's Web site and compile and run it on your computer. Additional drawing capabilities will be explored throughout the text in sections that can either be covered or skipped.



Figure 2.10: This is how the graphic program run.

This page titled 2.3: Drawing Shapes with the Graphics Object (Optional) is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **2.3: Drawing Shapes with the Graphics Object (Optional)** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

2.4: Class Definition

To program in Java the main thing you do is write class definitions for the various objects that will make up the program. A class definition *encapsulates* its objects' data and behavior. Once a class has been defined, it serves as a *template*, or blueprint, for creating individual *objects* or instances of the class.

A class definition contains two types of elements: variables and methods. *Variables* are used to store the object's information. *Methods* are used to process the information. To design an object you need to answer five basic questions:

1. What role will the object perform in the program?
2. What data or information will it need?
3. What actions will it take?
4. What interface will it present to other objects?
5. What information will it hide from other objects?

The Riddle Class

Recall our definition of the Riddle class from Chapter 1, which is summarized in the UML diagram in Figure 2.11. A Riddle has two attributes, question and answer. Each of these variables stores a string of characters, which Java treats as data of type String. The Riddle class contains three methods. The Riddle() constructor method assigns initial values (qand a) to its question and answer variables. The getQuestion() and getAnswer() methods return the data stored in question ands answer respectively.

The instance variables question and answer are designated as private (−), but the Riddle(), getQuestion() and getAnswer() methods are designated as public (+). These designations follow two important object-oriented design conventions, whose justification will become apparent as we discuss the Riddle class:

> JAVA EFFECTIVE DESIGN
>
> Private Variables. Instance variables are usually declared private so that they cannot be directly accessed by other objects.

> JAVA EFFECTIVE DESIGN
>
> Public Methods. An object's public methods can be used by other objects to interact with the object. The public methods and variables of an object make up its interface.

Figure 2.12 shows the Java class definition that corresponds to the design given in the UML diagram. It contains the two private instance variables and defines the three public methods listed in the UML diagram. In a Java class definition, access to a class element, such as a variable or a method, is controlled by labeling it with either the private, or public access modifier. An **access modifier** is a declaration that controls access to a class or one of its elements. Note also that the Riddle class itself is declared public. This lets other classes have access to the class and to its public variables and methods.

```
public class Riddle
{ private String question; //Instance variables
  private String answer;

  public Riddle(String q, String a) // Constructor
  { question = q;
    answer = a;
  } // Riddle constructor

  public String getQuestion() // Instance method
  { return question;
  } // getQuestion()

  public String getAnswer() // Instance method
  { return answer;
  } //getAnswer()
} //Riddle class
```

Figure 2.12: Definition of the Riddle class.

Recall that a class is like a blueprint or a cookie cutter. The Riddle class defines the type of information (attributes) that each individual Riddle has, but it doesn't contain any actual values. It defines the methods (operations) that each Riddlecan perform, but it doesn't actually perform the methods. In short, a class serves as a template, providing a detailed blueprint of the objects (or instances) of that class.

The RiddleUser Class

Now that we have defined the Riddle class, we can test that it works correctly by creating Riddle objects and "asking" them to tell us their riddles. To do this we need to define a main() method, which can be defined either within the Riddleclass itself or in a second class named something like RiddleUser.

One advantage of using a second class is that it gets us in the habit of thinking about the need for a separate class to serve as a user interface, with a separate set of tasks from the Riddle class. A **user interface** is an object or class that handles the interaction between a program's user and the rest of the program's computational tasks. This concept is illustrated in Figure 2.13. Note that we use the general term *computational object* to distinguish the rest of the program's computations from the user interface. Obviously, the exact nature of the computation will vary from program to program, just as will the details of the user interface. The computation done by our Riddle class is just the storing and displaying of a riddle's question and answer.

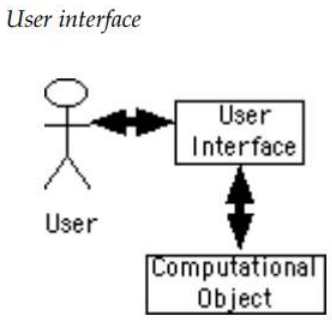


Figure 2.13: The user interfaces handles interaction between the user and the rest of the program.

By separating user interface tasks from riddle tasks this design employs the divide-and-conquer principle: the RiddleUser class will create Riddle objects and handle interactions with the user, and the Riddle class will handle the storing and transmission of riddle information. Thus, as shown in Figure 2.14, this particular Java program will involve interaction between two types of objects: a RiddleUser and one or more Riddles. Note that we characterize the relationship between Riddle and RiddleUser with a one-way

arrow labeled “Uses.” This is because the RiddleUser will create an instance of Riddle and use its methods to display (for the user) a riddle.

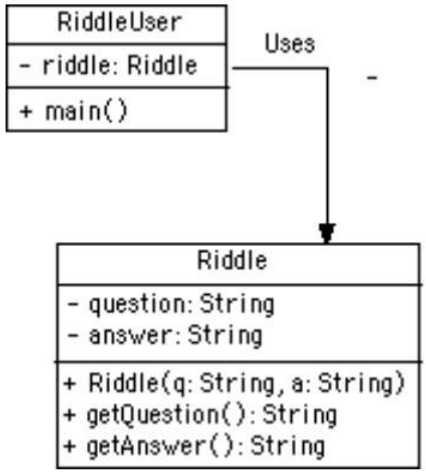


Figure 2.14

Because almost all of our programs will involve some form of a user interface, we can generalize this design approach and follow it throughout the book. One way to think about this approach is as a division of labor between a user interface class and a second *computational* class, which performs whatever computations are needed by the particular program. In this case the computations are the simple Riddle methods that we have defined. In subsequent programs the computations will become more complex, which will make all the more clear that they should be separated from the user interface.

Object Instantiation: Creating Riddle Instances

Figure 2.15 shows the complete definition of the RiddleUser class, which serves as a very simple user interface. It creates two Riddle objects, named riddle1 and riddle2. It then asks each object to request each riddle’s question and answer and displays them on the console.

```
public class RiddleUser
{
  public static void main(String argv[])
  { Riddle riddle1 = new Riddle(
      "What is black and white and red all over?",
      "An embarrassed zebra.");
    Riddle riddle2 = new Riddle(
      "What is black and white and read all over?",
      "A newspaper.");
    System.out.println("Here are two riddles:");
    System.out.println(riddle1.getQuestion());
    System.out.println(riddle2.getQuestion());
    System.out.println("The answer to the first riddle is:");
    System.out.println(riddle1.getAnswer());
    System.out.println("The answer to the second is:");
    System.out.println(riddle2.getAnswer());
  } // main()
} // RiddleUser
```

Figure 2.15: The RiddleUser class.

Let’s now discuss the statements that make up RiddleUser’s main() method. The following statements use the Riddle()constructor to create, or *instantiate*, two instances of the Riddle class:

```
Riddle riddle1 = new Riddle(
     "What is black and white and red all over?",
     "An embarrassed zebra.");
Riddle riddle2 = new Riddle(
    "What is black and white and read all over?",
     "A newspaper.");
```

Note how the constructor gives each object a pair of Strings that serve as the values of their two instance variables. Each object has its own question and its own answer, and each object has its own unique name, riddle1 and riddle2.

Interacting with Riddles

Once we have created Riddle instances with values assigned to their question and answer instance variables, we can ask each riddle to tell us either of its values. The following expression is an example of a *method call*:

```
riddle1.getQuestion()
```

Calling (or invoking) a method is a means of executing its code. The above method call just gets the String value that is stored in the question instance variable of riddle1.

JAVA PROGRAMMING TIP

Method Call versus Method Definition. Don't confuse method calls with method definitions. The definition specifies the method's actions. The method call takes those actions.

If we want to display the value of riddle1's question, we can embed this method call within a println() statement

```
System.out.println(riddle1.getQuestion());
```

This tells the System.out object to execute its println() method, which displays the string given to it by riddle1 on the console. Thus, the output produced by this statement will be

```
    What is black and white and red all over?
```

Define, Create, Use

As our Riddle example illustrates, writing a Java program is a matter of three basic steps:

- Define one or more classes (class definition).
- Create objects as instances of the classes (object instantiation).
- Use the objects to do tasks (object use).

The Java class definition determines what information will be stored in each object and what methods each object can perform. Instantiation creates an instance and associates a name with it in the program. The object's methods can then be called as a way of getting the object to perform certain tasks.

? Exercise 2.4.1

Identify the following elements in the Riddle class (Fig. 2.12):

- The name of the class.
- The names of two instance variables.
- The names of three methods.

Exercise 2.4.2

Identify the following elements in the RiddleUser class (Fig. 2.15):

- The names of two Riddle instances.
- All six method calls of the Riddle objects in the program.
- Two examples of qualified names.

This page titled 2.4: Class Definition is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **2.4: Class Definition** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

2.5: Case Study- Simulating a Two-Person Game

In this section, we will design and write the definition for a class that keeps track of the details of a well known, two-person game. We will focus on details of designing the definition of a class in the Java language. Our objective is to understand what the program is doing and how it works without necessarily understanding why it works the way it does. We will get to “why” later in the book.

The game we will consider is played by two persons with a row of sticks or coins or other objects. The players alternate turns. A player must remove one, two, or three sticks from the row on his or her turn. The player who removes the last stick from the row loses. The game can be played with any number of sticks but starting with twenty one sticks is quite common. This game is sometimes referred to as the game of "Nim", but there is a similar game involving multiple rows of sticks that is more frequently given that name. Thus we will refer to this game as "One Row Nim".

Designing a OneRowNim class

Problem Specification

Let’s design a class named OneRowNim that simulates the game of One Row Nim with a row of sticks. An object constructed with this class should manage data that corresponds to having some specified number of sticks when the game begins. It should keep track of whose turn it is and it should allow a player to diminish the number of sticks remaining by one, two, or three. Finally, a OneRowNim object should be able to decide when the game is over and which player has won.

Problem Decomposition

Let’s design OneRowNim so that it can be used in with different kinds of user interfaces. One user interface could manage a game played by two persons who alternately designate their moves to the computer. Another user interface could let a human player play against moves made by the computer. In either of these cases we could have a human player designate a move by typing from the keyboard after being prompted in a console window or, alternatively, by inputting a number into a text field or selecting a radio button on a window. In this chapter, we will be concerned only with designing an object for managing the game. We will design user interfaces for the game in subsequent chapters.

Class Design: OneRowNim

As we saw in the Riddle example, class definitions can usually be broken down into two parts: (1) the information or attributes that the object needs which must be stored in variables, and (2) the behavior or actions the object can take which are defined in methods. In this chapter, we will focus on choosing appropriate instance variables and on designing methods as blocks of reusable code. Recall that a parameter is a variable that temporarily stores data values that are being passed to a method when that method is called. In this chapter, we will restrict our design to methods that do not have parameters and do not return values. We will return to the problem of designing changes to this class in the next chapter after an in-depth discussion of method parameters and return values.

The OneRowNim object should manage two pieces of information that vary as the game is played. One is the number of sticks remaining in the row and the other is which player has the next turn. Clearly, the number of sticks remaining corresponds to a positive integer that can be stored in a variable of type int. One suitable name for such a variable is nSticks. For this chapter, let us assume that the game starts with 7 sticks, rather than 21, to simplify discussion of the program.

Data designating which player takes the next turn could be stored in different ways. One way to do this is to think of the players as player one and player two and store a 1 or 2 in an int variable. Let’s use player as the name for such a variable and assume that player one has the first turn.

The values of these two variable for a particular OneRowNim object at a particular time describes the object’s state. An object’s state at the beginning of a game is a 7 stored in nSticks and 1 stored in player. After player one removes, say, two sticks on the first turn, the values 5 and 2 will be stored in the two variables.

Method Decomposition

Now that we have decided what information the OneRowNim object should manage, we need to decide what actions it should be able to perform. We should think of methods that would be needed to communicate with a user interface that is both prompting some human players as well as receiving moves from them. Clearly, methods are needed for taking a turn in the game. If a message to a OneRowNim object has no argument to indicate the number of sticks taken, there will need to be three methods corresponding

to taking one, two, or three sticks. The method names takeOne(), takeTwo(), and takeThree() are descriptive of this action. Each of these methods will be responsible for reducing the value of nSticks as well as changing the value of player.

We should also have a method that gives the information that a user needs when considering a move. Reporting the number of sticks remaining and whose turn it is to the console window would be an appropriate action. We can use report() as a name for this action.

Figure 2.16 is a UML class diagram that summarizes this design of the OneRowNim class. Note that the methods are declared public (+) and will thereby form the interface for a OneRowNim object. These will be the methods that other objects will use to interact with it. Similarly, we have followed the convention of designating an object's instance variables—the 's instance variables —be kept hidden from other objects, and so we have designated them as private(–).

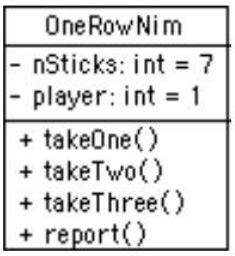


Figure 2.16: A UML class diagram for OneRowNim.

Defining the OneRowNim Class

Given our design of the OneRowNim class as described in Figure 2.16, the next step in building our simulation is to begin writing the Java class definition.

The Class Header

We need a *class header*, which will give the class a name and will specify its relationship to other classes. Like all classes that are designed to create objects that could be used by other objects or classes, the class OneRowNim should be preceded by the public modifier. Because the class OneRowNim has not been described as having any relationship to any other Java class, its header can omit the extends clause so it will be a direct subclass of Object (Figure 2.17).

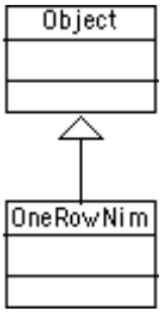


Figure 2.17: By default, OneRowNim is a subclass of Object.

Thus, the class header for OneRowNim will look like:

```
public class OneRowNim  // Class header
{                       // Beginning of class body

}                       // End of class body
```

The Class's Instance Variables

The body of a class definition consists of two parts: the class-level variables and the method definitions. A **class-level variable** is a variable whose definition applies to the entire class in which it is defined. Instance variables, which were introduced in Chapter 1,

are one kind of class-level variable.

In general, a class definition will take the form shown in Figure 2.18.

```
public class ClassName
{  // Instance and class variables
     VariableDeclaration1
     VariableDeclaration2
     ...
    // Instance and class methods
     MethodDefinition1
     MethodDefinition2
     ...
} // End of class
```

Figure 2.18: A template for constructing a Java class definition.

Although Java does not impose any particular order on variable and method declarations, in this book we'll define the class's class-level variables at the beginning of the class definition, followed by method definitions. Class-level variables are distinguished from local variables. A **local variable** is a variable that is defined within a method. Examples would be the variables q and a that were defined in the Riddle(String q, String a) constructor (Fig. 2.12). As we will see better in Chapter 3, Java handles each type of variable differently.

A declaration for a variable at class level must follow the rules for declaring variables that were described in Section with the added restriction that they should be modified by one of the access modifiers public, private, or protected. The rules associated with these access modifiers are:

- A private class-level variable cannot be accessed outside the class in which it is declared.
- A public class-level variable can be referenced and, hence, modified by any other class.
- A protected class-level variable can only be accessed by subclasses of the class in which it is declared or by other classes that belong to the same package.

When a class, instance variable, or method is defined, you can declare it public, protected, or private. Or you can leave its access unspecified, in which case Java's default accessibility will apply.

Java determines accessibility in a top-down manner. Instance variables and methods are contained in classes, which are contained in packages. To determine whether a instance variable or method is accessible, Java starts by determining whether its containing package is accessible, and then whether its containing class is accessible. Access to classes, instance variables, and methods is defined according to the rules shown in Table 2.2.

Table 2.2: Java's accessibility rules.

Element	Modifier	Rule
Class	public	Accessible if its package is accessible.
	by default	Accessible only within its package.

Element Modifier	Rule
Instance variable	
public	Accessible to all other objects.
protected	Accessible to its subclasses and to other classes in its package.
private	Accessible only within the class.
by default	Accessible only within the package.
Instance method	

Recall the distinction we made in Chapter 0 between class variables and instance variables. A class variable is associated with the class itself, whereas an instance variable is associated with each of the class's instances. In other words, each object contains its own copy of the class's instance variables, but only the class itself contains the single copy of a class variable. To designate a variable as a class variable it must be declared static.

The Riddle class that we considered earlier has the following two examples of valid declarations of instance variables:

```
private String question;
private String answer;
```

Class Level Variables for OneRowNim

Let's now consider how to declare the class level variables for the OneRowNim class. The UML class diagram for OneRowNim in Figure 2.16 contains all the information we need. The variables nSticks and player will store data for playing one game of One Row Nim, so they should clearly be private instance variables. They both will store integer values, so they should be declared as variables of type int. Because we wish to start a game of One Row Nim using 7 sticks with player one making the first move, we will assign 7 as the initial value for nSticks and 1 as the initial value for player. If we add the declarations for our instance variable declarations to the class header for the OneRowNimclass, we get the following:

```
public class OneRowNim
{
  private int nSticks = 7;
  private int player = 1;

  //Method definitions go here
} // OneRowNim
```

To summarize, despite its apparent simplicity, a class level variable declaration actually accomplishes five tasks:

1. Sets aside a portion of the object's memory that can be used to store a certain type of data.
2. Specifies the type of data that can be stored in that location.
3. Associates an identifier (or name) with that location.
4. Determines which objects have access to the variable's name.
5. Assigns an initial value to the location.

OneRowNim's Methods

Designing and defining methods is a form of abstraction. By defining a certain sequence of actions as a method, you encapsulate those actions under a single name that can be invoked whenever needed. Instead of having to list the entire sequence again each time you want it performed, you simply call it by name. As you recall from Chapter 1, a method definition consists of two parts, the method header and the method body. The method header declares the name of the method and other general information about the method. The method body contains the executable statements that the method performs.

```
public void methodName()  // Method header
{                  // Beginning of method body
}                  // End of method body
```

The Method Header

The method header follows a general format that consists of one or more *MethodModifiers*, the method's *ResultType*, the *MethodName*, and the method's *FormalParameterList*, which is enclosed in parentheses. The following table illustrates the method header form, and includes several examples of method headers that we have already encountered. The method body follows the method header.

$MethodModifiers_{opt}$	ResultType	MethodName	(FormalParameterList)
public static	void	main	(String argv[])
public	void	paint	(Graphics g)
public		Riddle	(String q, String a)
public	String	getQuestion	()
public	String	getAnswer	()

The rules on method access are the same as the rules on instance variable access: private methods are accessible only within the class itself, protected methods are accessible only to subclasses of the class in which the method is defined and to other classes in the same package, and public methods are accessible to all other classes.

JAVA EFFECTIVE DESIGN

Public versus Private Methods. If a method is used to communicate with an object, or if it passes information to or from an object, it should be declared public. If a method is intended to be used solely for internal operations within the object, it should be declared private. These methods are sometimes called utility methods or helper methods.

Recall the distinction from Chapter 0 between instance methods and class methods. Methods declared at the class level are assumed to be instance methods unless they are also declared static. The **static modifier** is used to declare that a class method or variable is associated with the class itself, rather than with its instances. Just as for static variables, methods that are declared static are associated with the class and are therefore called *class methods*. As its name implies, an instance method can only be used in association with an object (or instance) of a class. Most of the class-level methods we declare will be instance methods. Class methods are used only rarely in Java and mainly in situations where it is necessary to perform some kind calculation before objects of the class are created. We will see examples of class methods when we discuss the Math class, which has such methods as sqrt(N) to calculate the square root of *N*.

JAVA PROGRAMMING TIP

Class versus Instance Methods. If a method is designed to be used by an object, it is referred to as an instance method. No modifier is needed to designate an instance method. Class methods, which are used infrequently compared to instance methods, must be declared static.

All four of the methods in the OneRowNim class are instance methods (Fig. 2.19). They all perform actions associated with a particular instance of OneRowNim. That is, they are all used to manage a particular One Row Nimgame. Moreover, all four methods should be declared public, because they are designed for communicating with other objects rather than for performing internal calculations. Three of the methods are described as changing the values of the instance variables nSticks and player and the fourth, report(), writes information to the console. All four methods will receive no data when being called and will not return any values. Thus they should all have void as a return type and should all have empty parameter lists.

```
public class OneRowNim
{ private int nSticks = 7; // Start with 7 sticks.
  private int player = 1;  // Player 1 plays first.

  public void takeOne(){ } // Method bodies need
  public void takeTwo(){ } //  to be defined.
  public void takeThree(){ }
  public void report(){ }
} //OneRowNim class
```

Figure 2.19: The Instance variables and method headers for the OneRowNim class.

Given these design decisions, we now can add method headers to our class definition of OneRowNim, in Figure 2.19. The figure displays the class header, instance variable declarations, and method headers.

The Method Body

The body of a method definition is a block of Java statements enclosed by braces, , which are executed in sequence when the method is called. The description of the action required of the takeOne() method is typical of many methods that change the state of an object. The body of the takeOne() method should use a series of assignment statements to reduce the value stored in nSticks by one and change the value in player from 2 to 1 or from 1 to 2. The first change is accomplished in a straightforward way by the assignment:

```
nSticks = nSticks - 1;
```

This statement says subtract 1 from the value stored in nSticks and assign the new value back to nSticks.

Deciding how to change the value in player is more difficult because we do not know whether its current value is 1 or 2. If its current value is 1, its new value should be 2; if its current value is 2, its new value should be 1. Notice, however, that in both cases the current value plus the desired new value are equal to 3. Therefore, the new value of player is equal to 3 minus its current value. Writing this as an assignment we have:

```
player = 3 - player;
```

One can easily verify that this clever assignment assigns 2 to player if its current value is 1 and assigns 1 to it if its current value is 2. In effect, this assignment will toggle the value off player between 1 and 2 each time it is executed. In the next chapter we will introduce the if-else control structure that would allow us to accomplish this same toggling action in a more straightforward manner. The complete definition of takeOne() method becomes:

```
public void takeOne()
{
   nSticks = nSticks - 1;  // Take one stick
   player = 3 - player;    // Change to other player
}
```

The takeTwo() and takeThree() methods are completely analogous to the takeOne() method with the only difference being the amount subtracted from nSticks.

The body of the report() method must merely print the current values of the instance variables to the console window with System.out.println(). To be understandable to someone using a OneRowNim object, the values should be clearly labeled. Thus the body of report() could contain:

```
System.out.println("Number of sticks left: " + nSticks);
System.out.println("Next turn by player " + player);
```

This completes the method bodies of the OneRowNim class. The completed class definition is shown in Figure 2.20. We will discuss alternative methods for this class in the next chapter. In Chapter 4, we will develop several One Row Nim user interface classes that will facilitate a user indicating certain moves to make.

```
public class OneRowNim
{ private int nSticks = 7; // Start with 7 sticks.
  private int player = 1;  //Player 1 plays first.

  public void takeOne()
  { nSticks = nSticks - 1;
    player = 3 - player;
  } // takeOne()

  public void takeTwo()
  { nSticks = nSticks - 2;
    player = 3 - player;
  } // takeTwo()

  public void takeThree()
  { nSticks = nSticks - 3;
    player = 3 - player;
  }  // takeThree()

  public void report()
  { System.out.println("Number of sticks left: " + nSticks);
    System.out.println("Next turn by player " + player);
  }   // report()
} // OneRowNim1 class
```

Figure 2.20: The OneRowNim class definition.

Testing the OneRowNim Class

Recall our *define*, *create*, *and use* mantra from Section. Now that we have defined the OneRowNim class, we can test whether it works correctly by creating OneRowNim objects and using them to perform the actions associated with the game. At this point, we can test OneRowNim by defining a main() method. Following the design we used in the riddle example, we will locate the main() method in separate, user interface class, named OneRowNimTester.

The body of main() should declare a variable of type OneRowNim and create an object for it to refer to. The variable can have any name, but a name like game would be consistent with it recording moves in a single game. To test the OneRowNim class, we should make a typical series of moves. For example, three moves taking 3, 3, and 1 sticks respectively would be one way that the 7 sticks could be removed. Also, executing the report() method before the first move and after each move should display the current state of the game in the console window so that we can determine whether it is working correctly.

The following pseudocode outlines an appropriate sequence of statements in a main() method:

1. Declare a variable of type OneRowNim named game.
2. Instantiate a OneRowNim object to which game refers.
3. Command game to report.
4. Command game to remove three sticks.
5. Command game to report.
6. Command game to remove three sticks.
7. Command game to report.
8. Command game to remove one stick.
9. Command game to report.

It is now an easy task to convert the steps in the pseudocode outline into Java statements. The resulting main() method is shown with the complete definition of the OneRowNimTester class:

```
public class OneRowNimTester
{ public static void main(String args[])
  {   OneRowNim1 game = new OneRowNim();
      game.report();
      game.takeThree();
      game.report();
      game.takeThree();
      game.report();
      game.takeOne();
      game.report();
  } //main()
}
```

When it is run, OneRowNimTester produces the following output:

```
    Number of sticks left: 7
    Next turn by player 1
    Number of sticks left: 4
    Next turn by player 2
    Number of sticks left: 1
    Next turn by player 1
    Number of sticks left: 0
    Next turn by player 2
```

This output indicates that player 1 removed the final stick and so player 2 is the winner of this game.

? Exercise 2.5.1

Add a new declaration to the Riddle class for a private String instance variable named hint. Assign the variable an initial value of "This riddle is too easy for a hint".

? Exercise 2.5.2

Write a header for a new method definition for Riddle named getHint(). Assume that this method requires no parameters and that it simply returns the String value stored in the hint instance variable. Should this method be declared public or private?

? Exercise 2.5.3

Write a header for the definition of a new public method for Riddle named setHint() which sets the value of the hintinstance variable to whatever String value it receives as a parameter. What should the result type be for this method?

? Exercise 2.5.4

Create a partial definition of a Student class. Create instance variables for the first name, last name, and an integer student identification number. Write the headers for three methods. One method uses three parameters to set values for the three instance variables. One method returns the student identification number. The last method returns a Stringcontaining the student's first name and last name. Write only the headers for these methods.

Flow of Control: Method Call and Return

A program's **flow of control** is the order in which its statements are executed. In an object-oriented program, control passes from one object to another during the program's execution. It's important to have a clear understanding of this process.

In order to understand a Java program, it is necessary to understand the **method call and return** mechanism. We will encounter it repeatedly. A method call causes a program to transfer control to a statement located in another method. Figure 2.21 shows the method call and return structure.

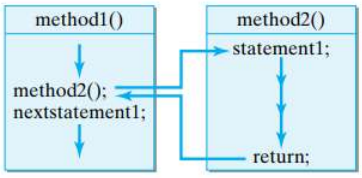


Figure 2.21: The method call and return control structure. It's important to realize that method1() and method2() may be contained in different classes.

In this example, we have two methods. We make no assumptions about where these methods are in relation to each other. They could be defined in the same class or in different classes. The method1() method executes sequentially until it calls method2(). This transfers control to the first statement in method2(). Execution continues sequentially through the statements in method2() until the return statement is executed.

JAVA LANGUAGE RULE

Return Statement. The return statement causes a method to return control to the calling statement—that is, to the statement that called the method in the first place.

Recall that if a void method does not contain a return statement, then control will automatically return to the calling statement after the invoked method executes its last statement.

Tracing the OneRowNim Program

To help us understand the flow of control in OneRowNim, we will perform a trace of its execution. Figure 2.22 shows all of the Java code involved in the program. In order to simplify our trace, we have moved the main() method from OneRowNimTester to the OneRowNim class. This does not affect the program's order of execution in any way. But keep in mind that the code in the main() method could just as well appear in the OneRowNimTester class. The listing in Figure 2.22 also adds line numbers to the program to show the order in which its statements are executed.

```
  public class OneRowNim
2 {  private int nSticks = 7; // Start with 7 sticks.
3    private int player = 1;  //Player 1 plays first.

      public void takeOne()
20    {  nSticks = nSticks - 1;
21       player = 3 - player;
      } // takeOne()
      public void takeTwo()
      {  nSticks = nSticks - 2;
         player = 3 - player;
      } // takeTwo()

      public void takeThree()
8,14  {  nSticks = nSticks - 3;
9,15     player = 3 - player;
      }  // takeThree()
      public void report()
5,11,17,23 { System.out.println("Number of sticks left: " + nSticks);
6,12,18,24   System.out.println("Next turn by player " + player);
      }   // report()

      public static void main(String args[])
1     {  OneRowNim1 game = new OneRowNim1();
4        game.report();
7        game.takeThree();
10       game.report();
13       game.takeThree();
16       game.report();
19       game.takeOne();
22       game.report();
23     } //main()
   } //OneRowNim1 class
```

Figure 2.22: A trace of the OneRowNim program

Execution of the OneRowNim program begins with the first statement in the main() method, labeled with line number 1. This statement declares a variable of type OneRowNim named game and calls a constructor OneRowNim() to create and initialize it. The constructor, which in this case is a default constructor, causes control to shift to the declaration of the instance variables nSticks and player in statements 2 and 3, and assigns them initial values of 7 and 1 respectively. Control then shifts back to the second statement in main(), which has the label 4. At this point, game refers to an instance of the OneRowNim class with an initial state shown in Figure 2.23. Executing statement 4 causes control to shift to the report() method where statements 5 and 6 use System.out.println() to write the following statements to the console.

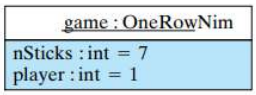


Figure 2.23: The initial state of game, a OneRowNim object.

```
    Number of sticks left: 7
    Next turn by player 1
```

Control shifts back to statement 7 in the main() method, which calls the takeThree() method, sending control to the first statement of that method. Executing statement 8 causes 3 to be subtracted from the int value stored in the instance variable nSticks of game, leaving the value of 4. Executing statement 9 subtracts the value stored in the playervariable, which is 1, from 3 and assigns the result (the value 2) back to player. The state of the object game, at this point, is shown in Figure 2.24. Tracing the remainder of the program follows in a similar manner. Notice that the main() method calls game.report() four different times so that the two statements in the report() method are both executed on four different occasions. Note also that there is no call of game.takeTwo() in main(). As a result, the two statements in that method are never executed.

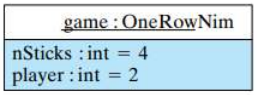


Figure 2.24: The state of game after line 9 is executed.

Object-Oriented Design: Basic Principles

We complete our discussion of the design and this first implementation of the OneRowNim class with a brief review of some of the object-oriented design principles that were employed in this example.

- **Encapsulation.** The OneRowNim class was designed to encapsulate a certain state and a certain set of actions. It was designed to simulate playing the One Row Nim game. In addition, OneRowNim's methods were designed to encapsulate the actions that make up their particular tasks.
- **Information Hiding.** OneRowNim's instance variables, nSticks and player are declared private so other objects can only change the values of these variables with the public methods of a OneRowNim instance. The bodies of the public methods are also hidden from users of OneRowNim instances. An instance and its methods can be used without any knowledge of method definitions.
- **Clearly Designed Interface.** OneRowNim's interface is defined in terms of the public methods. These methods constrain the way users can interact with OneRowNim objects and ensures that OneRowNim instances remain in a valid state. Those are the main purposes of a good interface.
- **Generality and Extensibility.** There is little in our design of that limits its use and its extensibility. Moreover, as we will see later, we can create several different kinds of user interfaces which interact with OneRowNim objects.

The OneRowNim class has some obvious shortcomings that are a result of our decision to limit methods to those without parameters or return values. These shortcomings include:

- A OneRowNim object cannot communicate to another object the number of remaining sticks, which player makes the next turn, or whether the game is over. It can only communicate by writing a report to the console window.
- The takeOne(), takeTwo() and takeThree() methods all have similar definitions. It would be a better design if a single method could take away a specified number of sticks.
- There is no way to play a OneRowNim game starting with a different number of sticks than 7. It would be nice to have a way of playing a game that starts with any number of sticks.
- In order to for a user to play a OneRowNim game, a user interface class would need to be developed that would allow the user to receive information about the state of the game and to input moves to make.

As we study other features of Java in the next two chapters, we will modify the OneRowNim class to address these identified shortcomings.

Special Topic: Alan Kay and the Smalltalk Language

Although *Simula* was the first programming language to use the concept of an object, the first pure object-oriented language was *Smalltalk*. Smalltalk was first started by Alan Kay in the late 1960s. Kay is an innovative thinker who has had a hand in the development of several advances, including windowing interfaces, laser printing, and the client/server model, all of which are now commonplace today.

One of the abiding themes throughout Kay's career has been the idea that computers should be easy enough for kids to use. In the late 1960s, while still in graduate school, Kay designed a computer model that consisted of a notebook-sized portable computer with a keyboard, screen, mouse, and high-quality graphics interface. He had become convinced that graphics and icons were a far better way to communicate with a computer than the command-line interfaces that were prevalent at the time.

In the early 1970s Kay went to work at the Xerox Palo Alto Research Center (PARC), where he developed a prototype of his system known as the *Dynabook*. Smalltalk was the computer language Kay developed for this project. Smalltalkwas designed along a biological model, in which individual entities or "objects" communicate with each other by passing messages back and forth. Another goal of Smalltalk was to enable children to invent their own concepts and build programs with them—hence, the name *Smalltalk*.

Xerox's management was unable to see the potential in Kay's innovations. However, during a visit to Xerox in 1979, Steve Jobs, the founder of Apple Computer, was so impressed by Kay's work that he made it the inspiration of the Macintosh computer, which was first released in 1984.

Kay left Xerox in 1983 and became an Apple Fellow in 1984. In addition to working for Apple, Kay spent considerable time teaching kids how to use computers at his Open School in West Hollywood. In 1996 Kay became a Fellow (an "Imagineer") at the Walt Disney Imagineering's Research and Development Organization, where he continues to explore innovative ways to enhance the educational and entertainment value of computers.

This page titled 2.5: Case Study- Simulating a Two-Person Game is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **2.5: Case Study- Simulating a Two-Person Game** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

2.6: From the Java Library- java.util.Scanner

If we wish to write useful interactive programs, we must be able to receive information from the user as well as send information to him or her. We saw, in the previous chapter, that output from a program can be sent to the console window by simply using the System.out.print() and System.out.println() statements. In this section we describe two simple ways that Java can handle keyboard input. Receiving input from the keyboard, together with sending output to the console window, creates one of the standard user interfaces for programs.

Recall, that in Java, any source or destination for I/O is considered a stream of bytes or characters. To perform keyboard input, we will extract characters from System.in, the input stream connected to the keyboard. Getting keyboard input from System.in involves two complications that are not present in dealing with System.out.println(). First, normal keyboard input data requested of a user consists of a sequence of characters or digits which represent a word, phrase, integer, or real number. Normally, an entire sequence of characters typed by the user will represent data to be stored in a single variable with the user hitting the return or enter key to signal the end of a piece of requested data. Java has a special class, BufferedReader, that uses an input stream and has a method that collects characters until it reads the character or characters that correspond to hitting the return or enter key. A second complication for reading input involves the problem of how to handle receiving data that is not in the same format as expected. The BufferedReaderclass handles this problem by using certain *exceptions*, a special kind of error message, that must be handled by the programmer. Chapter 11 is devoted to exceptions and we will avoid their use, as far as possible, until that time.

There is an alternate way to handle keyboard input in the Java 2 Platform Standard Edition 5.0 (Java SE 5.0). A Scannerclass has been added to the java.util package which permits keyboard input without forcing the programmer to handle exceptions. We introduce the Scanner class in the next subsection and then describe how a user defined class introduced in Chapter 4 can function in an equivalent fashion to permit simple keyboard input.

Keyboard Input with the Scanner Class

A partial definition of Scanner is shown in Figure 2.25. Note that the Scanner methods listed are but a small subset of the public methods of this class. The Scanner class is in the java.util package so classes that use it should import it with the following statement:

```
import java.util.Scanner;
```

The Scanner class is designed to be a very flexible way to recognize chunks of data that fit specified patterns from any input stream. To use the Scanner class for keyboard input, we must create a Scanner instance and associate it with System.in. The class has a constructor for this purpose, so the statement

```
Scanner sc = new Scanner(System.in);
```

declares and instantiates an object that can be used for keyboard input. After we create a Scanner object, we can make a call to nextInt(), nextDouble(), or next() to read, respectively, an integer, real number, or string from the keyboard. The program in Figure 2.26 demonstrates how an integer would be read and used.

```
import java.util.Scanner;

public class TestScanner
{
  public static void main(String[] args)
  {               // Create Scanner object
    Scanner sc = new Scanner(System.in);
    System.out.print("Input an integer:"); // Prompt
    int num = sc.nextInt();         // Read an integer
    System.out.println(num + " squared = " + num*num);
  } //main()
} // TestScanner class
```

Figure 2.26: A very brief program with a Scanner object used for keyboard input

When the nextInt() method is executed, no further statements are executed until an int value is returned by the method. Normally this does not happen until the user has typed in the digits of an integer and hit the return or enter key. Thus executing the main()method of the TestScanner class will result in the output

```
Input an integer:
```

to the console window and the program will wait for the user to type in an integer and hit the return or enter key. After this has been done the output will look something like:

```
Input an integer:123
123 squared = 15129
```

Keyboard input of real numbers and strings are handled in a similar manner.

Keyboard input will allow us to create examples of command line interfaces for interactive programs. For example, the code

```
Scanner sc = new Scanner(System.in);
Riddle riddle = new Riddle(
  "What is black and white and red all over?",
  "An embarrassed zebra.");
System.out.println("Here is a riddle:");
System.out.println(riddle.getQuestion());
System.out.print("To see the answer, ");  // Prompt
System.out.println("type a letter and enter.");
String str = sc.next();         // Wait for input
System.out.println(riddle.getAnswer());
```

will display a riddle question and prompt the user to type a letter and to hit the enter key to see the answer. In the next chapter, we will develop new methods for the OneRowNim class that will be able to use int values input from the keyboard for the next move.

We must mention that, since the Scanner class is designed as a flexible tool for recognizing chunks of data from any input stream, it has some properties that may be unexpected and not totally compatible with simple keyboard input. A Scanner object has a set of character strings that separate or **delimit** the chunks of data that it is looking for. By default, this set of delimiters consists of any non-empty sequence of *white space* characters, that is, the space, tab, return, and newline characters. This will allow a user to input several integers separated by spaces before hitting the enter key. This might be handled by code like:

```
System.out.print("Input two integers and an enter:");
int num1 = sc.nextInt();
int num2 = sc.nextInt();
```

White space as delimiters also means that the next() method cannot return an empty string nor can it return a string that contains any spaces. For example, consider the code:

```
System.out.print("Input the first president of the USA:");
String str = sc.next();
```

If one types "George Washington" and hits the enter key, the string str will store only "George". In order to get a Scanner object to read strings that contain spaces, we must use the useDelimiter() method to define the set of delimiters as just that character string generated by hitting the enter key. For example, for some Windows operating systems, the statement

```
sc = sc.useDelimiter("\r\n");
```

will result in the next() method returning the entire string of characters input from the keyboard up to but not including those generated by hitting the enter key.

You should also be aware that just because we can use a Scanner object to write Java code that ignores exceptions does not mean that exceptions will not be generated by keyboard input. If the user enters letters rather than digits for the nextInt() method to process, the program will be terminated with an error message.

It must be stressed that the strategy for handling keyboard input outlined above is a temporary strategy until the topic of exceptions is covered in Chapter 11. Real software applications that use keyboard input should carefully handle the possibility that a user will enter something unexpected. In Java, this can only be done by handling exceptions.

Keyboard Input with the KeyboardReader Class

If you are using an older version of Java that does not have the Scanner class, a user-defined class can be used instead. A KeyboardReader class that uses the BufferedReader class will be developed in Chapter 4. It has methods that read data from the keyboard in a manner very similar to those of the Scanner class. A partial list of its public methods is given in the UML class diagram shown in Figure 2.27. To use the KeyboardReader class for keyboard input, copy the source code KeyboardReader.java from Chapter 4 into the same directory as the source code of your current Java class (and add it to your current project if you are using a integrated development environment).

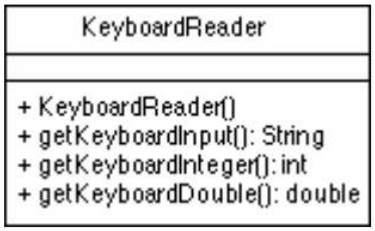


Figure 2.27: A UML class diagram of the KeyboardReader class.

To use a KeyboardReader object, we need to create an instance of the class with a constructor. Then calling one of the three methods will return an int, double, or String when data is input from the keyboard. Any of the three methods of a KeyboardReader object will attempt to process the entire string input from the keyboard up to the point that the enter key is hit. That is, the character or characters generated by hitting the return or enter key is the delimiter used by KeyboardReader. The TestKeyboardReader class definition in Figure 2.28 reads an integer from the keyboard and squares it just like the TestScanner class. In the remainder of the text, any time the Scanner class is used for keyboard input, the same program can be run using the KeyboardReader class after making the obvious substitutions.

```
public class TestKeyboardReader
{
  public static void main(String[] args)
  {             // Create KeyboardReader object
    KeyboardReader kb = new KeyboardReader();
    System.out.print("Input an integer:"); // Prompt
    int num = kb.getKeyboardInteger(); // Read an integer
    System.out.println(num + " squared = " + num*num);
  } //main()
} // TestKeyboardReader class
```

Figure 2.28: A very brief program with a KeyboardReader object used for keyboard input.

? Exercise 2.6.1

Modify the main() method of the TestScanner class so that it reads a real number from the keyboard rather than an integer.

Chapter Summary

access modifier

class-level variable

default value

delimiter

empty string

flow of control

interface

local variable

method call and return

null pointer

null pointer exception

pointer

reference

reference variable

static modifier

user interface

Dot notation is used to refer to an object's public elements.

Designing a class is a matter of deciding what role it will play and what information and actions it will have.

Writing a Java program is a matter of defining one or more classes. A class definition serves as a template for creating instance of the class.

Classes typically contain two kinds of elements, variables and methods. An object's state is defined by its instance variables.

Class elements that are declared public can be accessed by other objects. Elements that are declared private are hidden from other objects.

A class's instance variables are usually declared private so they cannot be accessed directly by other objects.

An object's public instance methods can be called by other objects. Thus, they make up the object's interface with other objects.

Object instantiation is the process of creating an object, using the new operator in conjunction with a constructor method.

A class definition consists of a header and a body. The header gives the class a name, specifies its accessibility (public), and its place in the Java class hierarchy (extends Object). The class body contains declarations of the class's variables and definitions of its methods.

By default, a newly defined class is consider a subclass of Object.

Class elements that are declared static, such as the main() method, are associated with the class (not with its instances).

A Java application program must contain a main() method, which is where it begins execution.

Methods that are used solely for the internal operations of the class should be declared private.

An instance variable declaration reserves memory for the instance variable within the object, associates a name and a type with the location, and specifies its accessibility.

A method definition consists of two parts: a header, which names the method and provides other general information about it, and a body, which contains its executable statements.

Declaring a variable creates a name for an object but does not create the object itself. An object is created by using the new operator and a constructor method.

The Java code fragment prints out the following:

```
The singing king.
```

For the Riddle class (Fig. [fig-riddleclass2]),

The name of the class: Riddle

The names of two instance variables: question, answer

The names of three methods: Riddle(), getQuestion(), getAnswer()

For RiddleUser class (Fig. [fig-riddleuser]),

The names of two Riddle instances: riddle1, riddle2

All six method calls of the Riddle objects in the program:

```
Riddle("What is black and white and red all over?",
                            "An embarrassed zebra.")
Riddle("What is black and white and read all over?",
                                     "A newspaper.")
riddle1.getQuestion()
riddle1.getAnswer()
riddle2.getQuestion()
riddle2.getAnswer()
```

Qualified names: riddle1.getQuestion() and riddle1.getAnswer()

Definition of new instance variable in the Riddle class:

```
private String hint = "This riddle is to easy for a hint";
```

The header for a getHint() method of the Riddle class, which should be a public method, is:

```
public String getHint();
```

The header for a setHint() method of the Riddle class is:

```
  public void setHint(String aHint);
```

The result type is void. Although the identifier used for the parameter is arbitrary, it is a good practice to make it descriptive, by referring in some way to the hint instance variable.

The partial definition of the Student class is given below.

```
public class Student
{  private String firstName;
   private String lastName;
   private int studentID;

   public void setStudent(String fName, String lName,
                                                int anID);
   public int getStudentID();
   public String getStudentName();
}
```

A main method that reads and squares a real number is given below.

```
public static void main(String[] args)
{               // Create Scanner object
  Scanner sc = Scanner.create(System.in);
  System.out.print("Input a real number:");    // Prompt
  double realNum= sc.nextDouble();      // Read a double
  System.out.println(num + " squared = " + realNum*realNum);
} //main()
```

Consider the transaction of asking your professor for your grade in your computer science course. Identify the objects in this transaction and the types of messages that would be passed among them.

Now suppose the professor in the previous exercise decides to automate the transaction of looking up a student's grade and has asked you to design a program to perform this task. The program should let a student type in his or her name and ID number and the program then should display his or her grades for the semester, with a final average. Suppose there are five quiz grades, three exams, and two programming exercise grades. Identify the objects in this program and the type of messages that would be passed among them. (*Hint*: The grades themselves are just data values, not objects.)

In the RiddleUser class (Fig. [fig-riddleuser]), give two examples of object instantiation and explain what is being done.

Explain the difference between a method definition and a method call. Give an example of each from the Riddle and RiddleUser examples discussed in this chapter.

In the RiddleUser class (Fig. [fig-riddleuser]), identify three examples of method calls and explain what is being done.

Describe how the slogan "define, create, manipulate" applies to the Riddle example.

An identifier is the name for a

,

, or a

.

Which of the following would be valid identifiers?

```
int  74ElmStreet  Big_N     L$&%#   boolean  Boolean  _number
Int  public       Private   Joe     j1       2*K      big numb
```

Explain the difference between a class variable and an instance variable.

Identify the syntax error (if any) in each declaration. Remember that some parts of an instance variable declaration are optional.

public boolean isEven ;

Private boolean isEven ;

private boolean isOdd

public boolean is Odd ;

string S ;

public String boolean ;

private boolean even = 0;

private String s = helloWorld ;

Write declarations for each of the following instance variables.

A private boolean variable named bool that has an initial value of true.

A public String variable named str that has an initial value of "hello".

A private int variable named nEmployees that is not assigned an initial value.

Identify the syntax error (if any) in each method header:

public String boolean()

private void String ()

private void myMethod

private myMethod()

public static void Main (String argv[])

Identify the syntax error (if any) in each assignment statement. Assume that the following variables have been declared:

```
public int m;
public boolean b;
public String s;
```

2

m = "86" ;

m = 86 ;

m = true ;

s = 1295 ;

s = "1295" ;

b = "true" ;

b = false

Given the following definition of the NumberAdder class, add statements to its main() method to create two instances of this class, named adder1 and adder2. Then add statements to set adder1's numbers to 10 and 15, and adder2's numbers to 100 and 200. Then add statements to print their respective sums.

```
public class NumberAdder
{
    private int num1;
    private int num2;

    public void setNums(int n1, int n2)
    {
      num1 = n1;
      num2 = n2;
    }
    public int getSum()
    {
      return num1 + num2 ;
    }

    public static void main(String args[])
    {
    }
}
```

For the NumberAdder class in the previous exercise, what are the names of its instance variables and instance methods? Identify three expressions that occur in the program and explain what they do. Identify two assignment statements and explain what they do.

Explain the difference between each of the following pairs of concepts.

A method definition and a method call.

Declaring a variable of reference type and creating an instance.

A variable of reference type and a variable of primitive type.

Define a Java class named NumberCruncher that has a single int variable as its only instance variable. Then define methods that perform the following operations on its number: get, double, triple, square, and cube. Set the initial value of the number with a constructor as was done with the instance variables in the Riddle class.

Write a main() method and add it to the NumberCruncher class defined in the previous problem. Use it to create a NumberCruncher instance, with a certain initial value, and then get it to report its double, triple, square, and cube.

Write a Java class definition for a Cube object, that has an integer attribute for the length of its side. The object should be capable of reporting its surface area and volume. The surface area of a cube is six times the area of any side. The volume is calculated by cubing the side.

Write a Java class definition for a CubeUser object that will use the Cube object defined in the previous exercise. This class should create three Cube instances, each with a different side, and then report their respective surface areas and volumes.

Challenge: Modify your solution to the previous exercise so that it lets the user input the side of the cube. Follow the example shown in this chapter's "From the Java Library" section.

Challenge: Define a Java class that represents an address book entry, Entry, which consists of a name, address, and phone number, all represented as Strings. For the class's interface, define methods to set and get the values of each of its instance variables. Thus, for the name variable, it should have a setName() and a getName() method.

Draw a UML class diagram to represent the following class hierarchy: There are two types of languages, natural languages and programming languages. The natural languages include Chinese, English, French, and German. The programming languages include Java, Smalltalk and C++, which are object-oriented languages, FORTRAN, COBOL, Pascal, and C, which are imperative languages, Lisp and ML, which are functional languages, and Prolog, which is a logic language.

Draw a UML class diagram to represent different kinds of automobiles, including trucks, sedans, wagons, SUVs, and the names and manufacturers of some popular models in each category.

Draw a UML object diagram of a triangle with attributes for three sides, containing the values 3, 4, and 5.

Suppose you are writing a Java program to implement an electronic address book. Your design is to have two classes, one to represent the user interface and one to represent the address book. Draw a UML diagram to depict this relationship. See Figure [fig-p63f2].

Draw an UML object diagram to depict the relationship between an applet, which serves as a user interface, and three Triangles, named t1, t2, and t3.

This page titled 2.6: From the Java Library- java.util.Scanner is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **2.6: From the Java Library- java.util.Scanner** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

CHAPTER OVERVIEW

3: Methods- Communicating with Objects

Learning Objectives

After studying this chapter, you will

- Understand the role that methods play in an object-oriented program.
- Know how to use parameters and arguments to pass data to an object.
- Understand how constructor methods are used to instantiate objects.
- Know the difference between passing a value and passing a reference to an object.
- Be able to design your own methods.
- Know how to use the if-else and while control structures.

3.1: Introduction
3.2: Passing Information to an Object
3.3: Constructor Methods
3.4: Retrieving Information from an Object
3.5: Passing a Value and Passing a Reference
3.6: Flow of Control- Control Structures
3.7: Testing an Improved OneRowNim
3.8: From the Java Library- java.lang.Object
3.9: Object-Oriented Design- Inheritance and Polymorphism
3.10: Drawing Lines and Defining Graphical Methods (Optional)

This page titled 3: Methods- Communicating with Objects is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

3.1: Introduction

In this chapter, we take a look at Java methods and parameters. Methods and parameters are the primary mechanisms for passing information into and out of an object. We will once again focus on the OneRowNim simulation that we designed in the previous chapter. That version was sufficient to introduce us to Java objects and classes, but it was limited in its ability to communicate with other objects.

In this chapter, we want to expand OneRowNim to make our simulation more realistic. We begin by learning how to pass information to an object. That will enable us to specify the number of sticks to remove using a single method. We then consider special methods called constructors, which are used to initialize an object's state when it is created. We also learn how to retrieve information from an object. That will enable us to request a OneRowNim object for several different bits of information. Then we consider the if-else and while control structures which allow us to define more useful methods and write more realistic test programs.

3.1: Introduction is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

- **3: Methods- Communicating with Objects** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

3.2: Passing Information to an Object

One convention of object-oriented programming is to provide public methods to *set* and *get* the values of some of its private instance variables. Methods that set or modify an object's instance variables are called **mutator methods**. Methods that get or retrieve the value of an instance variable are called **accessor methods**.

JAVA EFFECTIVE DESIGN

Accessor and Mutator Methods. An accessor method is a public method used to get the value of an object's instance variable. Such methods are often named getVarName() where VarName is the name of the variable that's being accessed. A mutator method is a public method used to modify the value of one or more instance variables. The special type of mutator method that sets or assigns a variable a specified value is often called setVarName().

It is up to the designer of the class to determine which private variables require accessor and mutator methods. If you were designing a BankAccount class, you might want a public getAccountNumber() method, so that clients could retrieve information about their bank accounts, but you would probably not want a public getAccountPassword() method or a public setAccountBalance() method.

In the remainder of this section, we will be concerned with mutator methods. We defined three mutator methods named takeOne(), takeTwo(), and takeThree as part of the OneRowNim class in the previous chapter. All three of these method change the values of the instance variables nSticks and player. All three methods have very similar bodies. The definition of the takeOne() is:

```
public void takeOne()
{   nSticks = nSticks - 1;
    player = 3 - player;
 }
```

The only difference in the bodies of the other two methods is that they subtract 2 and 3 from nSticks instead of 1. Instead of having three, virtually identical methods, It would be a more efficient design to define a single method where the number to be subtracted from nSticks would be supplied as an argument when the method is called. In order to be able to handle such an argument, we must design a new method that uses a *parameter* to handle the argument.

A **formal parameter**, or more simply, **parameter**, is a variable used to pass information into a method when the method is invoked. The *type* and *variable name* of the formal parameter must appear in the *formal parameter list* that follows the method's name in the method header. The formal parameter is used to hold a value that it is passed while the method is executing.

JAVA LANGUAGE RULE

Formal Parameter. A formal parameter is a variable that serves as a storage location for information that is passed to a method. To specify a formal parameter, you must provide a type identifier followed by variable identifier, and you must place this declaration inside the parentheses that follow the method's name.

Consider the following definition for a takeSticks() method:

```
public void takeSticks(int num)
{    nSticks = nSticks - num;
     player = 3 - player;
}
```

Notice that executing the body of takeSticks() when the parameter num stores the value 1 accomplishes precisely the same task as executing takeOne(). If, instead, a value of 2 or 3 is stored in num, then calling the method acts like takeTwo() or takeThree() respectively. Thus, using parameters enables us to design methods that are more general in what they do, which is an important principle of good program design.

Another example of a mutator method is one in which define a *set method* to allow the starting number of sticks to be set for an instance of OneRowNim. For this, we could define:

```
public void setSticks(int sticks)
{    nSticks = sticks;
} // setSticks()
```

As we will see in Section 3.3, we can also define a constructor method that can be used, when the game is created, to set the initial value of nSticks. It is often desirable to have more than one method that sets the values of an objects' instance variables.

If a method uses more than one parameter, use a comma to separate the individual parameter declarations in the method header. For example, if we wanted a method for OneRowNim that specified both the number of sticks for the start of a game and which player takes a turn first, it could be defined:

```
public void setGame(int sticks, int starter)
{   nSticks = sticks;
    player = starter;
} // setGame()
```

The Scope of Parameters, Variables, and Methods

The bodies of the mutator methods in the previous section make use of both instance variables and parameters. It is important to note that there is a difference in where these two types of variables can be used. The **scope** of a variable or method refers to where it can be used in a program.

A parameter's scope is limited to the body of the method in which it is declared. Variables that are declared in the body of a method have scope which extends from the point where they are declared to the end of the block of code in which they are declared. Parameters are local variables which are declared in the parameter list of a method's header and which have initial values specified by the arguments in a method call. The scope of a parameter is the same as the scope of a variable declared at the very beginning of the body of a method. Once the flow of execution leaves a method, its parameters and other local variables cease to exist. The scope of local variables is referred to as **local scope**.

JAVA LANGUAGE RULE

Scope. Local variables, that is, parameters and variables declared in the body of a method, have local scope which extends from the point at which they are defined to the end of the block of code in which they are defined. In particular, the scope of a parameter is the entire body of the method in which it is declared.

By contrast, instance variables, class variables, and all methods have scope that extends throughout the entire class, that is, **class scope**. They can be used in the body of any method and in the expressions that assign initial values to class level variables. There are two restrictions to remember. First, instance variables and instance methods cannot be used in the body of a class method, one modified with static, unless an instance of the class is created there and then the dot notation of qualified names must be used to refer to the variable or method. This is because class methods are called without reference to a particular instance of the class. The main() method of the OneRowNim class that we defined in the previous chapter is an example of such a class method. In that case, to test the instance methods of OneRowNim we first created an instance of OneRowNim and used it to call its instance methods:

```
OneRowNim game = new OneRowNim(); // Create instance
game.report();   // Call an instance method
```

The second restriction involved in class scope is that one class level variable can be used in the expression that initializes a second class level variable only if the first is declared before the second. There is no similar restriction on methods.

JAVA LANGUAGE RULE

Scope. Class level variables, that is, instance variables and class variables have class scope, which extends throughout the class. Methods also have class scope.

Except for the restrictions noted above, methods and class level variables can be referred to within the same class by their simple names, with just the method (or variable) name itself, rather than by their qualified names, with the dot operator. Thus, in OneRowNim, we can refer to nSticks and report() in the bodies of other instance methods. In a class method, such as main(), we would have to create an instance of OneRowNim with a name like game and refer to game.report().

JAVA LANGUAGE RULE

Qualified Names. Within the same class, references to class methods or class variables can be made in terms of simple names. Within the bodies of instance methods, references to instance variables and references to other instance methods can also be made in terms of simple names. However, within the bodies of class methods, qualified names, or dot notation, must be used to refer to instance methods or instance variables just like how they are referred to in other classes.

JAVA DEBUGGING TIP

Scope Error. It would be a syntax error to refer to a method's parameters or other local variables from outside the method.

Arguments and Parameters

The new class definition for OneRowNim is given in Figure 3.1. Note that now that we have a single method, takeSticks(), that can be used to take away a variable number of sticks, we have removed the three methods we wrote in the previous chapter, takeOne(), takeTwo(), and takeThree(), from OneRowNim. Using a single method, with a parameter, is clearly a better design. To see this, just imagine what we would have to do if we didn't use a parameter and we wanted to be able to take away four sticks, or five, or more. If we didn't have parameters, we'd have to write a separate method for each case, which is clearly a bad idea. Using parameters in this way leads to a more general useful method and thus is an example of the generality principle.

```
public class OneRowNim
{ private int nSticks = 7; // Start with 7 sticks
  private int player = 1;  //Player 1 plays first

  public void takeSticks(int num)
  { nSticks = nSticks - num;
    player = 3 - player;
  }  // takeSticks()

  public void report()
  { System.out.println("Number of sticks left: " + nSticks);
    System.out.println("Next turn by player " + player);
  }   // report()
} //OneRowNim1 class
```

Figure 3.1: The OneRowNim class definition with takeSticks() method.

Now let's consider how we would create a OneRowNim instance and use the new method in the main() method or in a different class. If we want to have an instance of OneRowNim object to remove 3 sticks on the first move by using the takeSticks() method, we need to pass the int value 3 to the method. In order to effect this action, we would use the following statements:

```
OneRowNim game = new OneRowNim();
game.takeSticks(3);
```

Because the definition of takeSticks() includes a single int parameter, we must supply a single int value (such as 3), when we invoke it. When the method is invoked, its formal parameter (num) will be set to the value we supply (3). The value we supply does not have to be a literal int value. We can supply any expression or variable that evaluates to an int value. For example:

```
int val = 7 - 5;
game.takeSticks(val);
```

In this case, the value being passed to takeSticks() is 2, the value that val has at the time the method call is made.

It would be an error to try to pass a value that was not a int to takeSticks(). For example, each of the following invocations of takeSticks() results in a syntax error:

```
game.takeSticks();     // no argument is supplied
game.takeSticks("3");  // "3" is a String, not an int
game.takeSticks(int);  // int not is an int value
```

As you recall from Chapter 0, the value that is passed to a method when it is invoked is called an argument. Even though the terms argument and parameter are sometimes used interchangeably, it will be useful to observe a distinction. We will use the term *parameter* to refer to the formal parameter—the variable used to pass data to a method—that occurs in the method definition. We use the term *argument* to refer to the actual value that is supplied when the method is invoked.

JAVA DEBUGGING TIP

Type Error. It would be a syntax error to use an argument whose type doesn't match the type of its corresponding parameter.

The distinction between parameter and argument is related to the difference between *defining* a method and *invoking* a method. Defining a method is a matter of writing a method definition, such as

```
public void printStr(String s)
{   System.out.println(s);
}
```

This definition defines a method that takes a single String parameter, s, and simply prints the value of its parameter. On the other hand, invoking a method is a matter of writing a method call statement, such as

```
printStr("HelloWorld");
```

This statement calls the printStr() method and passes it the string "HelloWorld". This notation assumes that the call to the instance method printStr() is made within the body of another instance method of the same class.

Passing an int value to a OneRowNim method.

To get a clearer picture of the interaction that takes place when we invoke takeSticks() and pass it an int value, let's write a main() method to test our new version of OneRowNim.

Our first version of main() is shown in Figure 3.2. We will use it to trace how the parameter of takeSticks() interacts with the instance variables nSticks and player. The statements in the main() program simply create an instance of OneRowNim that is referenced by game and invoke the setSticks() method with an argument of 3.

```
public static void main (String argv[])
{  OneRowNim game;        // Declare a OneRowNim object
   game = new OneRowNim(); // Instantiate the references
   game.takeSticks(3);     // remove 3 sticks
} // main()
```

Figure 3.2: A main() method to test the takeSticks() method.

To get a clearer understanding of how a parameter works, it will be instructive to trace through the code in main(). Figure 3.3 displays how the states of the instance variables of game and the parameter of takeSticks() interact.

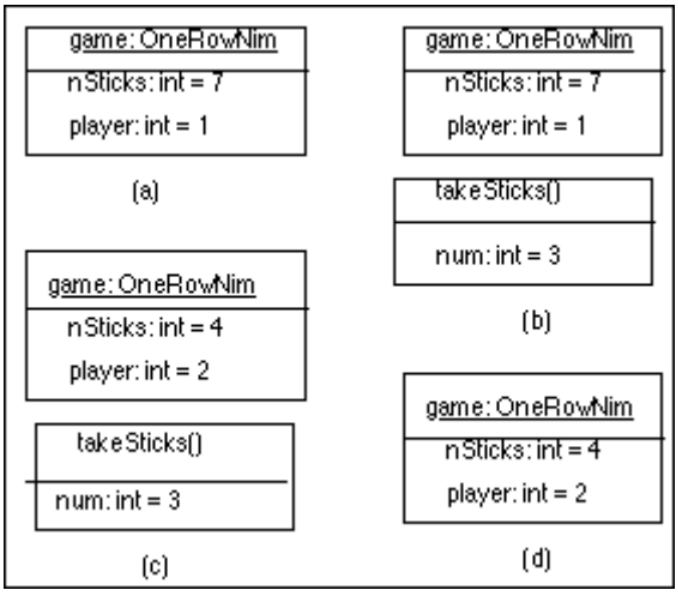


Figure 3.3: Tracing the state of game (a) Just before calling takeSticks(3). (b) Just before executing the body of takeSticks(3). (c) Just after executing the body of takeSticks(3). (d) After flow of control leaves takeSticks().

Executing the first two statements of main() creates the instance game of OneRowNim. Figure 3.3(a) shows the initial state of game. When the takeSticks(3) method call is made, a parameter (which is a local variable) named num is created and the value 3 is stored in it. The state of the instance variables and the parameter are shown in (b). Then the body of takeSticks() is executed. The new state of game is shown in (c). After the flow of control leaves the body of takeSticks() and returns to main(), the memory location which stored the value of the parameter num is released for other uses. The state of game at this point is shown in (d). Notice that the value of nSticks has been reduced to 4.

Passing keyboard input to takeSticks()

To complete this section, let's modify our main() method in Figure 3.2 so that it prompts the user to input an integer from the keyboard and then uses a Scanner object, introduced in the previous chapter, to read the integer. That integer will then be used as the argument in a call to takeSticks(). These modifications have been incorporated into the revised version of the main() method shown in Figure 3.4. If we now run this program the following output will be generated in the console window before waiting for keyboard input:

```
Number of sticks left: 7
Next turn by player 1
Input 1, 2, or 3 and hit enter:
```


https://eng.libretexts.org/@go/page/136116

```
import java.util.Scanner;

public static void main (String argv[])
{ Scanner sc;             // Declare a Scanner variable
  sc = Scanner.create(System.in); // Instantiate it
  OneRowNim game;         // Declare a OneRowNim variable
  game = new OneRowNim(); // Instantiate it
  game.report();          // Report state of game
  System.out.println("Input 1, 2, or 3 and hit enter:");
  int num = sc.nextInt(); // Read an int from keyboard
  game.takeSticks(num);   // Use the value read
  game.report();          // Report state of game
} // main()
```

Figure 3.4: A main() method with keyboard input for OneRowNim.

If the user then inputs a 2 from the keyboard, that input will be read and the takeSticks() method will remove 2 sticks. The output in the console window will now look like:

```
Number of sticks left: 7
Next turn by player 1
Input 1, 2, or 3 and hit enter:2
Number of sticks left: 5
Next turn by player 2
```

? Exercise 3.2.1

Explain the difference between a *method declaration* and a *method invocation*.

? Exercise 3.2.2

Explain the difference between a *formal parameter* and an *argument*.

? Exercise 3.2.3

Modify the OneRowNim class of Figure 3.4 by adding two instance variables of type String to store names of the two players. Choose names for the instance variables that would be appropriate for storing names for player one and player two. Write a method named setNames() with two string parameters which assigns the first parameter to the instance variable that you created for the name of player one. The second parameter should be assigned to the other new instance variable.

? Exercise 3.2.4

Write a statement that calls the setName() method of the previous exercise and sets the name of player one of game to “Xena” and sets the name of player two to “Yogi”.

This page titled 3.2: Passing Information to an Object is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **3.2: Passing Information to an Object** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

3.3: Constructor Methods

In the previous section, we looked at several examples of mutator methods that change the values of private instance variables of an object. It is possible to define mutator methods to set the initial values of instance variables after an object is created, but initial values can also be set by constructors.

As you recall from Chapter 0, a **constructor** method is used to create an instance (or object) of a class and to assign initial values to instance variables. A constructor declaration looks just like a method definition except it must have the same name as the class, and it cannot declare a result type. Unlike the class level variables and methods of a class, constructors are not considered members of the class. Therefore, they are not inherited by a class's subclasses. Access to constructors is governed by the access modifiers public and private. Here is a simple constructor for our OneRowNim class:

```
public OneRowNim()
{   nSticks = 7;
    player = 1;
}
```

This constructor merely sets the initial values of the instance variables, nSticks and player. In our current version of OneRowNim these variables are given initial values by using initializer statements when they are first declared:

```
 private int nSticks = 7;
 private int player = 1;
```

So we now have two ways to initialize a class's instance variables. In the OneRowNim class it doesn't really matter which way we do it. However, the constructor provides more flexibility because it allows the state of the object to be initialized at runtime. Of course, it would be somewhat redundant (though permissible) to initialize the same variable twice, once when it is declared and again in the constructor, so we should choose one or the other way to do this. For now, let's stick with initializing the instance variables when they are declared.

JAVA EFFECTIVE DESIGN

Constructors. Constructors provide a flexible way to initialize an object's instance variables when the object is created.

A constructor cannot return a value and, therefore, its declaration cannot include a return type. Because they cannot return values, constructors cannot be invoked by a regular method invocation. Instead, constructors are invoked as part of an *instance creation expression* when instance objects are created. An instance creation expression involves the keyword new followed by the constructor invocation:

```
OneRowNim game // Declare
    = new OneRowNim(); // and instantiate game1
OneRowNim game2 // Declare
    = new OneRowNim(); // and instantiate game2
```

Note here that we have combined variable declaration and instantiation into a single statement, whereas in some previous examples we used separate declaration and instantiation statements. Either way is acceptable.

JAVA LANGUAGE RULE

Constructors. Constructors cannot return a value. Therefore, no return type should be declared when the constructor is defined.

JAVA DEBUGGING TIP

When to Use Return. All method definitions except constructors must declare a return type.

Constructors should be used to perform the necessary initialization operations during object creation. In the case of a OneRowNim object, what initializations could be performed? One initialization that would seem appropriate is to initialize the initial number of sticks to a number specified. In order to do this, we would need a constructor with a single int parameter:

```
public OneRowNim(int sticks)
{   nSticks = sticks;
}
```

Now that we have this constructor we can use it when we create instances of OneRowNim:

```
OneRowNim game1 = new OneRowNim(21);
OneRowNim game2 = new OneRowNim(13);
```

The effect of these statements is the same as if we had used the setSticks() method that was discussed briefly on section 2.5. The difference is that we can now set the number of sticks when we create the object.

Should we keep the preceding constructor, or keep the setSticks() method or keep both in our class definition? The constructor can only be invoked as part of a new statement when the object is created but the setSticks() method could be called anytime we want. In many cases, having redundant methods for doing the same task in different ways would be an asset, because it allows for more flexibility in how the class could be used. However, for a game like One Row Nim, a major concern is that the two instance variables get changed only in a manner consistent with the rules for One Row Nim. The best way to guarantee this is to have takeSticks() as the only method that changes the instance variables nSticks and player. The only time that it should be possible to set the number of sticks for a game is when a constructor is used to create a new instance of OneRowNim.

? Exercise 3.3.1

What's wrong with the following constructor definition?

```
public void OneRowNim(int sticks)
{   nSticks = sticks;
}
```

? Exercise 3.3.2

Change the OneRowNim(int sticks) constructor so that it sets the number of sticks and also have it also set player two as the player who takes the first turn.

Default Constructors

As we noted in Chapter 2, Java automatically provides a *default constructor* when a class does not contain a constructor.

JAVA LANGUAGE RULE

Default Constructor. If a class contains no constructor declarations, Java will automatically supply a default constructor. The default constructor takes no parameters. If the class is public, the default constructor will also be public and, hence, accessible to other objects.

The default constructor's role is simply to create an instance (an object) of that class. It takes no parameters. In terms of what it does, the default constructor for OneRowNim would be equivalent to a public constructor method with an empty body:

```
public OneRowNim() { }
```

This explains why the following statement was valid when a class definition of OneRowNim contained no explicit definition of a constructor:

```
OneRowNim game = new OneRowNim();
```

Constructor Overloading and Method Signatures

It is often quite useful to have more than one constructor for a given class. For example, consider the following two OneRowNim constructors:

```
public OneRowNim() {} // Constructor #1

public OneRowNim(int sticks)   // Constructor #2
{   nSticks = sticks;
}
```

The first is an explicit representation of the default constructor. The second is the constructor we defined earlier to initialize the number of sticks in a OneRowNim object. Having multiple constructors lends flexibility to the design of a class. In this case, the first constructor merely accepts OneRowNim's default initial state. The second enables the user to initialize the number of sticks to something other than the default value.

In Java, as in some other programming languages, when two different methods have the same name, it is known as **method overloading**. In this case, OneRowNim is used as the name for two distinct constructor methods. What distinguishes one constructor from another is its *signature*, which consists of its name together with the number and types of formal parameters it takes. Thus, our OneRowNim constructors have the following distinct signatures:

```
OneRowNim()
OneRowNim(int)
```

Both have the same name, but the first takes no parameters, whereas the second takes a single int parameter.

The same point applies to methods in general. Two methods can have the same name as long as they have distinct signatures. A **method signature** consists of its name, and the number, types, and order of its formal parameters. A class may not contain two methods with the same signature, but it may contain several methods with the same name, provided each has a distinct signature.

JAVA LANGUAGE RULE

Method Signature. A method signature consists of the method's name, plus the number, types, and order of its formal parameters. A class may not contain two methods with the same signature.

There is no limit to the amount of overloading that can be done in designing constructors and methods. The only restriction is that each method have a distinct signature. For example, suppose in addition to the two constructors we have already defined, we want a constructor that would let us set both the number of sticks and the player who starts first. The following constructor will do what we want:

```
public OneRowNim(int sticks, int starter)
{   nSticks = sticks; // Set the number of sticks
    player = starter; // Set who starts
}
```

When calling this constructor, we would have to take care to pass the number of sticks as the value of the first argument and either 1 or 2 as the value of the second argument:

```
OneRowNim game3 = new OneRowNim(14, 2);
OneRowNim game4 = new OneRowNim(31, 1);
```

If we mistakenly reversed 14 and 2 in the first of these statements, we would end up with a OneRowNim game that starts with 2 sticks and has player 14 as the player with the first move.

We have now defined three constructor methods for the OneRowNim class. Each constructor has the name OneRowNim, but each has a distinct signature:

```
OneRowNim()
OneRowNim(int)
OneRowNim(int, int)
```

Constructor Invocation

A constructor method is invoked only as part of a new expression when an instance object is first created. Each of these is a valid invocation of a OneRowNim constructor:

```
                  // Default constructor
OneRowNim game1 = new OneRowNim();
                  // Sets number of sticks
OneRowNim game2 = new OneRowNim(21);
                  // Sets both instance variables
OneRowNim game3 = new OneRowNim(19, 2);
```

The following constructor invocations are invalid because there are no matching constructor definitions:

```
                  // No matching constructors
OneRowNim game4 = new OneRowNim("21");
OneRowNim game5 = new OneRowNim(12, 2, 5);
```

In the first case, there is no constructor method that takes a String parameter, so there's no matching constructor. In the second case, there is no constructor that takes three int arguments. In both cases, the Java compiler would complain that there is no constructor method that matches the invocation.

JAVA DEBUGGING TIP

Method Call. The signature of the method call—its name and the number, types, and order of its arguments—must exactly match the signature of the method definition.

This page titled 3.3: Constructor Methods is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **3.3: Constructor Methods** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

3.4: Retrieving Information from an Object

The modifications we've made to the OneRowNim class allow us to set the instance variables of a OneRowNim object with a constructor, but there is no way for us to retrieve their values other than to use the report() method to write a message to the console. We will want to be able to ask a OneRowNim object to provide us with the number of sticks remaining and who plays next when we develop a graphical user interface for OneRowNim in the next chapter. We declared nSticks and player as private variables, so we cannot access them directly. Therefore, we will need accessormethods to *get* the values of each of the instance variables. Consider the following method definitions:

```
public int getSticks()
{   return nSticks;
}

public int getPlayer()
{   return player;
}
```

Recall that a method's *ResultType* is specified just in front of the *MethodName*. We want the two methods to return intvalues that represent OneRowNim's instance variables. Therefore, their result types are both declared int.

Before we discuss how the value that is returned by a method is used when the method is called, let's consider one more method definition. Many methods that return a value do a computation rather than simply returning the value of an instance variable. For example, suppose we wish to define a method for the OneRowNim class that will notify the user of an instance of the class whether the game is over. Thus we want a method that, when called, returns a true or false depending on whether or not all the sticks have been taken. gameOver() is a descriptive name of such a method and the method should have a boolean result type. This method should return true when the instance variable nSticks no longer contains a positive int value. Thus we can define:

```
public boolean gameOver()
{   return (nSticks <= 0);
}
```

The expression (nSticks <= 0) evaluates to a false value if nSticks stores a positive value and it evaluates to trueotherwise. Thus the value returned is precisely what is required.

Invoking a Method That Returns a Value

When we invoke a method that returns a value, the invocation expression takes on, or is replaced by, the value that is returned. For example, if we execute the statements

```
OneRowNim game1 = new OneRowNim(11);
int sticksLeft = game1.getSticks();
```

the expression game1.getSticks() will take on the value 11 after the getSticks() method is finished executing. At that point, the second statement above can be treated as if expression is replaced with the value 11, which is assigned to sticksLeft. In effect, the second statement is equivalent to the following statement:

```
int sticksLeft = 11;
```

JAVA LANGUAGE RULE

Evaluating Method Calls. A nonvoid method call is an expression that has a value of a particular type. After the method is executed, the method call expression becomes the value returned.

We can use a value returned by a method call the same way we use a literal value of the same type. It can be assigned to variables, be part of a numerical expression, or be an argument of another method. All of the following statements involve valid calls of methods that return values:

```
int fewerSticks = game1.getSticks() - 1;
boolean done = game1.gameOver();
System.out.println(game1.getPlayer());
game1.getSticks();
```

In each statement, the method call can be replaced with the value it returns. Notice that the last statement is valid but does nothing useful. In Java and some other languages like C and C++, methods that return a value can simply be called without making use of the value returned. This may be useful to do if the method changes the state of instance variables or sends a message to another object or an output device. The method getSticks() does nothing but return the value of nSticks, so simply calling the method accomplishes nothing.

An Expanded OneRowNim Class

Let's add the new methods that return values to our OneRowNim class. We might note that the report() method from the previous chapter displays the values of nSticks and player in the console window which now could be done by using the methods getSticks() and getPlayer() with System.out.println(). However, calling report() is an easy way to display the values of both instance variables but it cannot provide access to either variable as an int value. So let's keep all three methods in our class definition. The inconvenience of a small amount of redundancy is outweighed by the added flexibility of being able to call all three methods.

JAVA EFFECTIVE DESIGN

Using Redundancy. Incorporating some redundancy into a class, such as providing more than one way to access the value of an instance variable, makes the class more widely usable.

Figure 3.5 provides a UML class diagram of the expanded OneRowNim class.

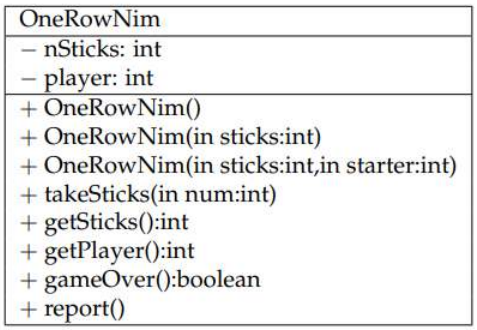


Figure 3.5: A UML class diagram for the expanded OneRowNim

Let's also consider a new main() method to test the new methods of the class. A very short list of statements that call each of the three new methods returning values is given in the main() method in Figure 3.6.

```
public static void main(String[] args)
{ OneRowNim game = new OneRowNim(13,2);
  game.takeSticks(2);
  System.out.print("The game is over is: ");
  System.out.println(game.gameOver());
  System.out.print("The next turn is by player: ");
  System.out.println(game.getPlayer());
  System.out.print("Sticks remaining: ");
  System.out.println(game.getSticks());
} //main()
```

Figure 3.6: A main() method that tests the new methods for OneRowNim

The output to the console when this program is run will be:

```
  The game is over is: false
  The next turn is by player: 1
  Sticks remaining: 11
```

Note that the constructor sets player to 2, so player stores the value 1 after one turn.

? Exercise 3.4.1

What would these segments of Java code display on the screen?

```
OneRowNim myGame = new OneRowNim(10,2);
System.out.println(myGame.getPlayer());
System.out.println(2 * myGame.getSticks());
System.out.println(myGame.gameOver());
```

? Exercise 3.4.2

Suppose that an int instance variable named nMoves is added to the OneRowNim class that counts the number of moves taken in a One Row Nim game. Write a Java method for the OneRowNim class to *get* the value stored in nMoves.

? Exercise 3.4.3

Write a method for the OneRowNim class called playerOneGoesNext() that returns a boolean value. The value returned should be true if and only if player one has the next turn.

This page titled 3.4: Retrieving Information from an Object is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **3.4: Retrieving Information from an Object** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

3.5: Passing a Value and Passing a Reference

The effect of passing arguments to a method differs depending on whether you are passing a value of primitive type (such as 5 or true) or a value of reference type (such as “Hello” or game1). When an argument of primitive type is passed to a method, a copy of the argument is passed to the formal parameter. For example, consider the class shown in Figure 3.7. Note that we have an int variable k, which initially stores the value 5, and a method myMethod(), which takes an int parameter n. In this case, when we invoke myMethod(k), k’s value (5) is copied into n and stored there during the method.

```
public class PrimitiveCall
{
   public static void myMethod(int n)
   {   System.out.println("myMethod: n= " + n);
       n = 100;
       System.out.println("myMethod: n= " + n);
   } // myMethod()

   public static void main(String argv[])
   {   int k = 5;
       System.out.println("main: k= " + k);
       myMethod(k);
       System.out.println("main: k= " + k);
   } // main()
} // PrimitiveCall
```

Figure 3.7: Passing a primitive value to a method.

One implication of passing a copy of a primitive value to a method is that the original value of k in main() cannot be altered from inside the method. Thus, the output generated by PrimitiveCall would be

```
main: k= 5
myMethod: n= 5
myMethod: n= 100
main: k= 5
```

Note that in main(), k’s value is printed both before and after myMethod() is called, but that its value remains unaffected even though n’s value is changed within the method. This is because myMethod() contains just a *copy* of k’s value, not k itself. Any changes to the copy within myMethod() leave k unaltered (See Fig. 3.8).

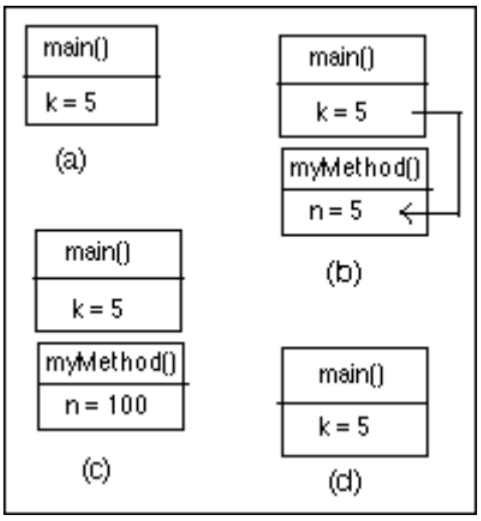


Figure 3.8: Tracing the state of variables k and n in PrimitiveCall (a) Just before calling myMethod(k) in main. (b) Just before executing the body of myMethod(). (c) Just after executing the body of myMethod(). (d) After flow of control returns to main().

JAVA LANGUAGE RULE

Passing a Primitive Value. When a value of a primitive type, like boolean or int, is passed to a method, a copy of the value is passed. That's why its original value remains unchanged outside the method, even if the copy is changed inside the method.

In contrast to this, when an argument of a reference type is passed to a method, a copy of the reference to the object itself is assigned to the parameter. For example, in the case of a String parameter or a OneRowNim parameter, the method would be given a reference to the object–that is, the address of the object. The object itself is *not* passed, because it would be too inefficient to copy the entire object with all its data and methods. However, because the object's reference gives the object's location in memory, the method will have access to the object and can make changes to the original object from within the method.

For example, consider the ReferenceCall class (Fig. 3.9). In this case, myMethod() takes a parameter g of type OneRowNim.

```
public class ReferenceCall
{
    public static void myMethod(OneRowNim g)
    {   System.out.print("myMethod: Number of sticks: ");
        System.out.println(g.getSticks());
        g.takeSticks(3);
        System.out.print("myMethod: Number of sticks: ");
        System.out.println(g.getSticks());
    } // myMethod()

    public static void main(String argv[])
    {   OneRowNim game = new OneRowNim(10);
         System.out.print("main: Number of sticks: ");
         System.out.println(game.getSticks());
         myMethod(game);
         System.out.print("main: Number of sticks: ");
         System.out.println(game.getSticks());
    }// main()
} // ReferenceCall
```

Figure 3.9: Passing a reference value to a method.

Because a instance is an object, g is a reference variable. So when myMethod(game) is invoked in main(), a reference to game is passed to myMethod(). Note that in myMethod(), we use takeSticks(3) to change the number of sticks of g from 10 to 7 and that this change persists even after the method returns control to main(). The reason is that during the method's execution, both game and g refer to the exact same object (see Fig. 3.10).

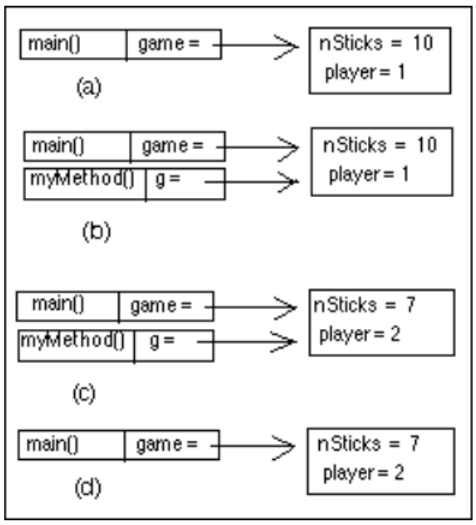


Figure 3.10: Tracing the state of OneRowNim object in ReferenceCall (a) Just before calling myMethod(game). (b) Just before executing the body of myMethod(). (c) Just after executing the body of myMethod(). (d) After flow of control returns to main().

The output generated by ReferenceCall would be

```
main: Number of sticks: 10
myMethod: Number of sticks: 10
myMethod: Number of sticks: 7
main: Number of sticks: 7
```

This illustrates that when passing a reference variable to a method, it is possible for the method to change the state of the object associated with the reference variable. In subsequent chapters we will see ways to make use of this feature of reference parameters.

JAVA LANGUAGE RULE

Passing a Reference. When a reference to an object is passed to a method, any changes made to the object from within the method will persist when the method is finished executing.

JAVA DEBUGGING TIP

Side Effects. An unintended change to an object is called a side effect. Care should be taken in designing methods that the method does not produce unwanted side effects in objects passed as reference parameters.

This page titled 3.5: Passing a Value and Passing a Reference is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **3.5: Passing a Value and Passing a Reference** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

3.6: Flow of Control- Control Structures

We have been ignoring a couple of problems with the definition of the OneRowNim class. One problem is that we would describe a One Row Nim game as two players taking turns until there are no more sticks. An object using OneRowNim would need a way to repeatedly execute a group of statements. One command in Java that controls the repetition of a block of statements is called a *while loop*. We will consider it later in this section.

A second problem is with the definition of takeSticks():

```
public void takeSticks(int num)
{   nSticks - num;
    player = 3 - player;
}
```

It is possible to call this method with an argument greater than 3 or less than 1. The call game.takeSticks(5) will remove 5 sticks even though the rules of One Row Nim say that you must remove 1, 2, or 3. While one might assume that the user interface should prevent the user from breaking this rule, it is a far better design if it was dealt with in OneRowNim. To do this we need a Java structure that executes different statements depending on whether the parameter is greater than 3, less than 1, or between 1 and 3. The Java *if-else* statement has this capability. A fuller treatment of control structures appears in Chapter 6, but in this section, we will briefly introduce a couple of simple control structures. This will enable us to write programs that take more interesting actions.

The Simple If Statement

A **selection** control structure, allows a program to select between two or more alternative paths of execution. The if statement is the most basic selection control structure in Java. Most programming languages have its equivalent.

JAVA LANGUAGE RULE

If Statement. The if statement has the following syntax:

if (boolean expression)

containedstatement ;

The statement contained in the if statement can be any valid Java statement, including a compound statement. (Recall from Chapter 1 that a compound statement is a set of statements contained within curly braces.) The boolean expression is an expression that is either true or false. We have seen examples of boolean expressions that involve int variables, int values, and the inequality or equality operators. A method call to a method with a boolean result type is another example of a boolean expression. Given this description of if statement syntax, the following are examples of valid if statements:

```
if (true) System.out.println("Hello");
if (nSticks <= 0) System.out.println("game is over");
```

For readability, we usually write an if statement with its contained statement indented on the next line:

```
if (true)
    System.out.println("Hello");
if (nSticks <= 0)
    System.out.println("game is over");
```

The following are all examples of syntax errors involving the if statement:

```
if true                    // Parentheses are missing
    System.out.println("Hello");

if (nSticks <= 0) return   // Semicolon missing

if ("true") return;  // "true" is not a boolean

if (true) "Hello";    // "Hello" is  not a statement
```

Semantically, the if statement has the following interpretation: First, the boolean condition is evaluated. If it is true, then the contained statement is executed; if it is false, then the contained statement is not executed. This is shown in Figure 3.11. The flowchart clearly shows that program flow will take one or the other of the alternative paths coming out of the diamond-shaped boolean condition box. The branch through the rectangular statement box will be taken when the boolean condition is true; otherwise the statement will be skipped.

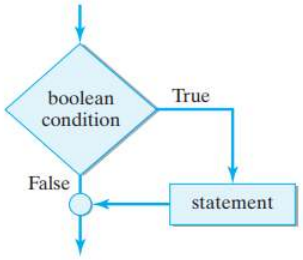


Figure 3.11: Flowchart of the if statement. Diamond-shaped symbols at the branch points contain boolean expressions. Rectangular symbols can only contain executable statements. Circles act simply as connectors, to connect two or more paths.

As another example, consider the definition of a getPlayerString() method for the OneRowNim class:

```
public String getPlayerString()
{
  if (player == 1)
    return "Player One"; // Exit the method
  if (player == 2)
    return "Player Two"; // Exit the method
  return "Player error";   // Exit the method
}
```

The flowchart in Figure 3.12 shows the program flow of the entire getPlayerString() method.

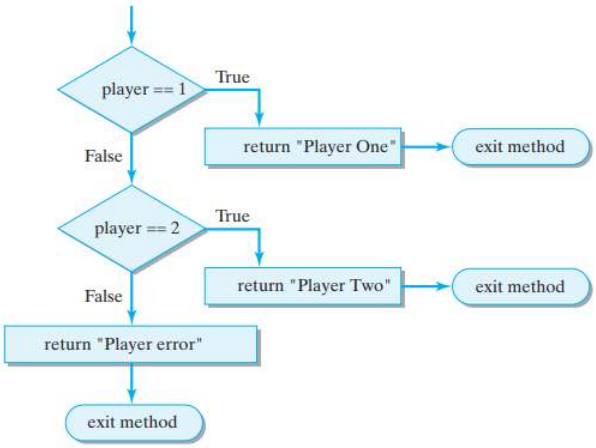


Figure 3.12: Flowchart of the getPlayerString() method.

It is important to note that when a return statement is executed in a method, control is returned immediately to the calling method. Thus, if player == 1 is true, the string “Player One” is returned to the calling method and the getPlayerString() method exits at this point. If it is false, then player == 2 should be true (if we have a consistent state) and the string “Player Two” should be returned and the method exited. Thus, if we have a consistent state —that is, if player has value 1 or 2—then the third return statement should never be reached.

The following example shows the more common case where the statement contained in an if statement can be a compound statement:

```
if (player == 1)
{   String s = "Player One";
    System.out.print (s);
    System.out.println (" plays next");
    System.out.println (" The game is not over");
}
```

If player == 1 is true, then all four statements in the contained compound statement will be executed. Note here that we are declaring the local variable, s, in this block. Its scope would extend only to the end of the block. Note also that when we use a compound statement, the compound statement itself is not followed by a semicolon because it is already enclosed in braces.

A common programming error is to forget the braces around the compound statement. Merely indenting the statements following the if clause doesn’t alter the logic of the if statement. For example, the following if statement still has only one statement in its if clause:

```
if (condition1)
  System.out.println("One");
  System.out.println("Two"); //Not part of if statement
```

This segment will always print “Two” because the second println() is not part of the if statement. To include it in the if statement, you must enclose both println() statements within braces:

```
if (condition1)
{  System.out.println("One");
   System.out.println("Two");
}
```

JAVA DEBUGGING TIP

Indentation. Indentation can improve the readability of a program but doesn't affect its logic. Braces must be used to group statements in the if clause.

The if-else Statement

A second version of the if statement incorporates an else clause into the structure. This allows us to execute either of two separate statements (simple or compound) as the result of one boolean expression. For example, the statement

```
if (player == 1)
    System.out.println("Player One");
else
    System.out.println("Player Two");
```

will print "Player One" if player == 1 is true. Otherwise, it will print "Player Two".

JAVA LANGUAGE RULE

If-else Statement. The if-else statement has the following syntax:

```
if     ( boolean expression )
       statement1 ;
else
       statement2 ;
```

As in the case of the simple if statement, the keyword if is followed by a parenthesized *boolean expression*, which is followed by *statement1*, which may be either simple or compound. If *statement1* is a simple statement, then it is followed by a semicolon. The *else clause* follows immediately after *statement1*. It begins with the keyword else, which is followed by *statement2*, which can also be either a simple or compound statement. Note that there is no boolean expression following the else keyword. In an if-else statement, the boolean expression following the keyword if goes with both the if and else clauses.

Semantically, the if-else statement has the following interpretation: If the *boolean expression* is true, execute *statement1*; otherwise execute *statement2*. This interpretation is shown in Figure 3.13.

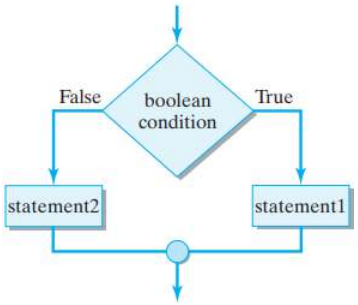


Figure 3.13: Flowchart of the if-else statement.

The Nested if/else Multiway Selection Structure

The statements that one inserts in place of *statement1* and *statement2* in the if-else statement can be any executable statement, including another if statement or if-else statement. In other words, it is possible to embed one or more if-else statements inside another if-else statement, thereby creating a *nested* control structure. As with most things, making a control structure too complex isn't a good idea, but there is a standard nested if-else control structure that is very useful. It is known as **multiway selection**. As shown in Figure 3.14, the multiway structure is used when you want to select one and only one option from several alternatives.

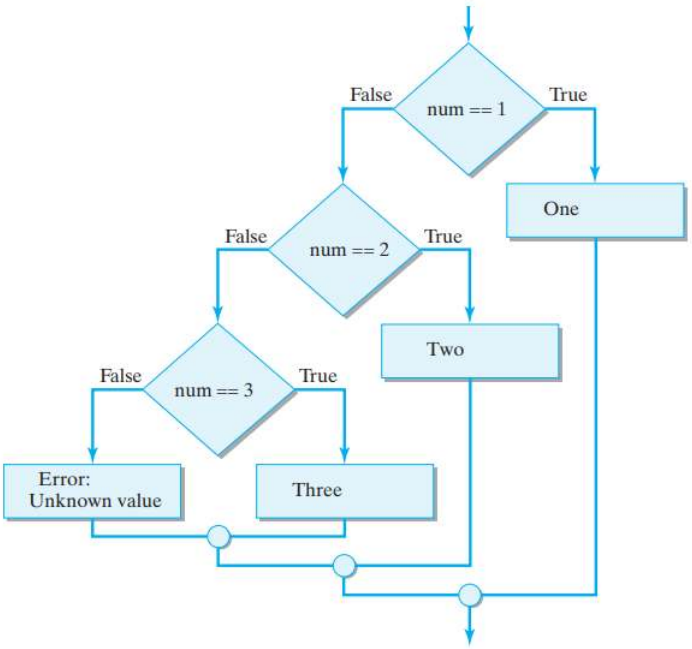


Figure 3.14: Flowchart of a nested if-else statement.

Suppose we have an int variable num that will contain one of the values 1, 2, or 3 unless there has been an error assigning a value to it. Suppose that we want to write code that will write out the English word for the value in num. In the example shown in Figure 3.14 there are three alternatives plus an error state. Here is the Java code for this example:

```
if (num == 1)
   System.out.println("One");
else if (num == 2)
   System.out.println("Two");
else if (num == 3)
   System.out.println("Three");
else
   System.out.println("Error: Unknown value");
```

Note that the multiway structure has a single entry point and that only one of the four possible alternatives is executed. The code will print exactly one of the strings.

We will have many occasions to use the if-else structure. Although it does not represent a significant change, we could rewrite our takeStick() method to make use of the if-else instead of the somewhat obscure statement :

```
player = 3 - player;
```

to change the value of player from 1 to 2 or vice versa:

```
public String takeSticks(int num)
{   nSticks = nSticks - num;
  if (player == 1)
    player = 2;   // From 1 to 2
  else
    player = 1;   // From 2 to 1
}
```

In some respects this version of takeSticks() involves four lines of code instead of one but is simpler to understand. The if-statement tests whether the value of player is 1. If it is, the value is changed to 2. If the value of player is not 1, then the value must be 2 and so the value is changed to 1. Both versions of the code will give precisely the same result, a programmer could choose to write the code either way.

? Exercise 3.6.1

Consider the following method with boolean parameter.

```
public String getStatus(boolean isDone)
{   if (isDone)
        return "Done";
    else
        return "Not Done";
}
```

Draw a flowchart for the if-else version of the method, using the figures in this section as a guide. The if-else structure should be drawn exactly as shown in Figure 3.11. It should have a single entry point that leads directly to the top of a diamond-shaped box that contains a boolean condition. There should be two branches coming out of the condition box. The one going to the right is the true case, and the one going to the left is the false case. Each of these branches should contain one rectangular box, which contains the statements that would be executed in that case. The left and right branches should be connected by a circular symbol that is aligned directly under the diamond box whose conditions it connects. There should be a single exit arrow pointing directly down.

? Exercise 3.6.2

Identify the error in the following statements:

```
if (isHeavy == true)
    System.out.println("Heavy");
else ;
    System.out.println("Light");

if (isLong == true)
    System.out.println("Long")
else
    System.out.println("Short");
```

? Exercise 3.6.3

Suppose we have an int instance variable named player in some class describing a three person game. Write a method named getPlayerName() that returns a String. It should return “Ann”, “Bill”, “Cal”, or “Error” when the value of player is respectively 1, 2, 3, or any other value.

? Exercise 3.6.4

How does a parameter for a primitive type differ from a parameter for a reference type

The While Structure

A **repetition structure** is a control structure that repeats a statement or sequence of statements in a controlled way. Repetition structures are also referred to as **loop structures**. Many types of programming tasks require a repetition structure. Consider some examples.

- You want to add up the squares of the numbers from 1 to 100.
- You want to compute compound interest on an amount of money in a savings account with a fixed interest rate if it is kept there for 30 years.
- A computer security employee wants to try every possible password in order to break into an account of a suspected spy.
- You want to have players input moves for a turn in a game until the game is over. Our OneRowNim is such an example.

We will study several different repetition structures of Java in depth in Chapter 6. We will briefly consider the while statement here so as to be able to define methods that are more powerful and more interesting. Let us write a method that solves a slight generalization of the first problem above. We will use the while statement to sum the squares of integers from 1 to a number specified as a parameter of the method. Thus, the method call sumSquares(3) should return the value 14 since $1 * 1 + 2 * 2 + 3 * 3 = 1 + 4 + 9 = 14$.

```
public int sumSquares(int max)
{ int num = 1;
  int sum = 0;
  while (num <=  max) { // While num <= max
    sum = sum + num*num; // Add square to sum
    num = num + 1;       // Add 1 to num
  } //while
  return sum;           // Return the sum
}
```

Note that in this example, the variable num gets assigned an initial value of 1 before the while statement. Note also that the boolean expression num < max in parentheses after while states the condition for which we wish to continue summing squares. Finally note that the last statement in the block following the boolean expression adds 1 to num–that is, this variable is updated at the end of the block.

The **while statement** is a loop statement in which the loop entry condition occurs before the loop body. It has the following general form:

JAVA LANGUAGE RULE

While Statement. The while statement has the following syntax:

while (loop entry condition)

loopbody ;

When the while statement is executed, the loop entry condition is evaluated and if this evaluates to false, execution continues at the statement immediately after the loop body. If the loop entry condition evaluates to true, the loop body is executed and then the entry condition is evaluated again. The loop body continues to be executed until the loop entry condition evaluates to false.

To have a while statement accomplish a task, the variable or variables in the loop entry condition must be initialized correctly before the while statement and these variables must be correctly updated at the end of the loop body. We can refer to the initializer statement followed by a while statement as a **while structure**. We can restate the above guidelines as a design principle:

JAVA EFFECTIVE DESIGN

Loop Structure. A properly designed while structure must include an initializer, a loop entry condition, and an updater. The updater should guarantee that the loop entry condition eventually fails, thereby allowing the loop to terminate.

In pseudocode, the *while structure* would take the following form:

```
InitializerStatements;          // Initializer
while (loop entry condition) { // Bound test
     Statements;                // Loop body
     UpdaterStatements;         // Updater
}
```

As its form suggests, the while structure is designed so that on some conditions the loop body will never be executed. Because it tests for the loop entry condition before the loop body, it is possible that the loop body is never executed. We might say that it is designed to perform 0 or more iterations.

For example, if the method call sumSquares(-3) is executed, the loop body will be skipped, because the loop entry condition num <= max is false to begin with. No iterations will be performed, and the algorithm will simply return the value 0.

Note also that in the while statement the bound test is preceded by initializer statements, and the loop body contains updater statements. The semantics of the while structure are shown in Figure 3.15.

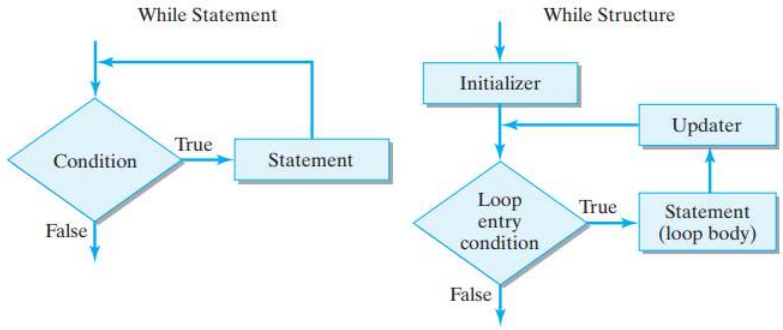


Figure 3.15: Flowchart of the while statement and while structure.

? Exercise 3.6.5

Modify the definition of the sumSquares() method to define a method named sumCubes() that sums the cubes of integers from a minimum value up to a maximum value and returns that sum. sumCubes() should have two parameters that will store the minimum and maximum values. Thus the method call sumCubes(2,3) should return 35 since $2*2*2+3*3*3=8+27=35$.

This page titled 3.6: Flow of Control- Control Structures is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **3.6: Flow of Control- Control Structures** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

3.7: Testing an Improved OneRowNim

Let's use the control structures that we have discussed to improve the definition of the takeSticks() method of OneRowNim. We noted earlier that our current definition allows 4 or more sticks to be removed from nSticks even though the rules of One Row Nim indicate that a player must take one, two, or three sticks on a turn. We can use if-else statements to make certain that no more than 3 sticks get removed.

What should happen if the method takeSticks() is called with an argument that does not represent a legal number of sticks to remove? In this case, it would probably make sense to remove no sticks at all and to keep the value of player the same so that the player whose turn it is does not change. In addition, it would be nice if the method could signal that an illegal move has been attempted. This can be accomplished if we redefine takeSticks() to return a boolean value. Let's have a return value of true represent the case that a valid number of sticks have been removed and the player to play next has been changed. A return of false will indicate that an illegal move has been attempted. Making these changes to the takeSticks() method will yield a method definition that looks like:

```
public boolean takeSticks(int num)
{   if (num < 1) {
       return false; // Error
    } else if ( num > 3) {
       return false; // Error
    } else {
       nSticks = nSticks - num;
       player = 3 - player;
       return true;
    } //else
} //takeSticks
```

Notice that the new definition of the takeSticks() method has a boolean return type. Also notice that the if/else multiway structure is used to handle the three cases of the parameter num being less than one, more than three, or a valid number.

Let us add one more method to the OneRowNim class. Let's define a method called getWinner()that will return the number of the winning player if the game is over. Recall that the player who takes the last stick loses, so after that last play, the player whose turn it is to play next is the winner. However, we should be concerned about what value to return if the game is not over when the method is called. A common strategy is to have a method return a special value to indicate that it is in a state in which it cannot return the value requested. Returning a 0 value is a good way to indicate that the game is not over so a winner cannot be identified. With this information, the if/else statement can be used in the definition of getWinner().

```
public int getWinner()
{   if (nSticks < 1)
        return player;
    else
        return 0;
} // getWinner()
```

We now have the final version (for this chapter) of the OneRowNim class whose implementation is given in Figure 3.16. We have turned a very simple class into one that contains quite a few elements. Compared to our first version (in Chapter 1), this Chapter's version of OneRowNim presents an interface (to other objects) that is easy and convenient to use. The constructor methods with parameters provide an easy way to create a OneRowNim instance with any number of sticks. The use of private instance variables and a single, carefully designed mutator method, takeSticks(), prevents other objects from tampering with the state of a OneRowNim object's state. The other methods provide a flexible way to find out the state of a OneRowNim object. The complete implementation of this OneRowNim is shown in Figure 3.16.

```
public class OneRowNim
{   private int nSticks = 7;
    private int player = 1;

    public OneRowNim()
    {
    } // OneRowNim() constructor

    public OneRowNim(int sticks)
    {   nSticks = sticks;
    }  // OneRowNim() constructor2

    public OneRowNim(int sticks, int starter)
    {   nSticks = sticks;
        player = starter;
    }  // OneRowNim() constructor3

    public boolean takeSticks(int num)
    {   if (num < 1) return false;       // Error
        else if ( num > 3) return false; // Error
        else              // this is a valid move
        {   nSticks = nSticks - num;
            player = 3 - player;
            return true;
        } // else
    } // takeSticks()

    public int getSticks()
    {   return nSticks;
    } // getSticks()

    public int getPlayer()
    {   return player;
    } // getPlayer()

    public boolean gameOver()
    {   return (nSticks <= 0);
    } // gameOver()

    public int getWinner()
    {   if (nSticks < 1) return getPlayer();
        else return 0;  //game is not over
    } // getWinner()

    public void report()
    {   System.out.println("Number of sticks left: " +
                                            getSticks());
        System.out.println("Next turn by player " +
                                            getPlayer());
    }   // report()
} // OneRowNim class
```

Figure 3.16: The OneRowNim class with improved methods.

Let's use a while statement to test the new methods of the class. A pseudocode description of how a game is played might look like:

```
  Choose the initial number of sticks for the game
  while the game is not over
  {   Report the state of the game
      Process the next move
  }
  Report the state of the game
  Report who the winner is
```

Translating this pseudocode into Java code in a main() method in a separate class gives us the class shown in Figure 3.17. We will use the Scanner class introduced in the previous chapter to get moves from the keyboard for both players. Before each move game.report() describes the state of the game before the user is prompted to input a move for one of the players. A reader interested in seeing the lengthy output to the console when the TestOneRowNim class is run is encouraged to actually run the program.

```
import java.util.Scanner;

public class TestOneRowNim
{
  public static void main(String argv[])
  { Scanner sc = Scanner.create(System.in);
    OneRowNim game = new OneRowNim(11);
    while(game.gameOver() == false)
    {   game.report();  // Prompt the user
      System.out.print("Input 1, 2, or 3: ");
      int sticks = sc.nextInt(); // Get move
      game.takeSticks(sticks);   // Do move
      System.out.println();
    } // while
    game.report();  // The game is now over
    System.out.print("Game won by player ");
    System.out.println(game.getWinner());
 } // main()
} // TestOneRowNim
```

Figure 3.17: The TestOneRowNim class with a while loop.

Note that the return value of the takeSticks() method is ignored in this test program. We will make use of the return value in test programs in the next chapter when better user interfaces are developed for OneRowNim. Note, however, that taken together, the public methods for OneRowNim provide other objects with an interface that they can use to communicate with individual OneRowNim objects.

JAVA EFFECTIVE DESIGN

Interfaces. Well-designed objects provide a useful public interface and protect the object's private elements from other objects.

To reiterate a point made at the outset, object-oriented programming is a process of constructing objects that will interact with each other. Object-oriented programs must ensure that the objects themselves are well designed in terms of their ability to carry out their designated functions. Good design in this sense requires careful selection of instance variables and careful design of methods to ensure that the object can carry out its assigned tasks. However, equal care must be taken to ensure that the interactions that take

place among objects are constrained in ways that make sense for that particular program. This aspect of designing objects comes into play in designing the methods—constructor, accessor, and mutator—that make up the object's interface.

Special Topic: Intelligent Agents

Wouldn't it be nice if we had a computer program that could schedule appointments for us, remind us of meetings and commitments, find information for us on the WWW, and manage our e-mail messages for us? Wouldn't it be nice to have a computerized personal assistant?

Actually, such programs are called *intelligent agents*, which are programs that are capable of acting autonomously to carry out certain tasks. Intelligent agent technology is becoming an important research area in computer science. Most agent programs incorporate some kind of machine learning capability, so that their performance improves over time.

As a typical agent activity, suppose I was able to tell my intelligent agent to buy me a copy of a certain book that I just heard about. Given a command like "buy me a copy of X," the agent would perform a search of online book sellers and come up with the best deal. Once it had found the best buy, the agent would communicate with a computer-based agent representing the book seller. My agent would make the order and pay for it (assuming I gave it authority to do so), and the book seller's agent would process the order.

As far-fetched as the capability may now seem, this is the direction that research in this area is headed. Researchers are developing agent languages and describing protocols that agents can use to exchange information in a reliable and trustworthy environment. Obviously, you wouldn't want your agent to give your money to a fraudulent book seller, so there are significant problems to solve in this area that go well beyond the problem of simply exchanging information between two agents.

The best way to learn more about this research area is to do a Web search using the search string "Intelligent Agent." There are numerous research groups and companies that provide online descriptions and demos of their products.

This page titled 3.7: Testing an Improved OneRowNim is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **3.7: Testing an Improved OneRowNim** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

3.8: From the Java Library- java.lang.Object

The most general class in Java's class hierarchy is the java.lang.Object class. It is the superclass of all classes that occur in Java programs. By default, it is the direct superclass of any class that does not explicitly specify a pedigree in its class definition.

All subclasses of Object **inherit** the public and protected methods contained in Object, so all such methods can be thought of as belonging to the subclasses. This means that all classes inherit the methods of the Object class, because every class is a subclass of it. In this section, let's look briefly at how we can use an inherited method and also at how we can **override** it–that is, redefine the method–if it doesn't exactly suit our purposes.

One of the most useful methods in the Object class is the toString() method:

```
public class Object
{
    public String toString() ;
}
```

The toString() method returns a String representation of its object. For example, o1.toString() will return a String that in some sense describes o1.

Because OneRowNim is a subclass of Object, it inherits the toString() method. To illustrate the default behavior of toString(), let's use it with a OneRowNim instance:

```
OneRowNim g1 = new OneRowNim(11);
OneRowNim g2 = new OneRowNim(13);
System.out.println(g1.toString());
System.out.println(g2.toString());
```

This code segment creates two OneRowNim instances, one named g1 and the other named g2. The inherited toString() method is then invoked on each OneRowNim instance, which produces the following output:

```
OneRowNim@1dc6077b
OneRowNim@1dc60776
```

What this experiment shows is that the default definition of toString() returns some kind of internal representation of its object. It looks as if it returns the name of the object's class concatenated with its memory address. This may be useful for some applications. But for most objects we will want to override the default definition to make the toString() method return a string that is more appropriate for OneRowNim.

What String should the g1.toString() method return? Let's have it return a String that reports the OneRowNim instances's current state, which are the values stored in the two instance variables. To override a method, you simply define a method with the same signature in the subclass. If you call toString() with an instance of the subclass, its version of the method will be used. In this way, the subclass method overrides the superclass version. Thus, OneRowNim.toString() will have the following signature:

```
public String toString();
```

Let us describe the state of a oneRowNim instance very briefly in the string returned by the toString() method:

```
public String toString()
{ return "nSticks = " + nSticks + ", player = " + player;
}
```

If we add the toString() method to the OneRowNim class and then run the program shown in Figure 3.18, we get the following output:

```
  nSticks = 9, player = 2
  nSticks = 13, player = 1
```

```
public class TestToString
{
  public static void main(String argv[])
  { OneRowNim g1 = new OneRowNim(11);
    OneRowNim g2 = new OneRowNim(13);
    g1.takeSticks(2);
    System.out.println(g1.toString());
    System.out.println(g2.toString());
  } //main
} //TestToString
```

Figure 3.18: An application to test the overridden toString() method.

While this new method may not play an important role in the OneRowNim class, it does provide a very brief, understandable description of the state of the object. This is the reason that the toString() method was included in the Object class.

This page titled 3.8: From the Java Library- java.lang.Object is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **3.8: From the Java Library- java.lang.Object** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

3.9: Object-Oriented Design- Inheritance and Polymorphism

This use of Object's toString() method provides our first look at Java's inheritance mechanism and how it promotes the generality and extensibility of the object-oriented approach. As a subclass of Object, our OneRowNim class automatically inherits toString() and any other public or protected methods defined in Object. We can simply use these methods as is, insofar as they are useful to us. As we saw in this case, the default version of toString() wasn't very useful. In that case, we can override the method by defining a method in our class with the exact same method signature. The new version of toString() can be customized to do exactly what is most appropriate for the subclass.

One of the great benefits of the object-oriented approach is the ability to define a task, such as toString(), at a very high level in the class hierarchy and let the inheritance mechanism spread that task throughout the rest of the hierarchy. Because toString() is defined in Object, you can invoke this method for any Java object. Moreover, if you override toString() in the classes you define, you will be contributing to its usefulness. Two important lessons from this example are

> **JAVA EFFECTIVE DESIGN**
>
> Inheritance. The higher up in the class hierarchy that a method is defined, the more widespread its use can be.

> **JAVA EFFECTIVE DESIGN**
>
> Overriding toString(). The toString() method can be overridden in any user defined Java class. It is a useful thing to do in any class where the state of an object can be defined briefly.

Obviously there is much more that needs to be explained about Java's inheritance mechanism. Therefore, we will be revisiting this topic on numerous occasions in subsequent chapters.

Another important concept of object-oriented design is polymorphism. The toString() method is an example of a polymorphic method. The term **polymorphism** is from the Greek terms *poly*, which means "many," and *morph*, which means "form." The toString() method is polymorphic because it has different behavior when invoked on different objects.

For example, suppose we design a class, Student, as a subclass of Object and define its toString() method to return the student ID number. Given this design, then obj.toString() will return a student ID if obj is an instance of Student, but if it is an instance of OneRowNim, it will return a the description of its state that we defined above. The following code segment illustrates this point:

```
Object obj;           // obj can refer to any Object
obj = new Student("12345");// obj refers to a Student
System.out.println(obj.toString()); // Prints "12345"
obj = new OneRowNim(11); // obj refers to a OneRowNim
System.out.println(obj.toString());

// Prints: nSticks = 11, player = 1
```

In this case, the variable obj is used to refer to a Student and then to a OneRowNim instance. This is okay because both classes are subclasses of Object. When toString() is invoked on obj, Java will figure out what subclass of Object the instance belongs to and invoke the appropriate toString() method.

This page titled 3.9: Object-Oriented Design- Inheritance and Polymorphism is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **3.9: Object-Oriented Design- Inheritance and Polymorphism** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

3.10: Drawing Lines and Defining Graphical Methods (Optional)

We used a Graphics object in the previous chapter to draw rectangles and ovals in a JFrame window. The Graphics class also possesses a method for drawing a line segment. Problems involving drawing pictures in an JFrame window using a series of line segments can be a source of examples of defining useful methods and also of making good use of loops.

The Graphics class has a public instance method with the header:

```
public  void  drawLine(int x1, int y1, int x2, int y2)
```

The method call g.drawLine(x1, y1, x2, y2) draws a line from the point $(x1, y1)$ to $(x2, y2)$ where (x, y) refers to a point that is x pixels from the left edge of the area that g is drawing in and y pixels from the top edge. Thus g.drawLine(10, 10, 10, 60) draws a vertical line segment that is 50 pixels long and is 10 pixels from the left edge of the drawing area, that is, a line segment from the point $(10, 10)$ to the point $(10, 60)$.

Consider the problem of creating an Swing program with a method called drawSticks() to draw any specified number of vertical line segments. This method might be useful for an graphical user interface to the OneRowNim game to draw the number of sticks at a given point in a game. Suppose that this method must have an int parameter to specify the number of vertical lines to draw and two int parameters to specify the location of the top endpoint of the left most line segment. The drawSticks() method will need to use a Graphics object connected to the JFrame window for drawing the line segment. The only such Graphics object available is the parameter in the paint() method of the Canvas. Thus the method must have a Graphics parameter and it will be called in the paint() method using the Graphics object there as an argument. Thus the header of the method should look like:

```
public void drawSticks(Graphics g,int x,int y,int num)
```

The length of the line segments and and the distance between them are not specified by parameters so we need to choose some fixed values for these quantities. Let us assume that the line segments are 10 pixels apart and 50 pixels long. We now have enough information to complete the definition of an applet to solve this problem. Such a class definition is reproduced in Figure 3.19.

```
/** DrawLineCanvas   demonstrates some graphics commands.
 *It draws a set of 12 vertical lines and a set of 7 lines.
 */
import java.awt.*;
import javax.swing.JFrame;

public class DrawSticksCanvas extends Canvas
  /** drawSticks(g,x,y,num) will draw num vertical line
   * segments.  The line segments are 10 pixels apart and
   * 50 pixels long. The top endpoint of the left most
   *line segment is at the point (x,y).
   */
  public void drawSticks(Graphics g, int x, int y, int num)
  {   int k = 0;
      while (k < num)
      {   g.drawLine(x, y, x, y + 50);
          x = x + 10;
          k = k + 1;
      } // while
  } // drawSticks()

  public void paint(Graphics g)
  {   drawSticks(g, 25, 25, 12);
      g.setColor(Color.cyan);
      drawSticks(g, 25, 125, 7);
  } // paint()


} // DrawSticksCanvas
```

Figure 3.19: A Swing Class with a method for drawing a set of sticks.

Note that the body of drawSticks() uses a while-loop to draw the lines and declares and initializes a local variable to zero to use for counting the number of lines drawn. The statement g.drawLine(x, y, x, y + 50); draws a vertical line which is 50 pixels long. Increasing the value of x by 10 each time through the loop moves the next line 10 pixels to the right.

The first call to drawSticks() in the paint() method draws 12 lines with $(25, 25)$ the top point of the left-most line. The second call to drawSticks() will draw 7 cyan sticks 100 pixels lower. Note that changing the color of g before passing it as an argument to drawSticks() changes the drawing color.

An image of the DrawSticksCanvas as it appears in a window is shown in Figure 3.20.

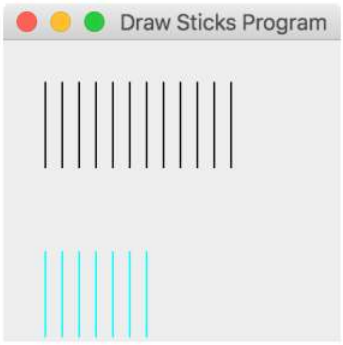


Figure 3.20: The DrawSticksCanvas as displayed in a Java window.

As we have seen in this example, defining methods with parameters to draw an object makes the code reusable and makes it possible to draw a complex scene by calling a collection of simpler methods. It is a typical use of the divide-and-conquer principle. The while-loop can be useful in drawing almost any geometrically symmetric object.

Chapter Summary

accessor method

class scope

formal parameter

if statement

if/else statement

inherit

local scope

loop structure

method overloading

method signature

mutator method

multiway selection

override

polymorphism

repetition structure

scope

selection

side effect

while statement

while structure

A *formal parameter* is a variable in a method declaration. It always consists of a type followed by a variable identifier. An *argument* is a value that is passed to a method via a formal parameter when the method is invoked. A method's *parameters* constrain the type of information that can be passed to a method.

When an argument of primitive type is passed to a method, it cannot be modified within the method. When an argument of reference type is passed to a method, the object it refers to can be modified within the method.

Except for void methods, a *method invocation* or *method call* is an expression which has a value of a certain type. For example, nim.getSticks() returns a int value.

The *signature* of a method consists of its name, and the number, types, and order of its formal parameters. A class may not contain more than one method with the same signature.

A *constructor* is a method that is invoked when an object is created. If a class does not contain a constructor method, the Java compiler supplies a *default constructor*.

Restricting access to certain portions of a class is a form of *information hiding*. Generally, instance variables are hidden by declaring them private. The class's public methods make up its interface.

The *if statement* executes a statement only if its boolean condition is true. The *if-else statement* executes one or the other of its statements depending on the value of its boolean condition. *Multiway selection* allows one and only one of several choices to be selected depending on the value of its boolean condition.

The while statement is used for coding loop structures that repeatedly execute a block of code while a boolean condition is satisfied.

A *method declaration* defines the method by specifying its name, qualifiers, return type, formal parameters, and its algorithm, thereby associating a name with a segment of executable code. A *method invocation* calls or uses a defined method.

A *formal parameter* is a variable in the method declaration, whose purpose is to store a value while the method is running. An *argument* is a value that is passed to a method in place of a formal parameter.

The following code declares two instance variables for names of players and defines a setName() method:

```
private String nameOne = "Player One";
private String nameTwo = "Player Two";

public void setNames(String name1, String name2)
{    nameOne = name1;
     nameTwo = name2;
}
```

Of course, there are many other appropriate names for the variables and parameters and other initial assignments.

A method call that sets the names of the players of game1 is:

```
game1.setNames("Xena","Yogi");
```

A constructor cannot have a return type, such as void.

One definition for the method is:

```
public OneRowNim(int sticks)
{    nSticks = sticks;
     player = 2;
}
```

The following would be displayed on the screen:

```
   1
   20
   false
```

One definition for the method is:

```
public int getMoves()
{   return nMoves;
}
```

One definition for the method is:

```
public boolean playerOneIsNext()
{   return (player == 1);
}
```

See Figure 3.21.

```
if (isHeavy == true)
     System.out.println("Heavy") ;
else ;  // Error (remove this semicolon)
     System.out.println("Light");

if (isLong == true)
     System.out.println("Long")
else   // Error (end line above with semicolon)
     System.out.println("Short");
```

```
public String getPlayerName()
{    if (player == 1)
         return "Ann";
     else if (player == 2)
         return "Bill";
     else if (player == 3)
         return "Cal";
     else
         return "Error";
}
```

When passing an argument for a primitive type, a copy of the argument's value is passed. The actual argument cannot be changed inside the method. When passing a reference to an object, the object can be changed within the method.

```
public int sumCubes(int min, int max)
{
    int num = min;
    int sum = 0;
    while (num <=  max) { // While num <= max
        sum = sum + num*num*num; // Add cube of num to sum
        num = num + 1;       // Add 1 to num
    } //while
    return sum;           // Return the sum
}
```

Fill in the blanks in each of the following sentences:

=14pt

When two different methods have the same name, this is an example of

_____.

Methods with the same name are distinguished by their

_____.

A method that is invoked when an object is created is known as a

_____method.

A method whose purpose is to provide access to an object's instance variables is known as an

_____method.

A boolean value is an example of a

type.

A OneRowNim variable is an example of a

type.

A method’s parameters have

scope.

A class’s instance variables have

scope.

Generally, a class’s instance variables should have

access.

The methods that make up an object’s interface should have

access.

A method that returns no value should be declared

.

Java’s if statement and if-else statement are both examples of

control structures.

An expression that evaluates to either true or false is known as a

.

In an if-else statement, an else clause matches

.

The ability to use a superclass method in a subclass is due to Java’s

mechanism.

The process of redefining a superclass method in a subclass is known as

the method.

=11pt

Explain the difference between the following pairs of concepts:

Parameter and *argument*.

Method definition and *method invocation*.

Local scope and *class scope*.

Primitive type and *reference type*.

Access method and *constructor method*.

Translate each of the following into Java code:

If b1 is true, then print “one”; otherwise, print “two”.

If b1 is false and if b2 is true, then print “one”; otherwise, print “two”.

If b1 is false and if b2 is true, then print “one”; otherwise, print “two”, or print “three”.

Identify and fix the syntax errors in each of the following:

```
if (isWalking == true) ;
    System.out.println("Walking");
else
    System.out.println("Not walking");
```

```
if (isWalking)
     System.out.println("Walking")
else
     System.out.println("Not walking");
```

```
if (isWalking)
     System.out.println("Walking");
else
     System.out.println("Not walking")
```

```
if (isWalking = false)
     System.out.println("Walking");
else
     System.out.println("Not walking");
```

For each of the following, suppose that isWalking is true and isTalking is false (first draw a flowchart for each statement and then determine what would be printed by each statement):

```
if (isWalking == false)
     System.out.println("One");
     System.out.println("Two");
```

```
if (isWalking == true)
     System.out.println("One");
     System.out.println("Two");
```

```
if (isWalking == false)
{
     System.out.println("One");
     System.out.println("Two");
}
```

```
if (isWalking == false)
     if (isTalking == true)
         System.out.println("One");
     else
         System.out.println("Two");
else
     System.out.println("Three");
```

Show what the output would be if the following version of main() were executed:

```
public static void main(String argv[])
{
     System.out.println("main() is starting");
     OneRowNim game1;
     game1  = new OneRowNim(21);
     OneRowNim game2;
     game2 = new OneRowNim(8);
     game1.takeSticks(3);
     game2.takeSticks(2);
     game1.takeSticks(1);
     game1.report();
     game2.report();
     System.out.println("main() is finished");
}
```

Determine the output of the following program:

```
public class Mystery
{
     public String myMethod(String s)
     {
         return("Hello" + s);
     }
     public static void main(String argv[])
     {
         Mystery mystery = new Mystery();
         System.out.println( mystery.myMethod(" dolly");
     }
}
```

Write a boolean method—a method that returns a boolean—that takes an int parameter and converts the integers 0 and 1 into false and true, respectively.

Define an int method that takes a boolean parameter. If the parameter's value is false, the method should return 0; otherwise, it should return 1.

Define a void method named hello that takes a single boolean parameter. The method should print "Hello" if its parameter is true; otherwise, it should print "Goodbye".

Define a method named hello that takes a single boolean parameter. The method should return "Hello" if its parameter is true; otherwise it should return "Goodbye". Note the difference between this method and the one in the previous exercise. This one returns a String. That one was a void method.

Write a method named hello that takes a single String parameter. The method should return a String that consists of the word "Hello" concatenated with the value of its parameter. For example, if you call this method with the expression hello("dolly"), it should return "hello dolly". If you call it with hello("young lovers wherever you are"), it should return "hello young lovers wherever you are".

Define a void method named day1 that prints "a partridge in a pear tree".

Write a Java application program called TwelveDays that prints the Christmas carol "Twelve Days of Christmas." For this version, write a void method named intro() that takes a single String parameter that gives the day of the verse and prints the intro to the song. For example, intro("first") should print, "On the first day of Christmas my true love gave to me". Then write methods day1(), day2(), and so on, each of which prints its version of the verse. Then write a main() method that calls the other methods to print the whole song.

Define a void method named verse that takes two String parameters and returns a verse of the Christmas carol “Twelve Days of Christmas.” For example, if you call this method with verse("first", "apartridge in a pear tree"), it should return, “On the first day of Christmas my true love gave to me, a partridge in a pear tree”.

Define a void method named permute, which takes three String parameters and prints out all possible arrangements of the three strings. For example, if you called permute("a", "b", "c"), it would produce the following output: abc, acb, bac, bca, cab, cba, with each permutation on a separate line.

Design a method that can produce limericks given a bunch of rhyming words. That is, create a limerick template that will take any five words or phrases and produce a limerick. For example, if you call

```
limerick("Jones","stones","rained","pained","bones");
```

your method might print (something better than)

```
There once a person named Jones
Who had a great liking for stones,
But whenever it rained,
Jones' expression was pained,
Because stones weren't good for the bones.
```

For each of the following exercises, write a complete Java application program:

Define a class named Donor that has two instance variables, the donor’s name and rating, both of which are Strings. The name can be any string, but the rating should be one of the following values: “high,” “medium,” or “none.” Write the following methods for this class: a constructor, Donor(String,String), that allows you to set both the donor’s name and rating; and access methods to set and get both the name and rating of a donor.

Challenge. Define a CopyMonitor class that solves the following problem. A company needs a monitor program to keep track of when a particular copy machine needs service. The device has two important (boolean) variables: its toner level (too low or not) and whether it has printed more than 100,000 pages since its last servicing (it either has or has not). The servicing rule that the company uses is that service is needed when either 100,000 pages have been printed or the toner is too low. Your program should contain a method that reports either “service needed” or “service not needed” based on the machine’s state. (Pretend that the machine has other methods that keep track of toner level and page count.)

Challenge. Design and write an OldMacdonald class that sings several verses of “Old MacDonald Had a Farm.” Use methods to generalize the verses. For example, write a method named eieio() to “sing” the part of the verse. Write another method with the signature hadAnX(String s), which sings the “had a duck” part of the verse, and a method withA(String sound) to sing the “with a quack quack here” part of the verse. Test your class by writing a main() method.

Suppose you have an Object A, with public methods a(), b(), and private method c(). And suppose you have a subclass of A named B with methods named b(), c() and d(). Draw a UML diagram showing the relationship between these two classes. Explain the inheritance relationships between them and identify those methods that would be considered polymorphic.

Consider the definition of the class C. Define a subclass of C named B that overrides method m1() so that it returns the difference between m and n instead of their sum.

```
public class C {
    private int m;
    private int n;
    public C(int mIn, int nIn) {
        m = mIn;
        n = nIn;
    }
    public int m1() {
        return m+n;
    }
}
```

This page titled 3.10: Drawing Lines and Defining Graphical Methods (Optional) is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **3.10: Drawing Lines and Defining Graphical Methods (Optional)** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

CHAPTER OVERVIEW

4: Input/Output- Designing the User Interface

Learning Objectives

After studying this chapter, you will

- Understand the importance of the user interface.
- Know how to use a simple command-line interface.
- Be able to program and use a simple Graphical User Interface (GUI).
- Understand the concept of event-driven programming.
- Know how to program and use a Java Swing program.

4.1: Introduction
4.2: The User Interface
4.3: A Command-line Interface
4.4: A Graphical User Interface (GUI)
4.5: Case Study- The One Row Nim Game
4.6: From the Java Library- java.io.File and file input (Optional)

This page titled 4: Input/Output- Designing the User Interface is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

4.1: Introduction

One of the most important parts of learning a programming language is learning how to program an application to accept input and produce outputs (I/O). Computers wouldn't be very useful if we could not give them data to manipulate and compute, and if we were not able to read or understand the results that they produce. In general, a computer program's input and output capabilities are known collectively as its user interface.

An **input operation** is any action that transfers data from the user to the computer's main memory via one of the computer's input devices. An **output operation** is any action that transfers data from the computer's main memory to one of the computer's output devices.

In this chapter, we will introduce three simple user interfaces: a **command-line interface** and two **graphical user interfaces** (GUIs). These interfaces can be used interchangeably with the material in most of the subsequent chapters. Indeed, one of the most important design principles that we emphasize in this chapter is that the user interface should be designed to function independently of the computational task. In other words, it should be possible to take an application, such as a computer game, and design it so that it can be used with a variety of different user interfaces.

4.1: Introduction is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

- **4: Input/Output- Designing the User Interface** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

4.2: The User Interface

The user interface is that part of the program that handles the input and output interactions between the user and the program. As an interface, it limits or constrains the manner in which the user can interact with the program.

Computer programs are just one of the many things that require a user interface. Virtually every device we use has one. For example, consider again the difference between the user interface of a digital versus an analog watch. On a digital watch, you have a display that tells you the time to the exact hour, minute, and second. On an analog watch, one with a sweep second hand, the time can never be displayed to the exact second. Similarly, on a digital watch there are buttons that let you set the time to the exact hour, minute, and second. On an analog watch, there is a small wheel that allows you to set the time only approximately. Thus, the user interface constrains the kinds of interactions that are possible between the user and the device.

With regard to our Java programs, one way to divide up the labor is to distinguish between the user interface and the computational functions. The role of the user interface is to transmit data back and forth between the user and the program. The role of the computational part of the program is to perform some kind of computation, in the broad sense of that term. The computation might be to play a game, or calculate a square root, or monitor a hospital patient. Figure 4.1 provides a generic picture of the relationship between the user interface and the computational object.

In this chapter we focus our attention on the user interface side of the relationship shown in Figure 4.1. In subsequent chapters we will focus more on the computational side of the relationship. What we desire is an approach that lets us combine a computational object with any one of the three different kinds of user interfaces.

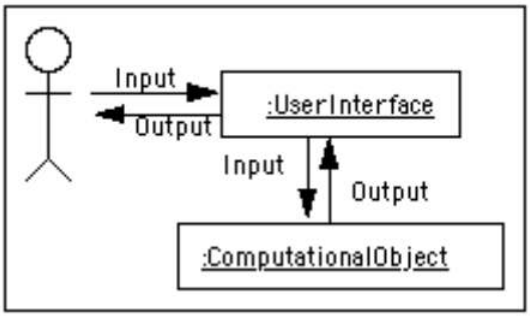


Figure 4.1: The user interface transmits data back and forth between the user and the program's computational objects.

JAVA EFFECTIVE DESIGN

The User Interface Module Separating the user interface from the computational object is a good way to divide up the labor in programs that perform I/O.

This page titled 4.2: The User Interface is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **4.2: The User Interface** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

4.3: A Command-line Interface

A command-line interface is perhaps the simplest, and most old-fashioned, way to design the interaction between a user and a program. According to this approach, user input is taken from the keyboard, and the program's output is displayed on some kind of console (Fig. 4.2).

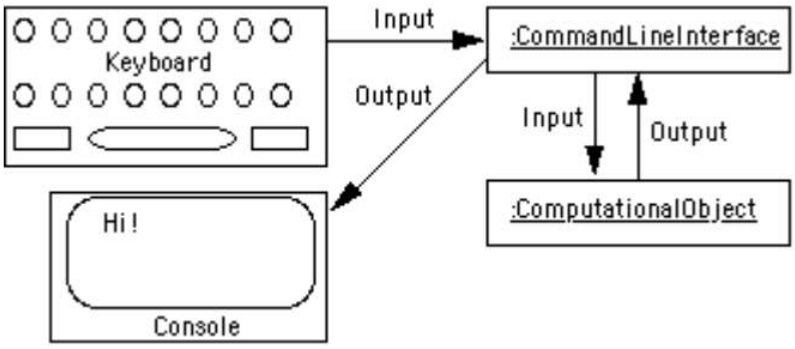


Figure 4.2: A command-line user interface.

The command-line approach might also be called *console interface*. In the early days of computers, before we had graphics-based computer monitors capable of displaying multiple windows, the console was the entire computer display. For today's computers the console might be a window provided by your programming environment, as in Figure 4.3.



Figure 4.3: The Java console window.

In Chapter 3 we described how to use the System.out.print() and System.out.println() methods to output strings to the console. That takes care of the output side of command-line interface. The more challenging task is managing the input-side of the interface.

In Java, input and output is handled by objects that are called **streams**. You can think of a stream as a kind of pipe through which data flow (Fig. 4.4). An **input stream** carries data from some kind of input device, such as a keyboard or network connection or a file, to the program's main memory. An **output stream** carries data from the program's memory to some kind of output device, such as a printer or a file.

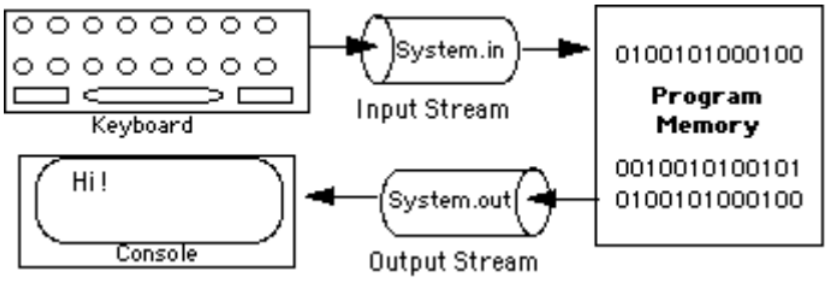


Figure 4.4: Input and output streams.

Each Java program has three standard streams available to it at startup: System.in, System.out, and System.err. System.in is a predefined input stream that is typically associated with the keyboard (Fig. 4.4). That is, it carries data from the keyboard to the

program. System.out and System.err are both output streams typically associated with the console. They both carry data from the program to the console. The difference is simply that System.out is used for normal program output and System.err is used to output error messages.

Using a BufferedReader to Input Strings from the Keyboard

We will use a BufferedReader object to handle data input from the keyboard. As its name implies, the BufferedReader class performs buffered input. A **buffer** is a portion of main memory where input is held until it is needed by the program. Using a buffer between the keyboard and the program allows you to use the Backspace key to delete a character. When you hit the Enter key, any characters that you deleted will be ignored when the program retrieves characters from the input buffer. If the user's input were not buffered in this way, it would contain every keystroke, including the Backspaces, and then it would be up to the program to eliminate the characters that were supposed to be deleted.

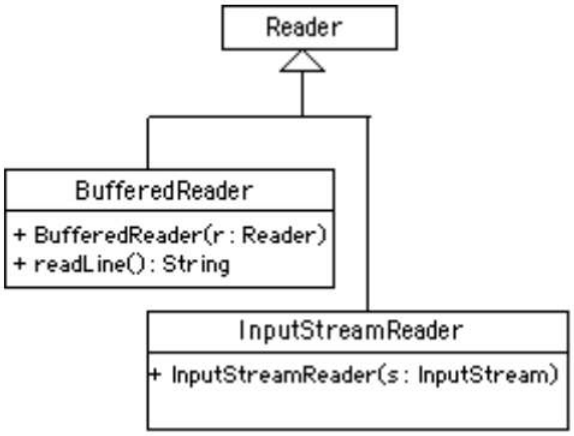


Figure 4.5: The BufferedReader class.

Figure 4.5 provides a UML diagram of the BufferedReader class and shows its relationship to other the classes that will be used for keyboard input . Note that along with InputStreamReader, BufferedReader is one of several subclasses of the Reader class. As the diagram shows, BufferedReader has two important methods. Its constructor method takes a Reader parameter, which means that when we create a BufferedReader we must provide it with a reference to some kind of Reader object. To perform keyboard input, we want to provide a reference to an object that can read System.in, the standard input stream. As the figure shows, InputStreamReader has a constructor that allows it to read an InputStream. Therefore, to construct a BufferedReader that will read System.in we use the following statement:

```
BufferedReader input = new BufferedReader
   (new InputStreamReader (System.in));
```

In this statement we are actually creating two objects. We first create an InputStreamReader, giving it a reference to System.in. We then pass that object to a BufferedReader. The result is a cooperation between two objects that enables us to do buffered reading of the keyboard.

By creating a BufferedReader in this way, whenever we use its readLine() method, it will read a line of characters from the keyboard. For example, having created a BufferedReader named input, the following code segment will read one line of input and assign it to the String variable named inputString.

```
String inputString = input.readLine();
```

When the program encounters the readLine() expression, it will wait for the user to hit the Enter key. It will then input whatever the user typed, minus any characters that were Backspaced over, into the String variable.

JAVA LANGUAGE RULE

Keyboard Input. The BufferedReader.readLine() method allows the user to backspace over errors during keyboard input.

Inputting Numbers from the Keyboard

As the previous section showed, we can use a BufferedReader object to input Strings from the keyboard. In Java, all keyboard input is represented as Strings. However, what if we want to input numbers? The answer is that we have to extract numbers from the input strings. To do this, Java provides us two special classes, known as wrapper classes: Integer and Double.

A **wrapper class** contains methods for converting primitive data into objects and for converting data from one type to another. The Integer class contains the parseInt() method, which extracts an int from its String argument. For example, in the following usage, the string "55" is converted into the number 55:

```
int m = Integer.parseInt("55");
```

Similarly, the Double class contains the parseDouble() method, which extracts a double value from its parameter. In this example, the number 55.2 is extracted from the string "55.2":

```
double num = Double.parseDouble("55.2");
```

If we are writing a program that requires us to input numbers from the keyboard, then assuming we have created a BufferedReader object named input, we can use these methods in combination with the readLine() method, to input and process numbers. For example, this code segment calculates a runner's race pace:

```
String inputString = new String();
System.out.println("How many total miles did you run? ");
inputString = input.readLine();   // Input a String}
double miles = Double.parseDouble(inputString); // Convert
System.out.println("How many minutes did it take you? ");
inputString = input.readLine();   // Input another String
double minutes = Double.parseDouble(inString);  // Convert
System.out.println("Your average pace was " +
                   minutes/miles + " minutes per mile");
```

Notice how we included prompts in this example so that the user knows what type of input is expected. Designing appropriate prompts is an important aspect of designing a good user interface.

JAVA EFFECTIVE DESIGN

Prompting. In a well-designed user interface, prompts should be used to guide the user through the input process.

Designing a Keyboard Reader Class

Now that we have introduced the library classes and methods that we will use for command-line input, lets design a class to encapsulate these functions. We want a class that will use a BufferedReader to read any kind of data—strings, integers, or real numbers—from keyboard. We also want this class to hide some of the messy details involved in performing keyboard input.

Figure 4.6 presents the design of KeyboardReader class. Note that instances of this class will use a BufferedReader object to perform the actual keyboard input. That's why we need a private instance variable of type BufferedReader. The constructor method will create a BufferedReader, which will then be used whenever a read operation is requested. Note that the KeyboardReader() has five public methods. The getKeyboardInput() method returns a String. This is the method we will call when we just want to get the string that the user typed from the keyboard. The getKeyboardInteger() method returns an int value. This is the method we will call when we want an integer from the keyboard. Similarly, the getKeyboardDouble() method returns a double. This is the method we

will call when we want to input a floating point value from the keyboard. Finally, the prompt() and display() methods will be used to perform two other important tasks of a user interface: that of prompting the user and that of displaying the program's output.

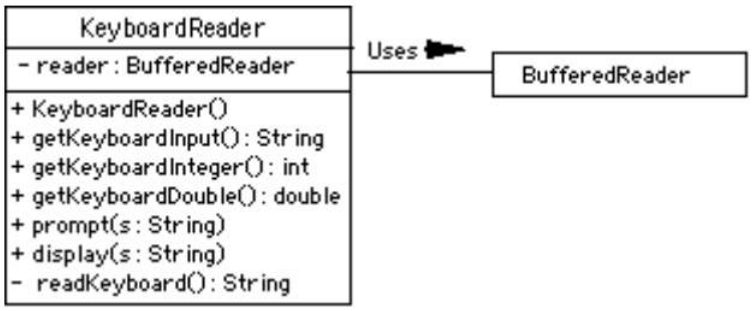


Figure 4.6: Design of the KeyboardReader class.

The following code segment illustrates how we will use a KeyboardReader object to input an integer:

```
KeyboardReader cmdline = new KeyboardReader();
int m = cmdline.getKeyboardInteger();
```

All we need to do is create an instance of the KeyboardReader and ask it to get an integer for us. This greatly simplifies the work we would have to do when we want to perform keyboard input.

Note that Figure 4.6 lists a private method named readKeyboard() in the KeyboardReader class. This is the method that does the actual work of reading data from the keyboard. Because it is private, it can only be called by the other methods in KeyboardReader. It cannot be called by other classes. The reason we make it private is to hide it, and the messy details of performing keyboard input, from other classes.

One of those messy details is the fact that whenever I/O is performed, it is possible for things to go wrong. The possibility of errors occurring applies to all forms of I/O, not just keyboard I/O. For example, when a program is trying to read a file, the file might be missing. Or when trying to download a web page, the Internet connection might malfunction.

Because these types of external errors are possible, Java requires that whenever a program performs certain types of I/O, it must watch out for certain kinds of error conditions, known as *exceptions*. Exceptions are covered in Chapter 11, so we will not attempt to cover them here. Instead, we will design the readKeyboard() method to take care of this detail for us.

JAVA LANGUAGE RULE

Exceptions. Java I/O methods require that programs check for certain error conditions during input.

Figure 4.7 gives the full implementation (for now) of the KeyboardReader class. Lets go through it line by line. The first thing to notice is the use of the import statement. Recall that importing a Java package enables us to refer to elements in the package by their short names (BufferedReader), rather than by their fully qualified names (java.io.BufferedReader).

```
import java.io.*;

public class KeyboardReader
{   private BufferedReader reader;

    public KeyboardReader() {
        reader = new BufferedReader
           (new InputStreamReader(System.in));
    }
    public String getKeyboardInput()
    {   return readKeyboard();
    }
    public int getKeyboardInteger()
    {   return Integer.parseInt(readKeyboard());
    }
    public double getKeyboardDouble()
    {   return Double.parseDouble(readKeyboard());
    }
    public void prompt(String s)
    {   System.out.print(s);
    }
    public void display(String s)
    {   System.out.print(s);
    }
    private String readKeyboard()
    {   String line = "";
        try
        {  line = reader.readLine();
        } catch (IOException e)
        {  e.printStackTrace();
        }
        return line;
    }
}
```

Figure 4.7: Definition of the KeyboardReader class.

Next notice how we create a BufferedReader object in the KeyboardReader() constructor:

```
reader = new BufferedReader
    (new InputStreamReader (System.in));
```

The resulting reader object will persist as long as our KeyboardReader object exists and can be used for all subsequent input operations.

Next notice the definition of the readKeyboard() method. It calls the inherited readLine() method to input a line from the keyboard and then it returns the line. Note, however, how the call to the readLine() method is embedded in a try...catch block. This is one way to handle the possibility that an exception might occur during the input operation. Java requires that our program do something to address the possibility of an I/O exception, and as we will learn in Chapter 11, there are other designs that we might have used here. The primary advantage of doing it this way is that we can hide this language detail from the rest of the program. The rest of the program—and any other programs that use the KeyboardReader class—will not have to worry about this exception issue. They can just ask the KeyboardReader to get them a string or an integer and it will deliver the goods.

Next, notice how the public input methods are defined. The getKeyboardInput() method just returns the line that it gets by calling readKeyboard(). The getKeyboardInteger() method also calls readKeyboard(), but instead of just returning the line, it extracts an integer from it and returns the integer. The getKeyboardDouble() method works the same way.

Finally, notice how the public output methods are defined. Both the prompt() and display() methods take a single String parameter and do exactly the same thing–they merely print their string. So why do we have two methods when one will suffice? The answer is that these methods encapsulate important and distinct user-interface functions—prompting the user and displaying output—that just happen to be implemented in exactly the same way in this case. As we will see when we design our GUI interface, we will use completely different objects to prompt the user and display output. So, despite their similarities, it is important that we distinguish the task of prompting the user from the more general task of displaying output.

Designing a Command-Line Interface

Now that we have defined a special class for performing keyboard input, we now show how it can be used as a user interface in cooperation with the other objects that make up a program. As described in Figure 4.1, the user interface will serve as an intermediary between the user and some type of computational object. Although our command-line interface should work with any application, no matter how complex, we begin with a very simple computational problem. This will allow us to focus on the user interface.

Let's design a program that prompts the user for his or her name and then says hello. Thus, the program's I/O should look like this:

```
Hi, please input your name here > Kim
Hi Kim, nice to meet you.
```

In the design we use there will be two primary objects involved. One will serve as the user interface. This will be our KeyboardReader. A second object will serve as the computational object. In this case it will "compute" an appropriate greeting. It will serve contain the main() method and will encapsulate the algorithm for this application. It will use a KeyboardReader to handle its I/O needs.

The main advantage of this division of labor is that it enables us to use the KeyboardReader, as is, with virtually any Java application. Moreover, despite its simplicity, our computational object in this example can serve as a template for future programs.

JAVA EFFECTIVE DESIGN

Modularity. By designing the user interface as a self-contained module, we can use it with just about any application.

Figure 4.8 provides the details the design we wish to implement. Note that GreeterApp contains an instance variable for a KeyboardReader. This will enable it to use the KeyboardReader whenever it needs to perform keyboard input. By giving GreeterApp a main() method, we allow it to be the main class for our application. Its run() method will contain the algorithm that controls the application, and its greet() method will handle the task of greeting the user.

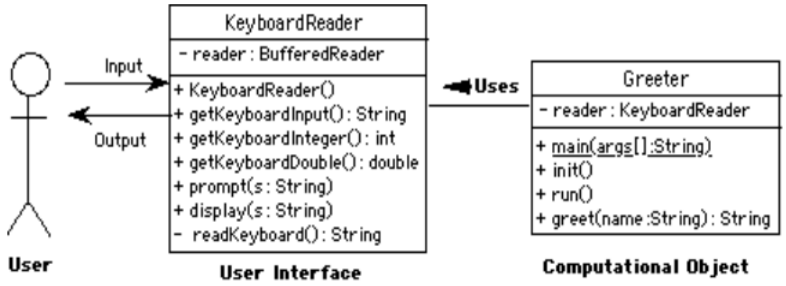


Figure 4.8: Using KeyboardReader as the user interface.

The full implementation of the GreeterApp class is shown in Figure 4.9. It begins by declaring an instance variable for the KeyboardReader, which is instantiated in the constructor method. This gives GreeterApp a way to refer directly to the user interface whenever it needs keyboard input. The run() method encapsulates the application's algorithm. Notice how it uses the

KeyboardReader to prompt the user, to input the user's name, and then to display the greeting. Finally, the main() method serves to create an instance of the computational object and calls its run() method.

```
public class GreeterApp
{   private KeyboardReader reader;

    public GreeterApp()
    {   reader = new KeyboardReader();
    } // GreeterApp()
    public void run()
    {   String name = "";
        reader.prompt("Please input your name here > ");
        name = reader.getKeyboardInput();
        reader.display(greet(name) + "\n");
    } // run()
    public String greet(String name)
    {   return "Hi " + name + " nice to meet you.";
    } // greet()
    public static void main(String args[])
    {   GreeterApp app = new GreeterApp();
        app.run();
    }
} // GreaterApp
```

Figure 4.9: Definition of the GreeterApp class.

To re-cap, we have designed a simple command-line interface that can be used, with minor changes, for virtually any programming task in subsequent chapters. Before moving on, it may be helpful to touch on some of the important object-oriented principles that went into our design.

- Divide-and-conquer: We see the usefulness of dividing a program into separate objects, one to handle the computations required by the application, and one to handle the user interface.
- Encapsulation: The classes we designed encapsulate just the information and behavior that is necessary to perform their specific roles.
- Information hiding: We use a private method to hide certain messy implementation details from other parts of the program.
- Generality and Extensibility: We have developed a design that is general enough that it can be extended to other applications.

? Exercise 4.3.1

Java's Math class has a static method that will generate a random number between 0 and 0.99999999—that is, between 0 and 1, not including 1. By using simple arithmetic, we can generate random numbers between any two values. For example, the following statement assigns a random integer between 1 and 100 to the variable:

```
secretNumber = 1 + (int)(Math.random() * 100);
```

Given this statement, design and implement an application that will play the following guessing game with the user. The computer generates a random number between 1 and 100 and then lets the user guess the number, telling the user when the guess is too high or too low. Note that for this problem, the user will have to input integers at the keyboard.

This page titled 4.3: A Command-line Interface is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **4.3: A Command-line Interface** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

4.4: A Graphical User Interface (GUI)

While command-line interfaces are useful, one of the great advantages of the Java language is that its extensive class library makes it relatively easy to develop applications that employ Graphical User Interfaces (GUIs). GUIs have been around now for many years, since the production of the Macintosh in the early 1980s. Today nearly all the personal computing applications are GUI-based. Therefore, it is important that beginning programmers be able design and write programs that resemble, albeit on a simpler scale, those programs that they use every day. Among other benefits, developing the ability to write GUI programs, like the ones everyone uses today, will make it easier for you to show off your work to others, which might help motivate further interest in learning to program.

In this and subsequent sections, we will develop an extensible GUI model that can be used with either a Java application or an applet. By *extensible* we mean a model that can be easily adapted and used in a wide variety of programs. GUI programming involves a computational model known as **event-driven programming**, which means that GUI programs react to events that are generated mostly by the user's interactions with elements in the GUI. Therefore, we will have to learn how to use Java's *event model* to handle simple events.

Given that this is our first look at some complex topics, we will keep the discussion as simple as possible. This means we will delay discussion of certain issues, which we take up in more depth in Chapter 13.

Java's GUI Components

The Java library comes with two separate but interrelated packages of GUI components, the older java.awt package and the newer javax.swing package. For the most part, the **Swing** classes supersede the **AWT** classes. For example, the java.awt.Button class is superseded by the javax.swing.JButton class, and the java.awt.TextField class is superseded by the javax.swing.JTextField class. As these examples show, the newer Swing components add an initial 'J' to the names of their corresponding AWT counterparts.

Figure 4.10 illustrates how some of the main components appear in a GUI interface. As shown there, a JLabel is simply a string of text displayed on the GUI, used here as a prompt. A JTextField is an input element that can hold a single line of text. In this case, the user has input his name. A JTextArea is an output component that can display multiple lines of text. In this example, it displays a simple greeting. A JButton is a labeled **control element**, which is an element that allows the user to control the interaction with the program. In this example, the user will be greeted by the name input into the JTextField, whenever the JButton is clicked. As we will learn, clicking on the JButton causes an event to occur, which leads the program to take the action of displaying the greeting. Finally, all of these components are contained in a JFrame, which is a top-level container. A **container** is a GUI component that can contain other GUI components.



Figure 4.10: Various GUI components from the javax.swing package. **[Artwork: We need to label the components.]**

The Swing classes are generally considered to be superior to their AWT counterparts. For one thing, Swing components use a sophisticated object-oriented design known as the **model-view-controller (MVC)** architecture, which gives them much greater functionality than their AWT counterparts. For example, whereas an AWT Button can only have a string as its label, a Swing JButton can use an image as a label. (See Chapter 13 for a detailed discussion of the MVC architecture.)

Second, Swing components are written entirely in Java which makes them more portable and enables them to behave the same way regardless of the operating system on which they are run. Because of their portability, Swing components are considered *lightweight*. By contrast, AWT classes use routines that are implemented in the underlying operating system and are therefore not easily portable. Hence, they are considered *heavyweight* components. Whereas a Swing JButton should look and act the same way regardless of platform, an AWT Button would have a different implementation, and hence a different look and feel, on a Macintosh and on a Windows system. In this book, we will use the new Swing classes in our programs.

Class Inheritance: Extending a Superclass

As you recall from Chapter 0, class inheritance is the mechanism by which a class of objects can acquire (*inherit*) the methods and variables of its superclasses. Just as a horse, by membership in the class of horses, inherits those attributes and behaviors of a mammal, and, more generally, those of an animal, a Java subclass inherits the variables and methods of its superclasses. We sometimes lump together an object's attributes and behaviors and refer to them collectively as its *functionality*. So we say that an object of a subclass inherits the functionality of all of its superclasses.

By the same token, just as a horse and a cow extend their mammalian attributes and behaviors in their own special ways, a Java subclass extends the functionality of its superclasses in its own special way. Thus, a subclass *specializes* its superclass.

In Chapter 3, we showed how all classes in the Java hierarchy inherit the toString() method from the Object class. The lesson there was that an object in a subclass can either use or override any public method defined in any of its superclasses. In order to implement GUI programs, we need to look at another way to employ inheritance. In particular, we need to learn how to define a new class by *extending* an existing class.

We noted in Chapter 2 that unless a class is explicitly defined as a subclass of some other class it is considered implicitly to be a direct subclass of Object. Thus, the GreeterApp class that we defined earlier in this chapter is a subclass of Object. We can make the relationship between GreeterApp and Object explicit by using the extends keyword when we define the GreeterApp class:

```
public class GreeterApp extends Object { ... }
```

Thus, the extends keyword is used to specify the subclass/superclass relationships that hold in the Java class hierarchy. We sometimes refer to the subclass/superclass relationship as the *isa* relationship, in the sense that a horse *isa* mammal, and a mammal *isa* animal. Thus, the extends keyword is used to define the *isa* relationship among the objects in the Java class hierarchy.

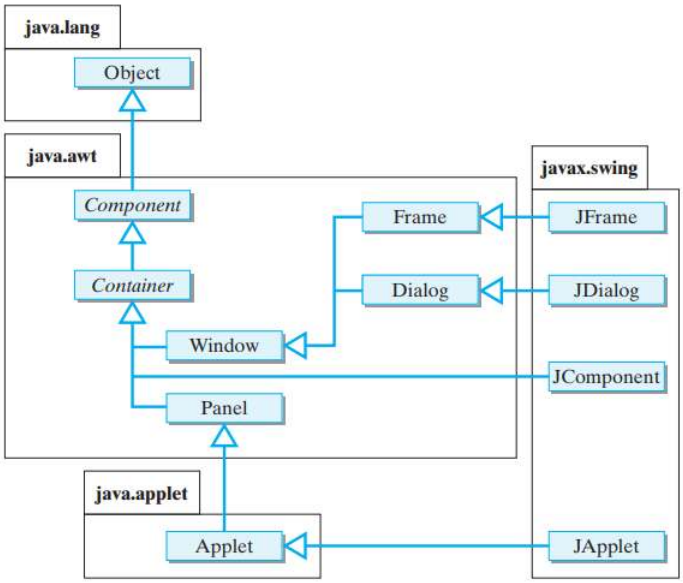


Figure 4.11: Top-level Swing and AWT classes. [NOTE: REDRAW JWindow is a subclass of Window.]

A **top-level container** is a GUI container that cannot be added to another container; it can only have components added to it. Figure 4.11 is a class hierarchy that shows the relationships among some of the top-level Swing and AWT classes. For example, the javax.swing.JFrame class, which represents a top-level window, is a subclass of java.awt.Frame, and the javax.swing.JPanel is a

subclass of java.awt.Panel. We can see from this figure that a JFrame *isa* Frame and an Frame *isa* Window and a Window *isa* Container. These subclass/superclass relationships are created in their respective class definitions by using the extends keyword as follows:

```
public class JFrame extends Frame { ... }
public class Frame extends Window { ... }
public class Window extends Container { ... }
```

As we will see in the next section, extending a class in this way enables us to create a new class by specializing an existing class.

Top-level Windows

Referring again to Figure 4.11, notice that all of the Swing components are subclasses of the AWT Container class. This means that Swing components are Containers. They inherit the functionality of the Container class. So Swing components can contain other GUI components. That is why a JButton can contain an image.

All GUI programs must be contained inside some kind of top-level container. Swing provides three top-level container classes: JFrame, JApplet and JDialog. For our basic GUI, we will use a JFrame as the top-level window for stand alone applications.

A JFrame encapsulates the basic functionality of a top-level window. It has what is called a *content pane*, to which other Swing components, such as buttons and text fields, can be added. Also, it comes with enough built-in functionality to respond to certain basic commands, such as when the user adjusts its size or closes it.

Figure 4.12 shows a simple top-level window as it would be displayed on the console. This window has a title ("My GUI"). It is 200 pixels wide, 150 pixels high, and its top-left corner is located at coordinates (100,150) on the console screen. Like in other graphical systems, points on the Java console always given as an ordered pair, *(X, Y)*, with the horizontal coordinate, *X*, listed first, followed by the vertical coordinate, *Y*. The horizontal x-axis extends positively from left to right, and the vertical y-axis extends positively from top to bottom.

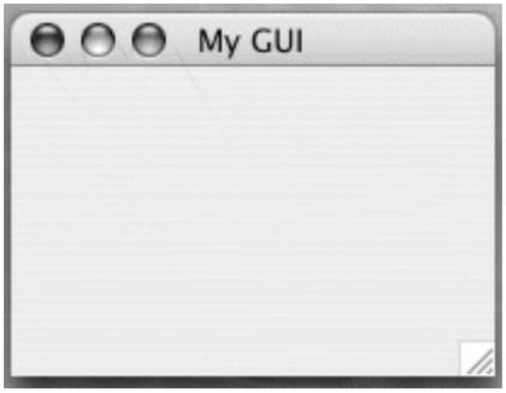


Figure 4.12: A simple window.

The class that created and displayed this window is shown in Figure 4.13.

```
import javax.swing.*;

public class SimpleGUI extends JFrame
{
    public SimpleGUI(String title)
    {   setSize(200,150);
        setLocation(100, 150);
        setTitle(title);
        setVisible(true); // Displays the JFrame
    } // SimpleGUI()

    public static void main(String args[])
    {   new SimpleGUI("My GUI");
    } // main()
} // SimpleGUI class
```

Figure 4.13: A top-level window with a title.

Note the use of the extends keyword to define SimpleGUI as a subclass of JFrame. As a subclass, SimpleGUI inherits all of the functionality of a JFrame (Fig. 4.14) . That is, it can contain other GUI components. It knows how to resize and close itself, and so on. The reason we want to define a subclass of JFrame, rather than just use a JFrame instance, is because we want eventually to give our subclass additional functionality that is specialized for our application.

JAVA EFFECTIVE DESIGN

Specialization. By creating a subclass of JFrame we can specialize its functionality for our application.

Note how SimpleGUI's main() program creates an instance of SimpleGUI by invoking its constructor. There is no need to use a variable here because there are no further references to this object in this class. However, simply constructing a SimpleGUI will not cause it to appear on the Java console. For that to happen, it is necessary to give it a size and to call its setVisible() method. This is done in the constructor method.

The constructor method illustrates how to use some of the methods inherited from JFrame. Figure 4.14 shows some of the methods that SimpleGUI inherits from JFrame. We use the setSize() and setLocation() methods to set SimpleGUI's size and location. We use the setTitle() method to set its title. And we use the setVisible() method to cause it to appear on the console.

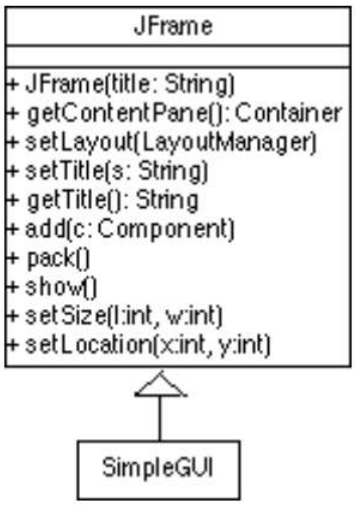


Figure 4.14: SimpleGUI is a subclass of JFrame.

GUI Components for Input, Output, and Control

To enable our top-level window to serve as a user interface, it will be necessary to give it some components. Figure 4.15 provides an overview of some of the main Swing components. Generally, there are three types of components, which correspond to the three main functions of a user interface: *input*, *output*, and *control*. A JTextField would be an example of an input component. The user can type text into the text field, which can then be transmitted into the program. A JTextArea is an example of an output component. The program can display text in the text area. Control components enable the user to control the actions of the program. A JButton would be an example of a control component. It can be associated with an action that can be initiated whenever the user clicks it. We might also consider a JLabel to be an output component, because we can use it to prompt the user as to what type of actions to take.

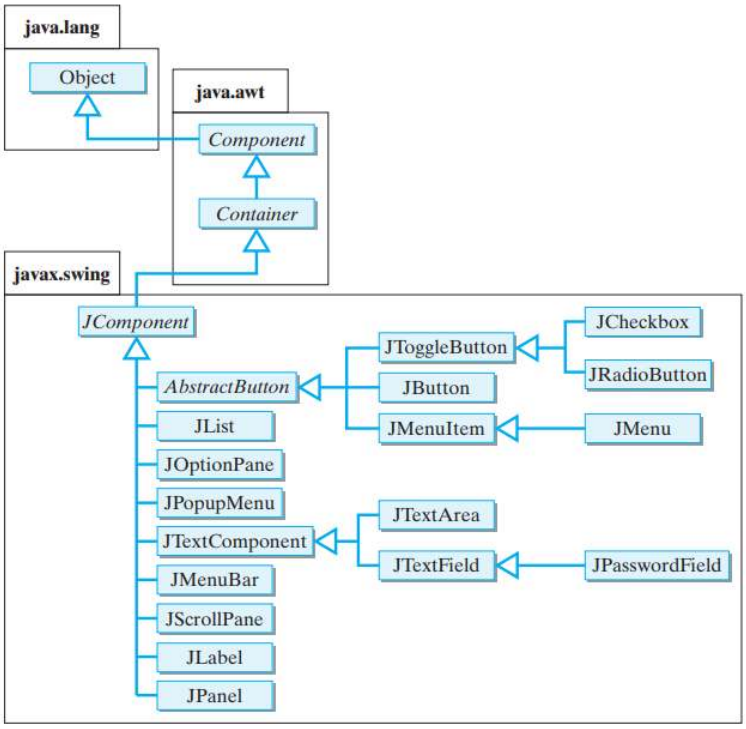


Figure 4.15: Swing components.

Let's begin by creating a simple user interface, one that enables us to perform basic input, output, and control operations with a minimum of Swing components. This will allow us to demonstrate the basic principles and techniques of user-interface design and will result in a GUI that can be extended for more sophisticated applications. For this example, we will limit our application to that of simply greeting the user, just as we did in designing our command-line interface. That means that the user will be prompted to input his or her name and the program will respond by displaying a greeting (Fig. 4.10). We will call our GUI GreeterGUI, to suggest its interdependence with the same Greeter computational object that we used with the command-line interface.

For this simple application, our GUI will make use of the following components:

- A JTextField will be used to accept user input.
- A JTextArea will serve to display the program's output.
- A JButton will allow the user to request the greeting.
- A JLabel will serve as a prompt for the JTextField.

Figure 4.16 shows some of the constructors and public methods for the JTextArea, JTextField, JButton, and JLabel components.

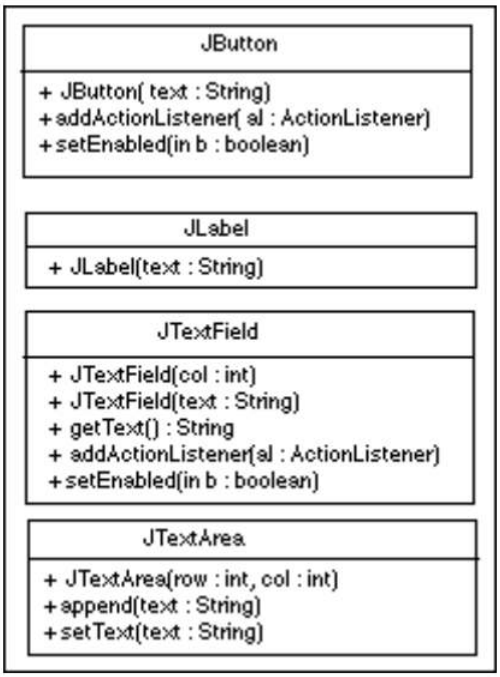


Figure 4.16: Public methods and constructors for basic Swing components.

The following code segments illustrate how to use these constructors to create instances of these components:

```
   // Declare instance variables for the components
private JLabel prompt;
private JTextField inField;
private JTextArea display;
private JButton goButton;
   // Instantiate the components
prompt = new JLabel("Please type your name here: ");
inField = new JTextField(10);   // 10 chars wide
display = new JTextArea(10, 30);// 10 rows x 30 columns
goButton = new JButton("Click here for a greeting!");
```

For this example, we use some of the simpler constructors. Thus, we create a JTextField with a size of 10. That means it can display 10 characters of input. We create a JTextArea with 10 rows of text, each 30 characters wide. We create a JButton with a simple text prompt meant to inform the user of how to use the button.

Adding GUI Components to a Top-Level Window

Now that we know how to create GUI components, the next task is to add them to the top-level window. A JFrame is a top-level Container (Fig. 4.11), but instead of adding the components directly to the JFrame we have to add them to the JFrame's content pane, which is also a Container.

Java's Container class has several add() methods that can be used to insert components into the container:

```
add(Component comp)// add comp to end of  container
add(Component comp, int index)// add comp at index
add(String region, Component comp) add comp at region
```

The particular add() method to use depends on how we want to arrange the components in the container. The layout of a container is controlled by its default **layout manager**, an object associated with the container that determines the sizing and the arrangement

of its contained components. For a content pane, the default layout manager is a BorderLayout. This is an arrangement whereby components may be placed in the center of the pane and along its north, south, east, and west borders (Fig. 4.17).

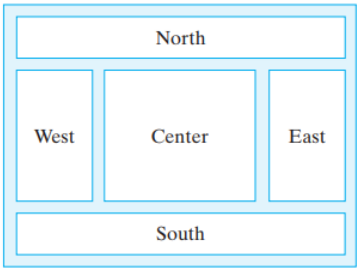


Figure 4.17: Arrangement of components in a border layout.

Components are added to a border layout by using the add(String region, Component comp) method, where the String parameter specifies either "North," "South," "East," "West," or "Center." For example, to add the JTextArea to the center of the JFrame we first create a reference to its content pane and we then add the component at its center:

```
Container contentPane = getContentPane(); // Get pane
contentPane.add("Center",display); // Add JTextArea
```

One limitation of the border layout is that only one component can be added to each area. This is a problem for our example because we want our prompt JLabel to be located right before the JTextField. To get around this problem, we will create another container, a JPanel, and add the prompt, the text field, and the goButton to it. That way, all of the components involved in getting the user's input will be organized into one panel. We then add the entire panel to one of the areas on the content pane.

```
JPanel inputPanel = new JPanel();
inputPanel.add(prompt);   // Add JLabel to panel
inputPanel.add(inField);  // Add JTextField to panel
inputPanel.add(goButton); // Add JButton to  panel
contentPane.add("South", inputPanel); // Add to JFrame
```

The default layout for a JPanel is FlowLayout, which means that components are added left to right with the last addition going at the end of the sequence. This is an appropriate layout for this JPanel because it will place the prompt just to the left of the input JTextField.

JAVA EFFECTIVE DESIGN

Encapsulation. JPanels can be used to group related components in a GUI.

Controlling the GUI's Action

Now that we know how to place all the components on the GUI, we need to design the GUI's controls. As mentioned earlier, GUIs use a form of event-driven programming. Anything that happens when you are using a computer—every keystroke and mouse movement—is classified as an event. As Figure 4.18 illustrates, events are generated by the computer's hardware and filtered up through the operating system and the application programs. Events are handled by special objects called listeners. A **listener** is a specialist that monitors constantly for a certain type of event. Some events, such as inserting a CD in the CD-ROM drive, are handled by listeners in the operating system. Others, such as typing input into a Web page or a Word document, are handled by listeners in a piece of application software, such as a browser or a word processor.

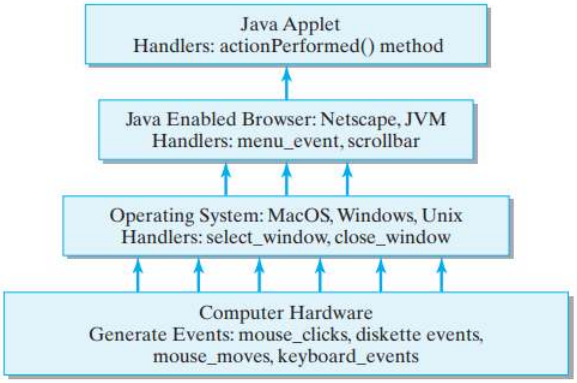


Figure 4.18: Java's event model.

In an event-driven programming model, the program is controlled by an **event loop**. That is, the program repeatedly listens for events, taking some kind of action whenever an event is generated. In effect, we might portray this event loop as follows:

```
Repeat forever or until the program is stopped
   Listen for events
   If event-A occurs, handle it with event-A-handler
   If event-B occurs, handle it with event-B-handler
    ...
```

The event loop listens constantly for the occurrence of events and then calls the appropriate object to handle each event.

Figure 4.19 shows some of the main types of events in the java.awt.event package. In most cases, the names of the event classes are suggestive of their roles. Thus, a MouseEvent occurs when the mouse is moved. A KeyEvent occurs when the keyboard is used. The only event that our program needs to listen for is an ActionEvent, the type of event that occurs when the user clicks the JButton.

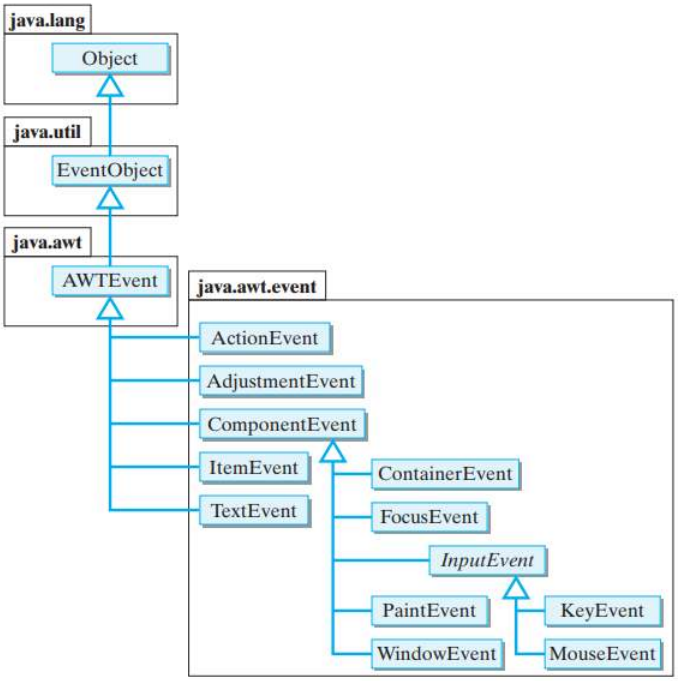


Figure 4.19: Java's event hierarchy.

When the user clicks the JButton, Java will create an ActionEvent object. This object contains important information about the event, such as the time that the event occurred and the object, such as a JButton, that was the locus of the event. For our application, when the user clicks the JButton, the program should input the user's name from the JTextField and display a greeting, such as "Hi John nice to meet you" in the JTextArea. That is, we want the program to execute the following code segment:

```
String name = inField.getText();
display.append(greeter.greet(name) + "\n");
```

The first line uses the JTextField.getText() method to get the text that the user typed into the JTextField and stores it in a local variable, name. The second line passes the name to the greeter.greet() method and passes the result it gets back to the JTextArea.append() method. This will have the effect of displaying the text at the end of the JTextArea.

In this example, we have used a couple of the standard public methods of the JTextField and JTextArea classes. For our simple GUI, the methods described in Figure 4.16 will be sufficient for our needs. However, if you would like to see the other methods available for these and other Swing components, you should check Java's online API documentation.

The ActionListener Interface

Given that the code segment just described will do the task of greeting the user, where should we put that code segment in our program? We want that code segment to be invoked whenever the user clicks on the goButton. You know enough Java to understand that we should put that code in a Java method. However, we need a special method in this case, one that will be called automatically by Java whenever the user clicks that button. In other words, we need a special method that the button's listener knows how to call whenever the button is clicked.

Java solves this problem by letting us define a pre-selected method that can be associated with the goButton. The name of the method is actionPerformed() and it is part of the ActionListener interface. In this case, an **interface** is a special Java class that contains only methods and constants (final variables). It cannot contain instance variables. (Be careful to distinguish this kind of interface, a particular type of Java class, form the more general kind of interface, whereby we say that a class's public methods make up its interface to other objects.) Here's the definition of the ActionListener interface:

```
public abstract interface ActionListener
                                extends EventListener
{    public abstract void actionPerformed(ActionEvent e);
}
```

This resembles a class definition, but the keyword interface replaces the keyword class in the definition. Note also that we are declaring this interface to be abstract. An **abstract interface** or **abstract class** is one that contains one or more abstract methods. An **abstract method** is one that consists entirely of its signature; it lacks an implementation—that is, it does not have a method body. Note that the actionPerformed() method in ActionListener places a semicolon where its body is supposed to be.

JAVA LANGUAGE RULE

Java Interface. A Java interface is like a Java class except that it cannot contain instance variables.

JAVA LANGUAGE RULE

Abstract Methods and Classes. An abstract method is a method that lacks an implementation. It has no method body.

Declaring a method abstract means that we are leaving its implementation up to the class that implements it. This way, its implementation can be tailored to a particular context, with its signature specifying generally what the method should do. Thus, actionPerformed() should take an ActionEvent object as a parameter and perform some kind of action.

What this means, in effect, is that any class that implements the actionPerformed() method can serve as a listener for ActionEvents. Thus, to create a listener for our JButton, all we need to do is give an implementation of the actionPerformed() method. For our

program, the action we want to take when the goButton is clicked, is to greet the user by name. Thus, we want to set things up so that the following actionPerformed() method is called whenever the goButton is clicked:

```
 public void actionPerformed(ActionEvent e)
{  if (e.getSource() == goButton)
   {   String name = inField.getText();
         display.append(greeter.greet(name) + "\n");
   }
 }
```

In other words, we place the code that we want executed when the button is clicked in the body of the actionPerformed() method. Note that in the if-statement we get the source of the action from the ActionEvent object and check that it was the goButton.

That explains what gets done when the button is clicked—namely, the code in actionPerformed() will get executed. But it doesn't explain how Java knows that it should call this method in the first place. To set that up we must do two further things. We must place the actionPerformed() method in our GreeterGUI class, and we must tell Java that GreeterGUI will be the ActionListener for the goButton.

The following stripped-down version of the GreeterGUI class illustrates how we put it all together:

```
 public class GreeterGUI extends Frame
                             implements ActionListener
{ ...
  public void buildGUI()
  {  ...
     goButton = new JButton("Click here for a greeting!");
     goButton.addActionListener(this);
     ...
  }
  ...
  public void actionPerformed(ActionEvent e)
  {   if (e.getSource() == goButton)
      {   String name = inField.getText();
          display.append(greeter.greet(name) + "\n");
      }
  }
  ...
}
```

First, we declare that GreeterGUI implements the ActionListener interface in the class header. This means that the class must provide a definition of the actionPerformed() method, which it does. It also means that GreeterGUI *isa* ActionListener. So SimpleGUI is both a JFrame and an ActionListener.

Second, note how we use the addActionListener() method to associate the listener with the goButton:

```
goButton.addActionListener(this)
```

The this keyword is a self-reference—that is, it always refers to the object in which it is used. It's like a person referring to himself by saying "I". When used here, the this keyword refers to this GreeterGUI. In other words, we are setting things up so that the GreeterGUI will serve as the listener for action events on the goButton.

JAVA LANGUAGE RULE

This Object. The this keyword always refers to the object that uses it. It is like saying “I” or “me.”

Connecting the GUI to the Computational Object

Figure 4.20 gives the complete source code for our GreeterGUI interface. Because there is a lot going on here, it might be helpful to go through the program carefully even though we have introduced most of its elements already. That will help us put together all of the various concepts that we have introduced.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class GreeterGUI extends JFrame
                            implements ActionListener
{ private JTextArea display;
  private JTextField inField;
  private JButton goButton;
  private Greeter greeter;

  public GreeterGUI(String title)
  { greeter = new Greeter();
    buildGUI();
    setTitle(title);
    pack();
    setVisible(true);
  } // GreeterGUI()
  private void buildGUI()
  { Container contentPane = getContentPane();
    contentPane.setLayout(new BorderLayout());
    display = new JTextArea(10,30);
    inField = new JTextField(10);
    goButton = new JButton("Click here for a greeting!");
    goButton.addActionListener(this);
    JPanel inputPanel = new JPanel();
    inputPanel.add(new JLabel("Input your name here: "));
    inputPanel.add(inField);
    inputPanel.add(goButton);
    contentPane.add("Center", display);
    contentPane.add("South", inputPanel);
  } // buildGUI()
  public void actionPerformed(ActionEvent e)
  { if (e.getSource() == goButton)
    { String name = inField.getText();
      display.append(greeter.greet(name) + "\n");
    }
  } // actionPerformed()
}
```

Figure 4.20: Definition of the GreeterGUI class.

To begin with, note the several Java packages that must be included in this program. The javax.swing package includes definitions for all of the Swing components. The java.awt.event package includes the ActionEvent class and the ActionListener interface, and the java.awt packages contain the Container class.

Next note how the GreeterGUI class is defined as a subclass of and as implementing the ActionListener interface. GreeterGUI thereby inherits all of the functionality of a JFrame. Plus, we are giving it additional functionality. One of its functions is to serve as an ActionListener for its goButton. The ActionListener interface consists entirely of the actionPerformed() method, which is defined in the program. This method encapsulates the actions that will be taken whenever the user clicks the goButton.

The next elements of the program are its four instance variables, the most important of which is the Greeter variable. This is the variable that sets up the relationship between the GUI and the computational object. In this case, because the variable is declared in the GUI, we say that the GUI uses the computation object, as illustrated in Figure 4.8. This is slightly different from the relationship we set up in the command-line interface, in which the computational object uses the interface (Fig. 4.2).

The other instance variables are for those GUI components that must be referred to throughout the class. For example, note that the goButton, inField, and display are instantiated in the buildGUI() method and referenced again in the actionPerformed() method.

The next element in the program is its constructor. It begins by creating an instance of the Greeter computational object. It is important to do this first in case we need information from the computational object in order to build the GUI. In this case we don't need anything from Greeter, but we will need such information in other programs.

We've already discussed the fact that the constructor's role is to coordinate the initialization of the GreeterGUI object. Thus, it invokes the buildGUI() method, which takes care of the details of laying out the GUI components. And, finally, it displays itself by calling the pack() and setVisible() methods, which are inherited from JFrame. The pack() method sizes the frame according to the sizes and layout of the components it contains. The setVisible() method is what actually causes the GUI to appear on the Java console.

Finally, note the details of the buildGUI() method. We have discussed each of the individual statements already. Here we see the order in which they are combined. Note that we can declare the contentPane and inputPanel variables locally, because they are not used elsewhere in the class.

? Exercise 4.4.1

There is a simple modification that we can make to GreeterGUI. The JTextField can serve both as an input element and as a control element for action events. An ActionEvent is generated whenever the user presses the Return or Enter key in a JTextField so that the JButton can be removed. Of course, it will be necessary to designate the inField as an ActionListener in order to take advantage of this feature. Make the appropriate changes to the buildGUI() and actionPerformed() methods so that the inField can function as both a control and input element. Call the new class GreeterGUI2

Using the GUI in a Java Application

As you know, a Java application is a stand alone program, one that can be run on its own. We have designed our GUI so that it can easily be used with a Java application. We saw in the previous section that the GUI has a reference to the Greeter object, which is the computational object. Therefore, all we need to get the program to run as an application is a main() method.

One way to use the GUI in an application is simply to create an instance in a main() method. The main() method can be placed in the GreeterGUI class itself or in a separate class. Here's an example with the main in a separate class:

```
public class GreeterApplication
{  public static void main(String args[])
   {
      new GreeterGUI("Greeter");
   }
}
```

The main() method creates an instance of GreeterGUI, passing it a string to use as its title. If you prefer, this same main() method can be incorporated directly into the GreeterGUI class.

This page titled 4.4: A Graphical User Interface (GUI) is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **4.4: A Graphical User Interface (GUI)** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

4.5: Case Study- The One Row Nim Game

In this section, we show how to develop alternative interfaces for our case study game of One Row Nim that was developed in the two previous chapters. As you recall, the One Row Nim game starts with, say, 21 sticks on a table. Players take turns picking up 1, 2 or 3 sticks, and the player to pick up the last stick loses. We wish to develop an application program so that the user of the program can play this game against the computer, that is, against the program.

As in our other examples in this chapter, our design will divide this problem into two primary objects: a computational object, in this case OneRowNim, and a user interface object, for which we will use either a KeyboardReader or a OneRowNimGUI. One goal of our design was to develop the OneRowNim class so that it can be used, without changes, with either a command-line interface or a GUI.

Recall that we designed the OneRowNim class to maintain the state of the game and to provide methods that enforce the rules of the game. Thus, we know that after each legal move, the number of sticks will decline, until it is 0 or less, which indicates that the game is over. Also, an instance of OneRowNim keeps track of whose turn it is and can determine if the game is over and who the winner is when the game is over. Finally, the game ensures that players cannot cheat, either by taking too few or too many sticks on one turn. Figure 4.21 shows the UML diagram of the OneRowNim class as described at the end of the previous chapter.



Figure 4.21: A UML diagram of the OneRowNim class.

A Command-line Interface to OneRowNim

Let's now focus on connecting a OneRowNim instance with a KeyboardReader instance, the command-line interface we developed at the beginning of this chapter. To do so requires no changes to KeyboardReader (Fig. 4.6). Unlike in the greeter example, we will use a third object to serve as the main program. As shown in Figure 4.22, the OneRowNimApp class will contain the run() method that controls the game's progress. OneRowNimApp will use the KeyboardReader object to prompt the user, to display the program's output, and to perform input from the keyboard. It will use the OneRowNim object to keep track of the game.

In fact, the main challenge for this part of our problem is designing the run() method, which will use a loop algorithm to play the game. The user and the computer will repeatedly take turns picking up sticks until the game is over. The game is over when there are no more sticks to pick up. Thus, we can use the game's state—the number of sticks left—as our loop's entry condition. We will repeat the loop while there are more than 0 sticks remaining.

The following pseudocode describes the remaining details of our algorithm. We refer to the OneRowNim instance as the game object, and we refer to the KeyboardReader instance as the reader object. We use the notation game:get the number of sticks left to indicate that we are sending a message to the game object.

```
Create a game object with 21 sticks
Create a reader object
sticksLeft = game:get the number of sticks left
reader:display the rules of the game
while (game: the game is not over)
    whoseMove = game: find out whose turn it is
    if (whoseMove == user)
        game: user chooses number of sticks to take
    else
        game: computer chooses number of sticks to take
    sticksLeft = game: get the number of sticks left
    reader: report the number of sticks left
// At this point the game is over.
if game: the user is the winner
       reader: report that the user wins
else
      reader: report that the computer wins
```

In this algorithm, the initializations we perform consist of creating the game and reader objects and initializing sticksLeft. We use a while loop structure to control the game. The loop's entry condition is that the 'the game is not over'. This is a piece of information that comes directly from the game object. As long as the game is not over, the body of the loop will be executed. Note that in the loop's body, either the player or the computer makes a move. Again, it is up to the game object to determine whose move it is. Following the move we ask the game how many sticks are left and we use the reader object to report this.

Note that the loop structure has the three necessary elements. The initializer in this case is the creation of a OneRowNim object. We know that this will cause the game to have 21 sticks and it will be the user's move. The loop-entry condition is that the game is not over, which is based on the fact that there are still sticks remaining to be picked up. But again, this knowledge is kept by the game object. Finally, we have an updater that consists of either the computer or the user picking up some sticks. This in turn changes the value of sticksLeft on each iteration, moving us ever closer to the condition that there are no sticks left, at which point the game will be over.

Note that we have left out of this algorithm the details of the user's moves and computer's moves. These are the kinds of actions that are good to put into separate methods, where we can worry about checking whether the user made a legal move and other such details.

Figure 4.22 provides the implementation of the OneRowNimApp application. It uses a KeyboardReader as a command-line interface and a OneRowNim instance as it computational object. Thus, it has private instance variables for each of these objects, which are instantiated in the constructor method. The algorithm we just described has been placed in the run() method, which is called from main() after the application is instantiated. The use of the boolean method gameOver() to control the loop makes this code segment easier to understand. Also, it leaves it up to the game object to determine when the game is over. From an object-oriented design perspective, this is an appropriate division of responsibility. If you doubt this, imagine what could go wrong if this determination was left up to the user interface. A user-interface programmer might end up, mistakenly, implementing the wrong rule for the game being over. A similar point applies to the getWinner() method. This determination rests with the game, not the user interface. If left up to the user interface, it is possible that a programming mistake could lead to the loss of the game's integrity.

```
public class OneRowNimApp
{ private KeyboardReader reader;
  private OneRowNim game;

  public OneRowNimApp()
  { reader = new KeyboardReader();
    game = new OneRowNim(21);
  } //OneRowNim()

  public void run()
  { int sticksLeft = game.getSticks();
    reader.display("Let's play One Row Nim. You go first.\n");
    reader.display("There are " + sticksLeft +
                                    " sticks left.\n");
    reader.display("You can pick up 1, 2, or 3 at a time\n.");
    while (game.gameOver() == false)
    { if (game.getPlayer() == 1)  userMove();
      else computerMove();
      sticksLeft = game.getSticks();
      reader.display("There are " + sticksLeft +
                                     " sticks left.\n");
    } // while
    if (game.getWinner() == 1)
      reader.display("Game over. You win. Nice game.\n");
    else  reader.display("Game over. I win. Nice game.\n");
  } //run()

  private void userMove()
  { reader.prompt("Do you take 1, 2, or 3 sticks?: ");
    int userTakes = reader.getKeyboardInteger();
    if (game.takeSticks(userTakes))
    { reader.display("You take " + userTakes + ".\n");
    } else
    { reader.display("You can't take " + userTakes +
                                        ". Try again\n");
    } // else
  } //userMove()

  private void computerMove()
  { game.takeAway(1);  // Temporary strategy.
    reader.display("I take 1 stick. ");
  } //computerMove()

  public static void main(String args[])
  { OneRowNimApp app = new OneRowNimApp();
    app.run();
  } //main()
} // OneRowNimApp
```

Figure 4.22: Definition of OneRowNimApp, a command-line interface to the OneRowNim.

The run() method calls userMove() and computerMove() to perform the specific set of actions associated with each type of move. The userMove() method uses the KeyboardReader() to prompt the user and input his or her move. It then passes the user's choice to game.takeSticks(). Note how it checks the return value to determine whether the move was legal or not and provides an appropriate response through the interface.

Finally, note how we use private methods to implement the actions associated with the user's and computer's moves. Because these private methods are not part of the object's interface and because they can only be used within the object themselves, they are in a sense secondary to the object's public instance methods. We sometimes refer to them as **helper methods**. This division of labor allows us to organize all of the details associated with the moves into a single module. The computerMove() method uses a temporary strategy of taking a single stick and passes the number `1` to game.takeSticks(). Finally, computerMove() reports its choice through the interface. After we have covered operators of the int data type in the next chapter, we will be able to describe better strategies for the computer to make a move.

This example shows how simple and straightforward it is to use our KeyboardReader user interface. In fact, for this problem, our interface didn't require any changes. Although there might be occasions where we will want to extend the functionality of KeyboardReader, it can be used without changes for a wide variety of problems in subsequent chapters.

JAVA EFFECTIVE DESIGN

Code Reuse. A well-designed user interface can be used with many computational objects.

A GUI for OneRowNim

The first task is designing a GUI for the OneRowNim is to decide how to use input, output, and control components to interact with the user. Following the design we used in the GUI for our greeter application, we can use a JTextField for the user's input and a JTextArea for the game's output. Thus, we will use the JTextArea to report on the progress of the game and to display any error messages that arise. As in the greeter example, we can use both the JTextField and JButton as control elements and a JLabel as a prompt for the input text field. For the most part then, the use of GUI components will remain the same as in our previous example. This is as we would expect. The relationship between the user and the interface are pretty similar in both this and the previous application.

In contrast, the relationship between the interface and the game are quite different from what we saw in the greeter application. As in the previous application, the GUI will still need a reference to its associated computational object, in this case the game:

```
    private OneRowNim game;
    ...
    game = new OneRowNim();
```

The biggest difference between this GUI and the one we used with the greeter application occurs in the details of the interaction between the GUI and the game. These details are the responsibility of the actionPerformed() method, whose actions depend on the actual progress of the individual game.

Unlike in the command-line version, there is no need to use a loop construct in the actionPerformed() method. Instead, because we are using event-driven programming here, we will rely on Java's event loop to move the game from one turn to another.

As in the greeter example, the actionPerformed() method will be called automatically whenever the JButton is clicked. It is the responsibility of the GUI to ensure that it is the user's turn whenever this action occurs. Therefore, we design actionPerformed() so that each time it is called, it first performs the user's move and then, assuming the game is not over and an error did not occur on the user's move, it performs the computer's move. Thus, the basic algorithm is as follows:

```
    Let the user move.
    If game:game is not over and computer turn
       let the computer move.
    Game: how many sticks are left.
    display: report how many sticks are left
    If game:game is over
        Stop accepting moves.
        Report the winner.
```

After the user's move, it is possible that the user picked up the last stick, which means that the game would be over. In that case, the computer would not get a move. Or, the user could have made an error. In that case it would still be the user's move. These possibilities have to be considered in the algorithm before the computer gets to move. As the pseudocode shows, it is the OneRowNim object's responsibility to keep track of whether the game is over and whose turn it is.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class OneRowNimGUI extends JFrame implements ActionListener
{ private JTextArea display;
  private JTextField inField;
  private JButton goButton;
  private OneRowNim game;

  public OneRowNimGUI(String title)
  { game = new OneRowNim(21);
    buildGUI();
    setTitle(title);
    pack();
    setVisible(true);
  } // OneRowNimGUI()
  private void buildGUI()
  { Container contentPane = getContentPane();
    contentPane.setLayout(new BorderLayout());
    display = new JTextArea(20,30);
    display.setText("Let's play Take Away. There are " + game.getSticks() +
      " sticks.\n" + "Pick up 1,2, or 3 at a time.\n" + "You go first.\n");
    inField = new JTextField(10);
    goButton = new JButton("Take Sticks");
    goButton.addActionListener(this);
    JPanel inputPanel = new JPanel();
    inputPanel.add(new JLabel("How many sticks do you take: "));
    inputPanel.add(inField);
    inputPanel.add(goButton);
    contentPane.add("Center", display);
    contentPane.add("South", inputPanel);
  } // buildGUI
  private void userMove()
  { int userTakes = Integer.parseInt(inField.getText());
    if (game.takeSticks(userTakes))
      display.append("You take " + userTakes + ".\n");
    else display.append("You can't take " + userTakes + ". Try again\n");
```

```
  }// userMove()
  private void computerMove()
  { if (game.gameOver()) return;
    if (game.getPlayer() == 2)
    {   game.takeSticks(1); // Temporary strategy
      display.append("I take one stick. ");
    } // if
  } // computerMove()
  private void endGame()
  { goButton.setEnabled(false);  // Disable button and textfield
    inField.setEnabled(false);
    if (game.getWinner() == 1)
      display.append("Game over. You win. Nice game.\n");
    else  display.append("Game over. I win. Nice game.\n");
  } // endGame()
  public void actionPerformed(ActionEvent e)
  { if (e.getSource() == goButton)
    { userMove();
      computerMove();
      int sticksLeft = game.getSticks();
      display.append("There are " + sticksLeft + " sticks left.\n");
      if (game.gameOver()) endGame();
    } // if
  } // actionPerformed()
} // OneRowNimGUI
```

Figure 4.23: The OneRowNimGUI class.

Figure 4.23 shows the complete implementation of the OneRowNimGUI class. In terms of its instance variables, constructor, and its buildGUI() method, there are only a few minor differences between this GUI and the GreeterGUI (Fig. 4.20). This GUI has instance variables for its JTextField, JTextArea, and JButton, as well as one for OneRowNim instance, its computational object. It needs to be able to refer to these objects throughout the class. Hence we give them class scope.

The constructor method plays the same role here as in the previous GUI: It creates an instance of the computational object, builds the GUI's layout, and then displays the interface on the console.

All of the changes in the buildGUI() method have to do with application-specific details, such as the text we use as the prompt and the goButton's label. One new method we use here is the setText() method. Unlike the append() method, which is used to add text to the existing text in a JTextArea, the setText() method replaces the text in a JTextArea or a JTextField.

Next let's consider the private userMove() and computerMove() methods. Their roles are very similar to the corresponding methods in the command-line interface: They encapsulate the details involved in performing the players' moves. The primary difference here is that for the user move we input the user's choice from a JTextField rather than from the keyboard. We use getText() to retrieve the user's input from the JTextField and we use Integer.parseInt() to convert to an int value:

```
int userTakes = Integer.parseInt(inField.getText());
```

Another difference is that we use a JTextField to display the program's messages to the user.

As we have noted, the main differences between this and the GreeterGUI occur in the actionPerformed() method. Note there how we use OneRowNim's public methods, getPlayer(), gameOver() and getWinner() to control the interaction with the user.

One issue that differs substantially from the command-line interface is: How do we handle the end of the game? Because we are using Java's built-in event loop, the GUI will continue to respond to user's events, unless we stop it from doing so. One way to do this is to disable the JButton and the JTextField. By disabling a control element, we render it unable to respond to events. To do this we use the setEnabled() method, passing it the value false to, in effect, "turn off" that component:

```
if (game.gameOver())
{  goButton.setEnabled(false);  // End the game
   inField.setEnabled(false);
   ...
}
```

Although it doesn't apply in this situation, the setEnabled() method can be used repeatedly in a GUI to turn components on and off as the context of the interaction dictates.

This example shows how simple and straightforward it can be to build a GUI for just about any application. One main design issue is deciding what kinds of input, output, and control elements to use. For most applications, we can use JTextField, JTextArea, JLabel, and JButton as the GUI's basic elements. A second design issue concerns the development of the actionPerformed() method, which must be designed in an application-specific way. Here we apply what we've learned regarding Java's event-programming model: We designate one or more of our elements to serve as an ActionListener and we design algorithms to handle the action events that occur on that element.

Of course, for some applications we may need two JTextFields to handle input. At some point, we also might want to introduce JMenus and other advanced GUI elements. Some of these options will be introduced in upcoming chapters. Others will be covered in Chapter 13, which provides a more comprehensive view of Java's GUI capabilities.

JAVA EFFECTIVE DESIGN

GUI Design A well-designed GUI makes appropriate use of input, output, and control elements.

This page titled 4.5: Case Study- The One Row Nim Game is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **4.5: Case Study- The One Row Nim Game** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

4.6: From the Java Library- java.io.File and file input (Optional)

In addition to command-line and GUI user interfaces, there is one more standard user interface, files. In this section we show how the Scanner class, that was used in Chapter 2 for keyboard input, can also read input from files. Reading input from a file is relevant to only certain types of programming problems. It is hard to imagine how a file would be used in playing the One Row Nim game but a file might very well be useful to store a collection of riddles that could be read and displayed by a Java program. We will develop such a program later in this section.

Java has two types of files, *text files* and *binary files*. A **text file** stores a sequence of characters and is the type of file created by standard text editors like *NotePad* and *WordPad* on a Windows computer or *SimpleText* on a Macintosh. A **binary file** has a more general format that can store numbers and other data the way they are stored in the computer. In this section we will consider only text files. Binary files are considered in Chapter 11.

File Input with the File and Scanner Classes

An instance of the java.io.File class stores information that a Scanner object needs to create an input stream that is connected to the sequence of characters in a text file. A partial list of the public methods of the File class is given in the UML class diagram in Figure 4.24.



Figure 4.24: A UML class diagram of the File class with a partial list of public methods

We will need to use only the File() constructor in this section. The File instance created with the statement

```
File theFile = new File("riddles.txt");
```

will obtain and store information about the "riddles.txt" file in the same directory as the java code being executed, if such a file exists. If no such file exists, the File object stores information needed to create such a file but does not create it. In Chapter 11, we will describe how other objects can use a file object to create a file in which to write data. If we wish to create a File object that describes a file in a directory other than the one containing the Java program, we must call the constructor with a string argument that specifies the file's complete path name—that is, one that lists the sequence of directories containing the file. In any case, while we will not use it at this time, the exists() method of a File instance can be used to determine whether or not a file has been found with the specified name.

In order to read data from a file with a Scanner object we will need to use methods that were not discussed in Chapter 2. An expanded list of methods of the Scanner class is given in Figure 4.25.

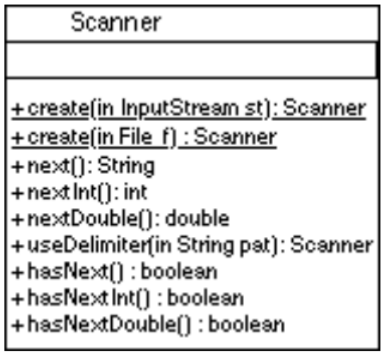


Figure 4.25: A UML class diagram of the Scanner class with an expanded list of public methods

Note the there is a Scanner() constructor with a File object as an argument. Unlike the other create() method that was used in Chapter 2, this create() throws an exception that must be handled. The following code will create a Scanner object that will be connected to an input stream that can read from a file:

```
try
{   File theFile = new File("riddles.txt");
    fileScan = new Scanner(theFile);
    fileScan = fileScan.useDelimiter("\r\n");
} catch (IOException e)
{    e.printStackTrace();
} //catch()
```

We will discuss the try-catch commands when exceptions are covered in Chapter 10. Until then, the try-catch structures can be copied exactly as above, if you wish to use a Scanner object for file input. In the code above, the useDelimiter() method has been used to set the Scanner object so that spaces can occur in strings that are read by the Scanner object. For the definition of a class to read riddles from a file, the above code belongs in a constructor method.

After we create a Scanner object connected to a file, we can make a call to nextInt(), nextDouble(), or next() method to read, respectively, an integer, real number, or string from the file. Unlike the strategy for using a Scanner object to get keyboard input, it is suggested that you test to see if there is more data in a file before reading it. This can be done with the hasNext(), hasNextInt(), and hasNextDouble() methods. These methods return the value true if there are more data in the file.

```
import java.io.*;
import java.util.Scanner;
public class RiddleFileReader
{  private Scanner fileScan; // For file input
   private Scanner kbScan;   // For keyboard input

   public RiddleFileReader(String fName)
   {   kbScan = new Scanner(System.in);
       try
       {   File theFile = new File(fName);
           fileScan = new Scanner(theFile);
           fileScan = fileScan.useDelimiter("\r\n");
       } catch (IOException e)
       {    e.printStackTrace();
       } //catch()
   } //RiddleFileReader() constructor
   public Riddle readRiddle()
   {   String ques = null;
       String ans = null;
       Riddle theRiddle = null;
       if (fileScan.hasNext())
           ques = fileScan.next();
       if (fileScan.hasNext())
       {   ans =  fileScan.next();
           theRiddle = new Riddle(ques, ans);
       } // if
       return theRiddle;
   } // readRiddle()
   public void displayRiddle(Riddle aRiddle)
   {   System.out.println(aRiddle.getQuestion());
       System.out.print("Input any letter to see answer:");
       String str = kbScan .next();  //Ignore KB input
       System.out.println(aRiddle.getAnswer());
       System.out.println();
   } // displayRiddle()
   public static void main(String[] args)
   {   RiddleFileReader rfr =
           new RiddleFileReader("riddles.txt");
       Riddle riddle = rfr.readRiddle();
       while (riddle != null)
       {   rfr.displayRiddle(riddle);
           riddle = rfr.readRiddle();
       } // while
   } //main()
}  //RiddleFileReader class
```

Figure 4.26: A program which reads riddles from a file and displays them.

The program in Figure 4.26 is the complete listing of a class that reads riddles from a file and displays them. Note that, in the body of the method readRiddles(), the statements:

```
String ques = null;
String ans = null;
Riddle theRiddle = null;
```

make explicit the fact that variables that refer to objects are assigned null as a value when they are declared. The statements:

```
if (fileScan.hasNext())
    ques = fileScan.next();
if (fileScan.hasNext())
{   ans =  fileScan.next();
    theRiddle = new Riddle(ques, ans);
}
```

will read Strings into the variables ques and ans only if the file contains lines of data for them. Otherwise the readRiddle() method will return a null value. The main() method uses this fact to terminate a while loop when it runs out of string data to assign to Riddle questions and answers. There is a separate method, displayRiddle() using a separate instance of Scanner attached to the keyboard to display the question of a riddle before the answer.

The contents of the "riddles.txt" file should be a list of riddles with each question and answer on a separate line. For example The following three riddles saved in a text file would form a good example to test the RiddleFileReader class.

```
What is black and white and red all over?
An embarrassed zebra
What is black and white and read all over?
A newspaper
What other word can be made with the letters of ALGORITHM?
LOGARITHM
```

When the main() method is executed, the user will see output in the console window that looks like:

```
What is black and white and red all over?
Input any letter to see answer: X
An embarrassed zebra

What is black and white and read all over?
Input any letter to see answer:
```

Files are covered in depth in Chapter 11. Information on writing data to a file and reading data from a file without using the Scanner class can be found in that chapter.

? Exercise 4.6.1

Modify the RiddleFileReader class to create a program NumberFileReaderthat opens a file named "numbers.txt" and reports the sum of the squares of the integers in the file. Assume that the file "numbers.txt" contains a list of integers in which each integer is on a separate line. The program should print the sum of the squares in the System.out console window. In this case, there is no need to have a method to display the data being read or a Scanner object connected to the keyboard. You will want a constructor method and a method that reads the numbers and computes the sum of squares.

Chapter Summary

abstract class

abstract interface

abstract method

AWT

binary file

buffer

command-line interface

container

control element

event-driven programming

event loop

graphical user interface (GUI)

helper method

inheritance

input operation

input stream

interface

layout manager

listener

model-view-controller (MVC) architecture

output operation

output stream

stream

Swing

text file

top-level container

user interface

wrapper class

An input operation is any action that transfers data from the user to the computer's main memory via one of the computer's input devices. An output operation is any action that transfers data from the computer's main memory to one of the computer's output devices.

The user interface is that part of the program that handles the input and output interactions between the user and the program. As an interface, it limits or constrains the manner in which the user can interact with the program.

In a command-line interface, user input is taken from the keyboard, and the program's output is displayed on some kind of console.

A buffer is a portion of main memory where input is held until it is needed by the program. Using a buffer between the keyboard and the program allows you to use the Backspace key to delete a character.

A wrapper class contains methods for converting primitive data into objects and for converting data from one type to another.

Designing appropriate prompts is an important aspect of designing a good user interface.

I/O operations must watch out for certain types of I/O exceptions.

GUI programming involves a computational model known as event-driven programming, which means that GUI programs react to events that are generated mostly by the user's interactions with elements in the GUI.

Java has two packages of GUIs, the older java.awt and the newer javax.swing.

Swing components are based on the object-oriented model-view-controller (MVC) architecture.

The extends keyword is used to specify subclass/superclass relationships in the Java class hierarchy.

A top-level container is a GUI container that cannot be added to another container; it can only have components added to it. All GUI programs must be contained in a top-level container.

There are generally three kinds of GUI components, corresponding to the three main functions of a user interface: input, output, and control.

Events are handled by special objects called listeners. A listener is a specialist that listens constantly for a certain type of event.

An interface is a special Java class that contains only methods and constants (final variables).

The following modification of the GreeterApp class is an implementation of the High Low Game:

```
%%%[basicstyle=\scriptsize]
public class HighLowApp
{ private KeyboardReader reader;
  private int secretNumber;

  public HighLowApp()
  { reader = new KeyboardReader();
    secretNumber = 1 + (int)(Math.random() * 100);
  } // HighLowApp() constructor

  public void run()
  { int userGuess = -1;
    reader.display("Guess my secret number between 1 and 100.");
    while (userGuess != secretNumber)
    {   reader.prompt("Please input your guess here > ");
      userGuess = reader.getKeyboardInteger();
      if (userGuess > secretNumber)
        reader.display("Your guess was too high.");
      if (userGuess < secretNumber)
        reader.display("Your guess was too low.");
    } // while
    reader.display("Congratulations. Your guess was correct.");
  } // run()

  public static void main(String args[])
  { HighLowApp app = new HighLowApp();
    app.run();
  } // main()
}// HighLowApp
```

The following modification of GreeterGUI eliminates the JButton.

```
%%%[basicstyle=\scriptsize]
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class GreeterGUI2 extends JFrame
                         implements ActionListener
{ private JTextArea display;
  private JTextField inField;
  private Greeter greeter;

  public GreeterGUI2(String title)
  {   greeter = new Greeter();
      buildGUI();
      setTitle(title);
      pack();
      setVisible(true);
  } // GreeterGUI2()
  private void buildGUI()
  {   Container contentPane = getContentPane();
      contentPane.setLayout(new BorderLayout());
      display = new JTextArea(10,30);
      inField = new JTextField(10);
      inField.addActionListener(this);
      JPanel inputPanel = new JPanel();
      inputPanel.add(new
              JLabel("Input your name and type enter: "));
      inputPanel.add(inField);
      contentPane.add("Center", display);
      contentPane.add("South", inputPanel);
  } // buildGUI()
  public void actionPerformed(ActionEvent e)
  {   if (e.getSource() == inField)
      {   String name = inField.getText();
          display.append(greeter.greet(name) + "\n");
      }
  } // actionPerformed()
} // GreeterGUI2
```

Java code that prints out the sum of the squares of a set of integers read from a file named "numbers.txt":

```
%%%[basicstyle=\scriptsize]
import java.io.*;
import java.util.Scanner;

public class NumberFileReader
{   private Scanner fileScan; // For file input

    public NumberFileReader(String fName)
    {   try
        {   File theFile = new File(fName);
            fileScan = new Scanner(theFile);
        } catch (IOException e)
        {    e.printStackTrace();
        } //catch()
    } //NumberFileReader()

    public void readNumbers()
    {   int num = 0;      // To store integers read
        int sum = 0:      // To store sum of squares
        while (fileScan.hasNextInt())
        {   num = fileScan.nextInt();
            sum = sum + num * num;
        } // while
        System.out.println("The sum of squares = " + sum);
    } // readNumbers()

    public static void main(String[] args)
    {   NumberFileReader nfr =
            new NumberFileReader("numbers.txt");
        nfr.readNumbers()
    } //main()
}  //NumberFileReader
```

Fill in the blanks in each of the following sentences:

=13pt

An

---

is a Java program that can be embedded in a Web page.

A method that lacks a body is an

---

method.

An

---

is like a class except that it contains only instance methods, no instance variables.

In a Java class definition a class can

---

a class and

---

an interface.

Classes and methods not defined in a program must be

---

from the Java class library.

A subclass of a class inherits that class's

instance variables and instance methods.

An object can refer to itself by using the

keyword.

The JButton, JTextField, and JComponent classes are defined in the

package.

Java GUIs utilize a form of control known as

programming.

When the user clicks on a program's JButton, an

will automatically be generated.

Two kinds of objects that generate ActionEvents are

and

.

JButtons, JTextFields, and JLabels are all subclasses of

.

The JFrame class is a subclass of

.

If java class intends to handle ActionEvents, it must implement the

interface.

When an applet is started, its

method is called automatically.

=11pt

Explain the difference between the following pairs of concepts:

Class and *interface*.

Extending a class and *instantiating an object*.

Defining a method and *implementing a method*.

A protected method and a public method.

A protected method and a private method.

An ActionEvent and an ActionListener() method.

Draw a hierarchy chart to represent the following situation. There are lots of languages in the world. English, French, Chinese, and Korean are examples of natural languages. Java, C, and C++ are examples of formal languages. French and Italian are considered romance languages, while Greek and Latin are considered classical languages.

Arrange the Java library classes mentioned in the Chapter Summary into their proper hierarchy, using the Object class as the root of the hierarchy.

Look up the documentation for the JButton class on Sun's Web site:

```
http://java.sun.com/j2se/1.5.0/docs/api/
```

List the signatures of all its constructors.

Suppose we want to set the text in our program's JTextField. What method should we use and where is this method defined? (*Hint*: Look up the documentation for JTextField. If no appropriate method is defined there, see if it is inherited from a superclass.)

Does a JApplet have an init() method? Explain.

Does a JApplet have an add() method? Explain.

Does a JButton have an init() method? Explain.

Does a JButton have an add() method? Explain.

Suppose you type the URL for a "Hello, World!" applet into your browser. Describe what happens—that is, describe the processing that takes place in order for the applet to display "Hello, World!" in your browser.

Suppose you have a program containing a JButton named button. Describe what happens, in terms of Java's event handling model, when the user clicks the button.

Java's Object class contains a public method, toString(), which returns a string that represents this object. Because every class is a subclass of Object, the toString() method can be used by any object. Show how you would invoke this method for a JButton object named button.

The JFrame that follows contains a semantic error in its SomeFrame() constructor. The error will cause the actionPerformed() method never to display "Clicked" even though the user clicks the button in the JFrame. Why? (*Hint*: Think scope!)

```
public class SomeFrame extends JFrame
                       implements ActionListener
{
     // Declare instance variables
    private JButton button;

    public JFrame()
    {
       // Instantiate the instance variable
        JButton button = new JButton("Click me");
        add(button);
        button.addActionListener(this);
    } // init()

    public void actionPerformed(ActionEvent e)
    {
        if (e.getSource() == button)
           System.out.println("Clicked");
    } // actionPerformed()
} // SomeFrame
```

What would be output by the following program?

```
public class SomeFrame2 extends JFrame
{
  // Declare instance variables
  private JButton button;
  private JTextField field;

  public SomeFrame()
  {
    // Instantiate instance variables
    button = new JButton("Click me");
    add(button);
    field = new JTextField("Field me");
    add(field);
    System.out.println(field.getText() + button.getText());
  } // init()
  public static void main(String[] args) {
     SomeFrame2 frame = new SomeFrame2();
     frame.setSize(400,400);
     frame.setVisible(true);
  }
} // SomeFrame2
}
```

Design and implement a GUI that has a JButton, a JTextField, and a JLabel and then uses the toString() method to display each object's string representation.

The JButton class inherits a setText(String s) from its AbstractButton() superclass. Using that method, design and implement a GUI that has a single button labeled initially, "The Doctor is out." Each time the button is clicked, it should toggle its label to, "The Doctor is in" and vice versa.

Design and implement a GUI that contains two JButtons, initially labeled, "Me first!" and "Me next!" Each time the user clicks either button, the labels on both buttons should be exchanged. (*Hint*: You don't need an if-else statement for this problem.)

Modify the GUI in the previous exercise so that it contains three JButtons, initially labeled "First," "Second," and "Third." Each time the user clicks one of the buttons, the labels on the buttons should be rotated. Second should get first's label, third should get second's, and first should get third's label.

Design and implement a GUI that contains a JTextField and two JButtons, initially labeled "Left" and "Right." Each time the user clicks a button, display its label in the JTextField. A JButton()'s label can be gotten with the getText() method.

You can change the size of a JFrame by using the setSize(int h, int v) method, where *h* and *v* give its horizontal and vertical dimensions pixels. Write a GUI application that contains two JButtons, labeled "Big" and "Small." Whenever the user clicks on small, set the JFrame's dimensions to 200×100, and whenever the user clicks on big, set the dimensions to 300×200.

Rewrite your solution to the previous exercise so that it uses a single button whose label is toggled appropriately each time it is clicked. Obviously, when the JButton is labeled "Big," clicking it should give the JFrame its big dimensions.

Challenge: Design and write a Java GUI application that allows the user to change the JFrame's background color to one of three choices, indicated by buttons. Like all other Java Components, JFrame's have an associated background color, which can be set by the following commands:

```
setBackground(Color.red);
setBackground(Color.yellow);
```

The setBackground() method is defined in the Component class, and 13 primary colors—black, blue, cyan, darkGray, gray, green, lightGray, , orange, pink, red, white, yellow—are defined in the java.awt.Color class.

Given the classes with the following headers

```
public class Animal ...
public class DomesticAnimal extends Animal ...
public class FarmAnimal extends DomesticAnimal...
public class HousePet extends DomesticAnimal...
public class Cow extends FarmAnimal ...
public class Goat extends FarmAnimal ...
public class DairyCow extends Cow ...
```

draw a UML class diagram representing the hierarchy created by these

Given the preceding hierarchy of classes, which of the following are legal assignment statements?

```
DairyCow dc = new FarmAnimal();
FarmAnimal fa = new Goat();
Cow c1 = new DomesticAnimal();
Cow c2 = new DairyCow();
DomesticAnimal dom = new HousePet();
```

This page titled 4.6: From the Java Library- java.io.File and file input (Optional) is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **4.6: From the Java Library- java.io.File and file input (Optional)** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

CHAPTER OVERVIEW

5: Java Data and Operators

Learning Objectives

After studying this chapter, you will

- Understand the role that data play in effective program design.
- Be able to use all of Java's primitive types and their operators.
- Appreciate the importance of information hiding.
- Be able to use class constants and class methods.
- Know how to use Java's Math and NumberFormat classes.
- Be able to perform various kinds of data conversions.

5.1: Introduction
5.2: Boolean Data and Operators
5.3: Numeric Data and Operators
5.4: From the Java Library java.lang.Math
5.5: Numeric Processing Examples
5.6: From the Java Library- java.text.NumberFormat
5.7: Character Data and Operators
5.8: Example- Character Conversions
5.9: Problem Solving = Representation + Action

This page titled 5: Java Data and Operators is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

5.1: Introduction

This chapter has two primary goals. One is to elaborate on Java's *primitive data types*, which were first introduced in Chapter 1. We will cover boolean, integer, character, and real number data types, including the various operations that you can perform on these types. We will provide examples, including several modifications of the OneRowNim class, to show typical uses of the various data types.

Our second goal is to illustrate the idea that programming is a matter of choosing an appropriate way to represent a problem as well as choosing an appropriate sequence of actions to solve the problem. Programming is a form of problem solving that can be viewed as a two-part process: *representation* and *action*.

Representation means finding a way to look at the problem. This might involve seeing the problem as closely related to a known problem or seeing that parts of the problem can be broken up into smaller problems that you already know how to solve. In terms of programming problems, representation often means choosing the right kinds of objects and structures.

Action is the process of taking well-defined steps to solve a problem. Given a particular way of representing the problem, what steps must we take to arrive at its solution?

Choosing an appropriate representation is often the key to solving a problem. For example, consider this problem: Can a chess board, with its top-left and bottom-right squares removed, be completely tiled by dominoes that cover two squares at a time?

One way to solve this problem might be to represent the chess board and dominoes as shown in Figure 5.1. If we represent the board in this way, then the actions needed to arrive at a solution involve searching for a tiling that completely covers the board. In other words, we can try one way of placing the dominoes on the board. If that doesn't work, we try another way. And so on. This process will be very time consuming, because there are millions of different ways of trying to tile the board.

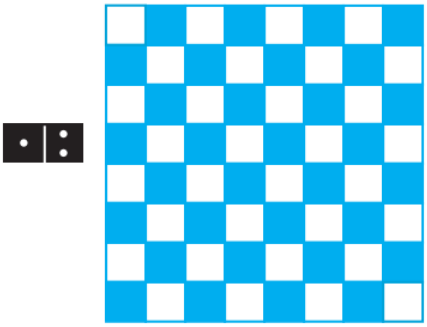


Figure 5.1: Can the chess board be tiled with dominoes?

An alternative way to represent this problem comes from seeing that the top-left and bottom-right squares of the board are both white. If you remove them, you'll have a board with 62 squares, 32 black and 30 white. Because each domino must cover one white and one black square, it is impossible to tile a board with an unequal number of black and white squares.

Thus, by representing the problem as the total number of black and white squares, the actions required to solve it involve a very simple reasoning process. This representation makes it almost trivial to find the solution. On the other hand, the *brute force* representation presented first—trying all possible combinations—made it almost impossible to solve the problem.

5.1: Introduction is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

- **5: Java Data and Operators** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

5.2: Boolean Data and Operators

As we learned in Chapter 1, the boolean type is one of Java's primitive types. For this type, there are only two possible values, true and false. The boolean type is derived from the work of British mathematician George Boole, who in the 1850s, developed an algebra to process logical expressions such as *p and q*. Such *boolean expressions* produce a value that is either *true* or *false*. Every modern programming language provides some means of representing boolean expressions.

The boolean type has several important uses. As we saw in Chapter 1, expressions of the form num == 7 and 5 < 7 have boolean values. Similarly, as we saw in Chapter 3, the boolean type is also used to represent the condition in the if statement:

```
if (boolean expression)
    statement;
```

For this reason, boolean expressions are also called *conditions*. Along these same lines, a boolean variable can be used as a *flag* or a *signal* to "remember" whether or not a certain condition holds. For example, in the following code fragment, we use isDone to mark when a particular process is completed:

```
boolean isDone = false; // Initialize the flag
...                     // Do some processing task
isDone = true;          // Set flag when the task done
...                     // Do some other stuff
if (isDone)             // Check if finished the task
...                     //  If so, do something
else
...                     //  Or, do something else
```

Boolean (or Logical) Operations

Like all the other simple data types, the boolean type consists of certain data—the values true and false—and certain actions or operations that can be performed on those data. For the boolean type there are four basic operations: AND (signified by &&), OR (signified by ||), EXCLUSIVE-OR (signified by ∧), and NOT (signified by !). These are defined in the *truth table* shown in Table 5.1. A truth tables defines boolean operators by giving their values in all possible situations. The first two columns of the table give possible boolean values for two *operands*, o1 and o2. An **operand** is a value used in an operation. Note that each row gives a different value assignment to the two operands, so that all possible assignments are represented. The remaining columns give the values that result for the various operators given the assignment of values to o1 and o2.

Table 5.1 Truth-table definitions of the boolean operators: AND (&&), OR (||), EXCLUSIVE-OR (∧), and NOT (!)

o1	o2	o1 && o2	o1 || o2	o1 ∧ o2	!o1
true	true	true	true	false	false
true	false	false	true	true	false
false	true	false	true	true	true
false	false	false	false	false	true

To see how to read this table, let's look at the AND operation, which is defined in column 3. The AND operator is a **binary operator**—that is, it requires two operands, and *o2*. If both *o1* and *o2* are true, then (*o1* && *o2*) is true (row1). If either *o1* or *o2* or both *o1* and *o2* are false, then the expression (*o1* && *o2*) is false (rows 2 and 3). The only case in which (*o1* && *o2*) is true is when both *o1* and *o2* are true (row 4).

The boolean OR operation (column 4 of Table 5.1) is also a binary operation. If both *o1* and *o2* are false, then $(o1 \mid\mid o2)$ is false (row 4). If either *o1* or *o2* or both *o1* and *o2* are true, then the expression $(o1 \mid\mid o2)$ is true (rows 1-3). Thus, the only case in which $(o1 \mid\mid o2)$ is false is when both *o1* and *o2* are false.

The boolean EXCLUSIVE-OR operation (column 5 of Table 5.1) is a binary operation, which differs from the OR operator in that it is true when either *o1* or *o2* is true (rows 2 and 3), but it is false when both *o1* and *o2* are true (row 1).

The NOT operation (the last column of Table 5.1) is a —it takes only one operand—and it simply reverses the truth value of its operand. Thus, if *o1* is true, !*o1* is false, and vice versa.

Precedence and Associativity

In order to evaluate complex boolean expressions, it is necessary to understand the order in which boolean operations are carried out by the computer. For example, what is the value of the following expression?

```
true || true && false
```

The value of this expression depends on whether we evaluate the || first or the && first. If we evaluate the || first, the expression's value will be false; if we evaluate the && first, the expression's value will be true. In the following example, we use parentheses to force one operation to be done before the other:

```
EXPRESSION                EVALUATION
----------                ----------
( true || true ) && false ==> true && false ==> false
true || ( true && false ) ==> true || false ==> true
```

As these evaluations show, we can use parentheses to force one operator or the other to be evaluated first. However, in Java, the && has higher precedence than the || operator. Therefore, the second alternative corresponds to the default interpretation that Java would apply to the expression that has no parentheses. In other words, given the expression $true \ || \ true \ \&\& \ false$, the AND operation would be evaluated before the OR operation even though the OR operator occurs first (i.e., to the left) in the unparenthesized expression.

Table 5.2 Precedence order of the boolean operators

Precedence Order	Operator	Operation
1	( )	Parentheses
2	!	NOT
3	^	EXCLUSIVE-OR
4	&&	AND
5	||	OR

As this example illustrates, the boolean operators have a built-in **precedence order** which is used to determine how boolean expressions are to be evaluated (Table 5.2). A simple method for evaluating an expression is to parenthesize the expression and then evaluate it. For example, to evaluate the complex expression

```
true || !false ^ false && true
```

we would first parenthesize it according to the precedence rules set out in Table 5.2, which gives the following expression:

```
true || (((!false) ^ false) && true)
```

We can then evaluate this fully parenthesized expression, step by step, starting at the innermost parentheses:

```
Step 1. true || ((true ^ false) && true)
Step 2. true || (true && true)
Step 3. true || true
Step 4. true
```

JAVA PROGRAMMING TIP

Parentheses. Parentheses can (and should) be used to clarify any expression that appears ambiguous or to override Java's default precedence rules.

In addition to operator precedence, it is necessary to know about anoperator's *associativity* in order to evaluate boolean expressions of the form `(op1 || op2 || op3)`. Should this expression be evaluated as `((op1 || op2) || op3)` or as `(op1 || (op2 || op3))`? The binary boolean operators all associate from left to right. Thus, the expressions

```
true ^ true ^ true   // Same as: (true ^ true) ^ true
true && true && true // Same as: (true && true) && true
true || true || true // Same as: (true || true) || true
```

would be evaluated as follows:

```
EXPRESSION               EVALUATION
----------------         -----------------
(true ^ true)  ^ true    ==> false ^ true  ==> true
(true && true)  && true  ==> true  && true ==> true
(true || true)  || true  ==> true  || true ==> true
```

Short-Circuit Evaluation

Another important feature of the boolean operators is that they utilize a form of evaluation known as *short-circuit evaluation*. In **short-circuit evaluation**, a boolean expression is evaluated from left to right, and the evaluation is discontinued as soon as the expression's value can be determined, regardless of whether it contains additional operators and operands. For example, in the expression

```
  expr1 && expr2
```

if expr1 is false, then the AND expression must be false, so expr2 need not evaluated. Similarly, in the expression

```
 expr1 || expr2
```

if expr1 is true, then the OR expression must be true, so expr2 need not evaluated.

In addition to being a more efficient form of evaluating boolean expressions, short-circuit evaluation has some practical uses. For example, we can use short-circuit evaluation to guard against null pointer exceptions. Recall from Chapter 2 that a null pointer exception results when you try to use an uninstantiated reference variable—that is, a reference variable that has not been assigned an object. For example, if we declare a OneRowNim variable without instantiating it and then try to use it, a null pointer exception will result:

```
OneRowNim game;        // Uninstantiated Reference
if (!game.gameOver())  // Null pointer exception
    game.takeSticks(num);
```

In this code, a null pointer exception results when we use game in the method call game.gameOver(). We can use short-circuit evaluation to prevent the exception from occurring:

```
if ((game != null) && (!game.gameOver())
    game.takeSticks(num);
```

In this case, because game != null is false, neither method call involving game is made, thus avoiding the exception.

Special Topic: Are We Computers?

George Boole published his seminal work, *An Investigation of the Laws of Thought*, in 1854. His achievement was in developing an algebra for logic—that is, a purely abstract and symbolic system for representing the laws of logic. Boole's was not the first attempt to explore the relationship between the human mind and an abstract system of computation. Back in 1655, Thomas Hobbes had already claimed that all thought was computation.

It is estimated that the human brain contains ($10^{12} = 10{,}000{,}000{,}000{,}000$) *neurons*, and each neuron contains something like 10,000 *dendrites*, the fibers that connect one neuron to another. Together, the neurons and dendrites make up a web of enormous complexity. Since the 1840s it has been known that the brain is primarily electrical, and by the 1940s scientists had developed a pretty good model of the electrical interactions among neurons. According to this model, neurons emit short bursts of electricity along their *axons*, which function like output wires. The bursts leap over the gap separating axons and dendrites, which function like the neurons' input wires.

In 1943, just before the first digital computers were developed, Warren McCulloch, a neurophysiologist, and Walter Pitts, a mathematician, published a paper titled, "A Logical Calculus of the Ideas Imminent in Nervous Activity." In this paper, they showed that all of the boolean operators—AND, OR, NOT, and EXCLUSIVE-OR—could be represented by the behavior of small sets of neurons. For example, they showed that three neurons could be connected together in such a way that the third neuron fired if and only if both of the other two neurons fired. This is exactly analogous to the definition of the boolean AND operator.

A few years later, when the first computers were built, many scientists and philosophers were struck by the similarity between the logic elements that made up the computer's circuits and the neuronal models that McCulloch and Pitts had developed.

The area of neural networks is a branch of artificial intelligence (one of the applied areas of computer science) and is based on this insight by McCulloch and Pitts. Researchers in this exciting and rapidly advancing field develop neural network models of various kinds of human thinking and perception.

Using Booleans in OneRowNim

Now that we have introduced the boolean data type, let's use it to improve the OneRowNim class, the latest version of which, from Chapter 3, is given in Figure 3.16. Previously we used an int variable, player, to represent who's turn it is. For a two-person game, such as One Row Nim, a boolean variable is well suited for this purpose, because it can toggle between true and false. For example, let's declare a variable, onePlaysNext, and initialize it to true, to represent the fact that player one will play first:

```
private boolean onePlaysNext = true;
```

When onePlaysNext is true, it will be player one's turn. When it is false, it will be player two's turn. Note that we are deliberately remaining uncommitted as to whether one or the other player is the computer.

Given this new variable, it is necessary to redefine the methods that had previously used the player variable. The first method that needs revision is the constructor:

```
public OneRowNim(int sticks, int starter)
{   nSticks = sticks;
    onePlaysNext = (starter == 1);
}  // OneRowNim() constructor3
```

In the constructor, the starter parameter is used with a value of 1 or 2 to set which player goes first. Note how we use an assignment statement to set onePlaysNext to true if starter equals 1; otherwise it is set to false. The assignment statement first evaluates the expression on its right hand side (starter == 1). Because this is a boolean expression, it will have a value of true or false, which will be assigned to onePlaysNext. Thus, the assignment statement is equivalent to the following if/else statement:

```
if (player == 1)
   onePlaysNext = true;
else
   onePlaysNext = false;
,
```

The remaining changes are shown in Figure 5.2. There are only two instance methods that need revision to accommodate the use of boolean variables. The takeSticks() method contains two revisions. The first uses the boolean OR operator to test whether a move is valid:

```
public boolean takeSticks(int num)
{   if (num < 1 || num > 3 || num > nSticks)
        return false;               // Error
    else                            // Valid move
    {   nSticks = nSticks - num;
        onePlaysNext = !onePlaysNext;
        return true;
    } //else
} // takeSticks()
```

It also uses the boolean NOT operator to toggle the value of onePlaysNext, to switch to the other player's turn:

```
onePlaysNext = !onePlaysNext;
```

Finally, the getPlayer() method now uses a if/else statement to return either 1 or 2 depending on who's turn it is:

```
public int getPlayer()
{   if (onePlaysNext)
        return 1;
    else return 2;
} // getPlayer()
```

```
public class OneRowNim
{    private int nSticks = 7;
     private boolean onePlaysNext = true;

     public OneRowNim()
     {
     } //OneRowNim() constructor1
     public OneRowNim(int sticks)
     {   nSticks = sticks;
     }  // OneRowNim() constructor2
     public OneRowNim(int sticks, int starter)
     {   nSticks = sticks;
         onePlaysNext = (starter == 1);
     }  // OneRowNim() constructor3
     public boolean takeSticks(int num)
     {   if (num < 1 || num > 3 || num > nSticks)
             return false;               // Error
         else                            // Valid move
         {   nSticks = nSticks - num;
             onePlaysNext = !onePlaysNext;
             return true;
         } //else
     } // takeSticks()
     public int getSticks()
     {   return nSticks;
     } {\color{cyan} // getSticks()}
     public int getPlayer()
     {   if (onePlaysNext) return 1;
         else return 2;
     } // getPlayer()
     public boolean gameOver()
     {   return (nSticks <= 0);
     } // gameOver()
     public int getWinner()
     {   if (nSticks < 1) return getPlayer();
         else return 0;  // game is not over
     } // getWinner()
     public void report()
     {   System.out.println("Number of sticks left: "
                            + getSticks());
         System.out.println("Next turn by player "
                            + getPlayer());
     }   // report()
} // OneRowNim class
```

Figure 5.2: The revised OneRowNim uses a boolean variable to keep track of who's turn it is.

This page titled 5.2: Boolean Data and Operators is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **5.2: Boolean Data and Operators** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

5.3: Numeric Data and Operators

Java has two kinds of numeric data: integers, which have no fractional part, and real numbers or *floating-point numbers*, which contain a fractional component. Java recognizes four different kinds of integers: byte, short, int, and long, which are distinguished by the number of *bits* used to represent them. A **binary digit**, or **bit**, is a 0 or a 1. (Recall that computers read instructions as series of 0s and 1s.) Java has two different kinds of real numbers, float and double, which are also distinguished by the number of bits used to represent them. See Table 5.3.

Table 5.3: Java's numeric types

Type	Bits	Range of Values
byte	8	−128 to +127
short	16	−32768 to 32767
int	32	−2147483648 to 2147483647
long	64	-2^{63} to \(2^{63}-1)
float	32	−3.40292347E +38 to +3.40292347E +38
double	64	−1.7976931348623157E +308 to +1.7976931348623157E +308

The more bits a data type has, the more values it can represent. One bit can represent two possible values, 1 and 0, which can be used to stand for true and false, respectively. Two bits can represent four possible values: 00, 01, 10, and 11; three bits can represent eight possible values: 000, 001, 010, 100, 101, 110, 011, 111. And, in general, an *n*-bit quantity can represent 2^n different values.

As illustrated in Table 5.3, the various integer types represent positive or negative whole numbers. Perhaps the most commonly used integer type in Java is the int type, which is represented in 32 bits. This means that Java can represent 2^{32} different int values, which range from $-2{,}147{,}483{,}648$ to $2{,}147{,}483{,}647$, that is, from -2^{31} to $(2^{31}-1)$. Similarly, an 8-bit integer, a byte, can represent 2^8 or 256 different values, ranging from -128 to $+127$. A 16-bit integer, a short, can represent 2^{16} different values, which range from -32768 to 32767. And a 64-bit integer, a long, can represent whole number values ranging from -2^{63} to $2^{63}-1$.

For floating-point numbers, a 32-bit float type can represent 2^{32} different real numbers and a 64-bit double value can represent 2^{64} different real numbers.

JAVA EFFECTIVE DESIGN

Platform Independence. In Java, a data type's size (number of bits) is part of its definition and, therefore, remains consistent across all platforms. In C and C++, the size of a data type is dependent on the compiler.

It is worth noting that just as model airplanes are representations of real airplanes, Java's numeric types are representations or models of the numbers we deal with in mathematics. In designing Java's data types, various trade-offs have been made in order to come up with practical implementations.

One trade-off is that the set of integers is infinite, but Java's int type can only represent a finite number of values. Similarly, Java cannot the infinite number of values that occur between, say, 1.111 and 1.112. So, certain real numbers cannot be represented at all. For example, because Java uses binary numbers to represent its numeric types, one number that cannot be represented exactly is $\frac{1}{10}$. This inability to exactly represent a value is known as **round-off error**. Being unable to represent certain values can cause problems in a program. For example, it might be difficult to represent dollars and cents accurately in a program.

Another source of problems in dealing with numeric data is due to limits in their precision. For example, a decimal number represented as a value can have a maximum of 17 *significant digits*, and a float can have a maximum 8. A significant digit is one that contributes to the number's value. If you tried to store values such as 12345.6789 or 0.123456789 in a float variable, they would be rounded off to 12345.679 and 0.12345679, respectively, causing a possible error.

JAVA DEBUGGING TIP

Significant Digits. In using numeric data, be sure the data type you choose has enough precision to represent the values your program needs.

Numeric Operations

The operations that can be done on numeric data include the standard algebraic operations: *addition* (+), *subtraction* (–), *multiplication* (*), *division* (/), as well as the *modulus* (%) operator. Note that in Java, the multiplication symbol is and not the $\times$. The arithmetic operators are binary operators, meaning that they each take two operands. Table 5.4 compares expressions involving the Java operators with their standard algebraic counterparts.

Table 5.4: The standard arithmetic operators in Java

Operation	Operator	Java	Algebra
Addition	+	$x + 2$	$x + 2$
Subtraction	–	$m - 2$	$m - 2$
Multiplication	*	$m * 2$	$2m$ or $2 \times m$
Division	/	x/y	$x \div y$ or $\frac{x}{y}$
Modulus	%	$x\%y$	x modulo y (for integers $x$and y

Although these operations should seem familiar, there are some important differences between their use in algebra and their use in a Java program. Consider the following list of expressions:

```
3 / 2     ==>  value 1     An integer result
3.0 / 2.0 ==>  value 1.5   A floating-point result
3 / 2.0   ==>  value 1.5   A floating-point result
3.0 / 2   ==>  value 1.5   A floating-point result
```

In each of these cases we are dividing the quantity 3 by the quantity 2. However, different results are obtained depending on the *type* of the operands involved. When both operands are integers, as in (3/2), the result must also be an integer. Hence, (3/2) has the

value 1, an integer. Because integers cannot have a fractional part, the 0.5 is simply discarded. Integer division (/) always gives an integer result. Thus, the value of (6/2) is 3 and the value of (7/2) is also 3. Because 3.5 is not an integer, the result of dividing 7 by 2 cannot be 3.5.

JAVA DEBUGGING TIP

Integer Division. A common source of error among beginning programmers is forgetting that integer division always gives an integer result.

On the other hand, when either operand is a real number, as in the last three cases, the result is a real number. Thus, while the same symbol (/) is used for dividing integers and real numbers, there are really two different operations involved here: *integer division* and *floating-point division*. Using the same symbol (/) for different operations (integer division and real division) is known as **operator overloading**. It is similar to method overloading, which was discussed in Chapter 3.

What if you want to keep the remainder of an integer division? Java provides the modulus operator (%), which takes two operands. The expression (7 % 5) gives the remainder after dividing 7 by 5—2 in this case. In general, the expression $(m \% n)$ (read *m* mod *n*) gives the remainder after *m* is divided by *n*. Here are several examples:

```
7 % 5   ==> 7 mod 5 equals 2
5 % 7   ==> 5 mod 7 equals 5
-7 % 5  ==> -7 mod 5 equals -2
7 % -5  ==> 7 mod -5 equals 2
```

The best way to interpret these examples is to perform long division on the operands keeping both the quotient and the remainder. For example, when you do long division on $-7 \div 5$, you get a quotient of -1 and a remainder of -2. The quotient is the value of $-7/5$ and the remainder is the value of $-7\%5$. When you do long division on $7 \div -5$, you get a quotient of -1 and a remainder of 2. The quotient is the value of $7/-5$ and the remainder is the value of $7\%-5$.

We will encounter many practical uses for the modulus operator in our programs. For a simple example, we use it when we want to determine whether an integer is even or odd. Numbers that leave a 0 remainder when divided by 2 are even:

```
if (N % 2 == 0)
    System.out.println(N + " is even");
```

More generally, we could use the mod operator to define divisibility by 3, 4, 10, or by any number.

Numeric Promotion Rules

Java is considered a *strongly typed* language because all expressions in Java, such as (3/2), have a type associated with them. In cases where one arithmetic operand is an integer and one is a floating-point number, Java *promotes* the integer into a floating-point value and performs a floating-point operation.

Promotion is a matter of converting one type to another type. For example, in the expression (5 + 4.0), the value 5 must be promoted to 5.0 before floating-point addition can be performed on (5.0 + 4.0). Generally speaking, automatic promotions such as these are allowed in Java whenever it is possible to perform the promotion *without loss of information*. Because an integer (5) does not have a fractional component, no information will be lost in promoting it to a real number (5.0). On the other hand, you cannot automatically convert a real number (5.4) to an integer (5) because that might lead to loss of information. This leads to the following rule:

JAVA LANGUAGE RULE

Integer Promotion. In an operation that contains an integer and a floating-point operand, the integer is promoted to a floating-point value before the operation is performed.

This rule is actually an instance of a more general rule, for whenever an expression involves operands of different types, some operands must be converted before the expression can be evaluated. Consider the following example:

```
byte n = 125;
short m = 32000;
n * m;
```

In this case, $n * m$ involves two different integer types, byte and short. Before evaluating this expression Java must first promote the byte to a short and carry out the operation as the multiplication of two shorts. Conversion of short to byte would not be possible because there's no way to represent the value 32000 as a byte.

It is important to note that this conversion rule applies regardless of the actual values of the operands. In applying the rule, Java looks at the operand's type, not its value. So even if *m* were assigned a value that could be represented as a byte (for example, 100), the promotion would still go from smaller to larger type. This leads to following the general rule:

JAVA LANGUAGE RULE

Type Promotion. In general, when two different types are involved in an operation, the smaller type—the one with fewer bits—is converted to the larger type before the operation is performed. To do otherwise would risk losing information.

Table 5.5 summarizes the actual promotion rules used by Java in evaluating expressions involving mixed operands. Note that the last rule implies that integer expressions involving byte or short or int are performed as int. This explains why integer literals—such as 56 or -108—are represented as int types in Java.

Table 5.5: Java promotion rules for mixed arithmetic operators. If two rules apply, choose the one that occurs first in this table.

If either operand is	The other is promoted to
double	double
float	float
long	long
byte or short	int

Operator Precedence

The built-in precedence order for arithmetic operators is shown in Table 5.6. Parenthesized expressions have highest precedence and are evaluated first. Next come the multiplication, division, and modulus operators, followed by addition and subtraction. When we have an unparenthesized expression that involves both multiplication and addition, the multiplication would be done first, even if it occurs to the right of the plus sign.

Table 5.6: Precedence order of the arithmetic operators

Precedence Order	Operator	Operation
1	( )	Parentheses
2	* / %	Multiplication, Division, Modulus
3	+ −	Addition, Subtraction

Operators at the same level in the precedence hierarchy are evaluated from left to right. For example, consider the following expression:

```
9 + 6 - 3 * 6 / 2
```

In this case, the first operation to be applied will be the multiplication (*), followed by division (/), followed by addition (+), and then finally the subtraction (–). We can use parentheses to clarify the order of evaluation. A parenthesized expression is evaluated outward from the innermost set of parentheses:

```
Step 1.  ( (9 + 6) - ((3 * 6) / 2 ) )
Step 2.  ( (9 + 6) - (18 / 2 ) )
Step 3.  ( (9 + 6) - 9 )
Step 4.  ( 15 - 9 )
Step 5.  6
```

Parentheses can (and should) always be used to clarify the order of operations in an expression. For example, addition will be performed before multiplication in the following expression:

```
(a + b) * c
```

Another reason to use parentheses is that Java's precedence and promotion rules will sometimes lead to expressions that look fine but contain subtle errors. For example, consider the following expressions:

```
System.out.println(5/3/2.0);   // 0.5
System.out.println(5/(3/2.0)); // 3.33
```

The first gives a result of 0.5, but the use of parentheses in the second gives a result of 3.33. If the second is the expected interpretation, then the parentheses here helped avoid a subtle semantic error.

JAVA PROGRAMMING TIP

Parenthesize! To avoid subtle bugs caused by Java's precedence and promotion rules, use parentheses to specify the order of evaluation in an expression.

Increment and Decrement Operators

Java provides a number of unary operators that are used to increment or decrement an integer variable. For example, the expression *k*++ uses the *increment operator* to increment the value of the integer variable *k*. The expression *k*++ is equivalent to the following Java statements:

```
int k;
k = k + 1;// Add 1 to k and assign the result back to k
```

The unary ++ operator applies to a single integer operand, in this case to the variable *k*. It increments *k*'s value by 1 and assigns the result back to *k*. It may be used either as a *preincrement* or a operator. In the expression *k*++, the operator *follows* the operand, indicating that it is being used as a *postincrement* operator. This means that the increment operation is done *after* the operand's value is used.

Contrast that with the expression ++*k* in which the ++ operator *precedes* its operand. In this case, it is used as a *preincrement* operator, which means that the increment operation is done *before* the operand's value is used.

When used in isolation, there is no practical difference between *k*++ and ++*k*. Both are equivalent to *k = k + 1*. However, when used in conjunction with other operators, there is a significant difference between preincrement and postincrement. For example, in the following code segment,

```
int j = 0, k = 0;   // Initially both j and k are 0
j = ++k;            // Final values of both j and k are 1
```

the variable *k* is incremented *before* its value is assigned to *j*. After execution of the assignment statement, *j* will equal 1 and *k* will equal 1. The sequence is equivalent to

```
int j = 0, k = 0;   // Initially both j and k are 0
k = k + 1;
j = k;              // Final values of both j and k are 1
```

However, in the following example,

```
int i = 0, k = 0; // Initially both i and k are 0
i = k++;          // Final value of i is 0 and k is 1
```

the variable *k* is incremented *after* its value is assigned to *i*. After execution of the assignment statement, *i* will have the value 0 and *k* will have the value 1. The preceding sequence is equivalent to

```
int i = 0, k = 0;   // Initially both i and k are 0
i = k;
k = k + 1;          // Final value of i is 0 and k is 1
```

In addition to the increment operator, Java also supplies the *decrement* operator −−, which can also be used in the predecrement and postdecrement forms. The expression −− *k* will first decrement *k*'s value by 1 and then use *k* in any expression in which it is embedded. The expression *k*−− will use the current value of *k* in the expression in which *k* is contained and then it will decrement *k*'s value by 1. Table 5.7 summarizes the increment and decrement operators. The unary increment and decrement operators have higher precedence than any of the binary arithmetic operators.

Table 5.7: Java's increment and decrement operators

Expression	Operation	Interpretation
$j = ++k$	Preincrement	\(k=k+1 ; j=k ;)
$j = k++$	Postincrement	$j = k; k = k + 1;$
$j = --k$	Predecrement	$k = k - 1; j = k;$
$j = k--$	Postdecrement	$j = k; k = k - 1;$

JAVA LANGUAGE RULE

Pre- and Postincrement/Decrement. If an expression like ++k or −−k occurs in an expression, k is incremented or decremented before its value is used in the rest of the expression. If an expression like k++ or k−− occurs in an expression, k is incremented or decremented after its value is used in the rest of the expression.

JAVA PROGRAMMING TIP

Increment and Decrement Operators. Because of their subtle behavior, be careful in how you use the unary increment and decrement operators. They are most appropriate and useful for incrementing and decrementing loop variables, as we'll see later.

Assignment Operators

In addition to the simple assignment operator (=), Java supplies a number of shortcut assignment operators that allow you to combine an arithmetic operation and an assignment in one operation. These operations can be used with either integer or floating-point operands. For example, the += operator allows you to combine addition and assignment into one expression. The statement

```
k += 3;
```

is equivalent to the statement

```
k = k + 3;
```

Similarly, the statement

```
r += 3.5 + 2.0 * 9.3 ;
```

is equivalent to

```
r = r + (3.5 + 2.0 * 9.3);  // i.e., r = r + 22.1;
```

As these examples illustrate, when using the += operator, the expression on its right-hand side is first evaluated and then *added* to the current value of the variable on its left-hand side.

Table 5.8 lists the other assignment operators that can be used in combination with the arithmetic operators. For each of these operations, the interpretation is the same: Evaluate the expression on the right-hand side of the operator and then perform the arithmetic operation (such as addition or multiplication) to the current value of the variable on the left of the operator.

Table 5.8: Java's assignment operators

Operator	Operation	Example	Interpretation
=	Simple assignment	$m = n;$	$m = n;$
+=	Addition then assignment	$m+ = 3;$	$m = m + 3;$
−=	Subtraction then assignment	$m- = 3;$	$m = m - 3;$
∗=	Multiplication then assignment	$m* = 3;$	$m = m * 3;$
/=	Division then assignment	$m/ = 3;$	$m = m/3;$
%=	Remainder then assignment	$m\% = 3;$	$m = m\%3;$

Relational Operators

There are several *relational* operations that can be performed on integers: <, >, <=, >=, ==, and !=. These correspond to the algebraic operators <, >, ≤, ≥, =, and ≠. Each of these operators takes two operands (integer or real) and returns a boolean result. They are defined in Table 5.9.

Table 5.9: Relational operators

Operator	Operation	Java Expression
<	Less than	5 < 10
>	Greater than	10 > 5
<=	Less than or equal to	5 <= 10
>=	Greater than or equal to	10 >= 5
==	Equal to	5 == 5
!=	Not equal to	5 != 4

Note that several of these relational operators require two symbols in Java. Thus, the familiar equals sign (=) is replaced in Java by ==. This is so the equality operator can be distinguished from the assignment operator. Also, less than or equal to (<=), greater than or equal to (>=), and not equal to (!=) require two symbols, instead of the familiar ≤, ≥, and ≠ from algebra. In each case, the two symbols should be consecutive. It is an error in Java for a space to appear between the < and = in <=.

```

```

JAVA DEBUGGING TIP

Equality and Assignment. A common semantic error among beginning programmers is to use the assignment operator (=) when the equality operator (==) is intended.

Among the relational operators, the inequalities (<, >, <=, and >=) have higher precedence than the equality operators (== and !=). In an expression that involves both kinds of operators, the inequalities would be evaluated first. Otherwise, the expression is evaluated from left to right.

Taken as a group the relational operators have lower precedence than the arithmetic operators. Therefore, in evaluating an expression that involves both arithmetic and relational operators, the arithmetic operations are done first. Table 5.10 includes all of the numeric operators introduced so far.

Table 5.10: Numeric operator precedence including relations

Precedence Order	Operator	Operation
1	( )	Parentheses
2	++ −−	Increment, decrement
3	∗ / %	Multiplication, division, modulus
4	+ −	Addition, subtraction
5	< > <= >=	Relational operators
6	== !=	Equality operators

To take an example, let us evaluate the following complex

```
9 + 6 <= 25 * 4 + 2
```

To clarify the implicit operator precedence, we first parenthesize the

```
( 9 + 6 ) <= ( (25 * 4 ) + 2 )
```

and then evaluate it step by step:

```
Step 1. ( 9 + 6 ) <= ( (25 * 4 ) + 2 )
Step 2. ( 9 + 6 ) <= ( 100 + 2 )
Step 3. 15 <= 102
Step 4. true
```

The following expression is an example of an ill-formed expression:

```
9 + 6 <= 25 * 4 == 2
```

That the expression is ill formed becomes obvious if we parenthesize it and then attempt to evaluate it:

```
Step 1. ( ( 9 + 6 )  <= ( 25 * 4 ) ) == 2
Step 2. ( 15  <= 100 ) == 2
Step 3. true == 2      // Syntax error results here
```

The problem here is that the expression true == 2 is an attempt to compare an int and a boolean value, which can't be done. As with any other binary operator, the == operator requires that both of its operands be of the same type. This is another example of Java's strong type checking.

This page titled 5.3: Numeric Data and Operators is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **5.3: Numeric Data and Operators** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

5.4: From the Java Library java.lang.Math

THE java.lang.Math class provides many common mathematical functions that will prove useful in performing numerical computations. As an element of the java.lang package, it is included implicitly in all Java programs. Table 5.11 lists some of the most commonly used Math class methods.

Table 5.11: A selection of Math class methods

Method	Description	Examples
int abs(int x) long abs(long x) float abs(float x)	Absolute value of x	if x >= 0 abs(x) is x if x < 0 abs(x) is −x
int ceil(double x)	Rounds x to the smallest integer not less than x	ceil(8.3) is 9 ceil(−8.3) is −8
int floor(double x)	Rounds x to the largest integer not greater than x	floor (8.9) is 8 floor(−8.9) is −9
double log(double x)	Natural logarithm of x	log (2.718282) is 1.0
double pow(double x, double y)	x raised to the y power (x^y)	pow(3, 4) is 81.0 pow(16.0, 0.5) is 4.0
double random()	Generates a random number in the interval [0,1)	random() is 0.5551 random() is 0.8712
long round(double x)	Rounds x to an integer	round(26.51) is 27 round (26.499) is 26
double sqrt(double x)	Square root of x	sqrt(4.0) is 2.0

All Math methods are static *class methods* and are, therefore, invoked through the class name. For example, we would calculate 2^4 as , which evaluates to 16. Similarly, we compute the square root of 225.0 as Math.sqrt(225.0), which evaluates to 15.0.

Indeed, Java's Math class cannot be instantiated and cannot be subclassed. Its basic definition is

```
public final class Math    // Final, can't subclass
{   private Math() {}      // Private, can't invoke
    ...
    public static native double sqrt(double a)
           throws ArithmeticException;
}
```

By declaring the Math class public final, we indicate that it can be accessed (public) but it cannot be extended or subclassed (final). By declaring its default constructor to be private, we prevent this class from being instantiated. The idea of a class that cannot be subclassed and cannot be instantiated may seem a little strange at first. The justification for it here is that it provides a convenient and efficient way to introduce helpful math functions into the Java language.

Defining the Math class in this way makes it easy to use its methods, because you don't have to create an instance of it. It is also a very efficient design because its methods are static elements of the java.lang package. This means they are loaded into memory at the beginning of your program's execution, and they persist in memory throughout your program's lifetime. Because Math class methods do not have to be loaded into memory each time they are invoked, their execution time will improve dramatically.

JAVA EFFECTIVE DESIGN

Static Methods. A method should be declared static if it is intended to be used whether or not there is an instance of its class.

This page titled 5.4: From the Java Library java.lang.Math is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **5.4: From the Java Library java.lang.Math** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

5.5: Numeric Processing Examples

In this section we consider several numeric programming examples. They are carefully chosen to illustrate different issues and concepts associated with processing numeric data.

Example: Rounding to Two Decimal Places

As an example of how to use Math class methods, let's consider the problem of rounding numbers. When dealing with applications that involve monetary values—dollars and cents—it is often necessary to round a calculated result to two decimal places. For example, suppose a program computes the value of a certificate of deposit (CD) to be 75.19999. Before we output this result, we would want to round it to two decimal places—to 75.20. The following algorithm can be used to accomplish this:

```
1. Multiply the number by 100, giving 7519.9999.
2. Add 0.5 to the number giving 7520.4999.
3. Drop the fractional part giving 7520
4. Divide the result by 100, giving 75.20
```

Step 3 of this algorithm can be done using the Math.floor(R) method, which rounds its real argument, *R*, to the largest integer not less than *R* (from Table 5.11). If the number to be rounded is stored in the double variable *R*, then the following expression will round R to two decimal places:

```
R = Math.floor(R * 100.0 + 0.5) / 100.0;
```

Alternatively, we could use the Math.round() method (Table 5.11). This method rounds a floating-point value to the nearest integer. For example, Math.round(65.3333) rounds to 65 and Math.round(65.6666) rounds to 66. The following expression uses it to round to two decimal places:

```
R = Math.round(100.0 * R) / 100.0;
```

Note that it is important here to divide by 100.0 and not by 100. Otherwise, the division will give an integer result and we'll lose the two decimal places.

JAVA DEBUGGING TIP

Division. Using the correct type of literal in division operations is necessary to ensure that you get the correct type of result.

Example: Converting Fahrenheit to Celsius

To illustrate some of the issues that arise in using numeric data, let's design a program that performs temperature conversions from Fahrenheit to Celsius and vice versa.

Problem Decomposition

This problem requires two classes, a Temperature class and a class. The Temperature class will perform the temperature conversions, and TemperatureUI will serve as the user interface (Fig. 5.3).

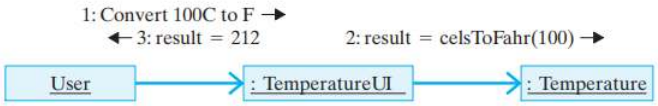


Figure 5.3: Interacting objects: The user interacts with the user interface (TemperatureUI), which interacts with the Temperature object.

Class Design: Temperature

The purpose of the Temperature class is to perform the temperature conversions. To convert a Celsius temperature to Fahrenheit or vice versa, it is not necessary to store the temperature value. Rather, a conversion method could take the Celsius (or Fahrenheit) temperature as a parameter, perform the conversion, and return the result. Therefore, the Temperature class does not need any instance variables. Note that in this respect the Temperature class resembles the Math class. Unlike OneRowNim, which stores the game's state—the number of sticks remaining and whose turn it is—the Math and Temperature classes are stateless.

Thus, following the design of the Math class, the Temperature class will have two public static methods: one to convert from Fahrenheit to Celsius and one to convert from Celsius to Fahrenheit. Recall that static methods are associated with the class rather than with its instances. Therefore, we needn't instantiate a Temperature object to use these methods. Instead, we can invoke the methods through the class itself.

The methods will use the standard conversion formulas: $F = \frac{9}{5}C + 32$ and $C = \frac{5}{9}(F - 32)$. Each of these methods should have a single parameter to store the temperature value that is being converted.

Because we want to be able to handle temperatures such as 98.6, we should use real-number data for the methods' parameters. Generally speaking, because Java represents real literals such as 98.6 as doubles, the double type is more widely used than float. Because doubles are more widely used in Java, using double wherever a floating point value is needed will cut down on the number of implicit data conversions that a program would have to perform. Therefore, each of our conversion methods should take a double parameter and return a double result. These considerations lead

JAVA PROGRAMMING TIP

Numeric Types. Java uses the int type for integer literals and double for real-number literals. When possible, using int and double for numeric variables and parameters reduces the number of implicit conversions a program would have to perform.

Implementation: Temperature

The implementation of the Temperature class is shown in Figure 5.5. Note that because celsToFahr() uses the double value temp in its calculation, it uses floating-point literals (9.0, 5.0, and 32.0) in its conversion expression. This helps to reduce the reliance on Java's built-in promotion rules, which can lead to subtle errors. For example, to the design shown in Figure 5.4,

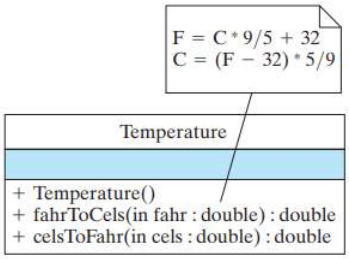


Figure 5.4: The Temperature class. Note that static elements are underlined in UML. (NEEDS REVISION)

suppose we had written what looks like an equivalent expression using integer literals:

```
return(9 / 5 * temp + 32); // Error: equals (temp + 32)
```

```
public class Temperature
{
    public Temperature() {}

    public static double fahrToCels(double temp)
    {
        return (5.0 * (temp - 32.0) / 9.0);
    }
    public static double celsToFahr(double temp)
    {
        return (9.0 * temp / 5.0 + 32.0);
    }
} // Temperature
```

Figure 5.5: The Temperature class

Because 9 divided by 5 gives the integer result 1, this expression is always equivalent to temp + 32, which is not the correct conversion formula. This kind of subtle semantic error can be avoided if you avoid mixing types wherever possible.

JAVA PROGRAMMING TIP

Don't Mix Types. You can reduce the incidence of semantic errors caused by implicit type conversions if, whenever possible, you explicitly change all the literals in an expression to the same type.

Testing and Debugging

The next question to be addressed is how should this program be tested? As always, you should test the program in a stepwise fashion. As each method is coded, you should test it both in isolation and in combination with the other methods, if

Also, you should develop appropriate *test data*. It is not enough to just plug in any values. The values you use should test for certain potential problems. For this program, the following tests are appropriate:

- Test converting 0 degrees C to 32 degrees F.
- Test converting 100 degrees C to 212 degrees F.
- Test converting 212 degrees F to 100 degrees C.
- Test converting 32 degrees F to 0 degrees C.

The first two tests use the celsToFahr() method to test the freezing point and boiling point temperatures, two boundary values for this problem. A **boundary value** is a value at the beginning or end of the range of values that a variable or calculation is meant to represent. The second pair of tests performs similar checks with the fahrToCels() method. One advantage of using these particular values is that we know what results the methods should return.

JAVA EFFECTIVE DESIGN

Test Data. Developing appropriate test data is an important part of program design. One type of test data should check the boundaries of the particular calculations you are making.

JAVA DEBUGGING TIP

Test, Test, Test! The fact that your program runs correctly on some data is no guarantee of its correctness. The more testing, and the more careful the testing you do, the better.

The TemperatureUI Class

The purpose of the TemperatureUI class is to serve as a user interface—that is, as an interface between the user and a Temperature object. It will accept a Fahrenheit or Celsius temperature from the user, pass it to one of the public methods of the Temperature object for conversion, and display the result that is returned.

As we discussed in Chapter 4, the user interface can take various forms, ranging from a command-line interface to a graphical interface. Figure 5.6 shows a design for the user interface based on the command-line interface developed in Chapter 4. The TemperatureUI uses a KeyboardReader to handle interaction with the user and uses static methods in the Temperature class to perform the temperature conversions.

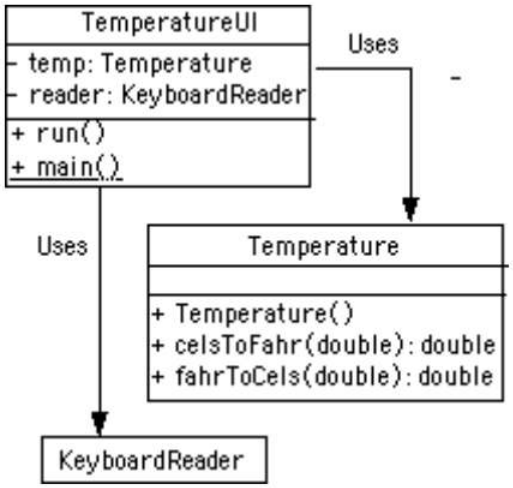


Figure 5.6: A command-line user interface.

? Exercise 5.5.1

Following the design in Figure 5.6, implement the TemperatureUI class and use it to test the methods in Temperature class. The run() method should use an **input-process-output** algorithm: Prompt the user for input, perform the necessary processing, and output the result. Note that because Temperature's conversion methods are class methods, you do not need to instantiate a Temperature object in this project. You can invoke the conversion methods directly through the Temperature class:

```
double fahr = Temperature.celsToFahr(98.6);
```

? Exercise 5.5.2

Following the design for the GUI developed in Chapter 4, implement a GUI to use for testing the Temperature class. The GUI should have the layout shown in Figure5.7

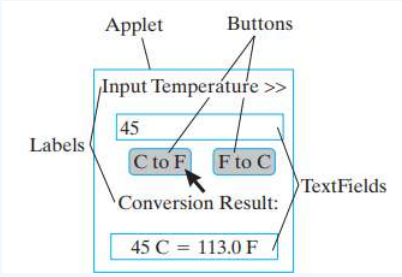


Figure 5.7: Layout design of a GUI that performs temperature conversions.

Example: Using Class Constants

As we noted in Chapter 0, in addition to instance variables, which are associated with instances (objects) of a class, Java also allows class variables, which are associated with the class itself. One of the most common uses of such variables is to define named

constants to replace literal values. A **named constant** is a variable that cannot be changed once it has been given an initial value. In this section, we use our running example, OneRowNim, to illustrate using class constants.

Recall that methods and variables that are associated with a class must be declared with the static modifier. If a variable is declared static, there is exactly one copy of that variable created no matter how many times its class is instantiated. To turn a variable into a constant, it must be declared with the final modifier. Thus, the following would be examples of a **class constants**, constant values that are associated with the class rather than with its instances:

```
public static final int PLAYER_ONE = 1;
public static final int PLAYER_TWO = 2;
public static final int MAX_PICKUP = 3;
public static final int MAX_STICKS = 7;
```

The final modifier indicates that the value of a variable cannot be changed. When final is used in a variable declaration, the variable must be assigned an initial value. After a final variable is properly declared, it is a syntax error to attempt to try to change its value. For example, given the preceding declarations, the following assignment statement would cause a compiler error:

```
PLAYER_ONE = 5;// Syntax error; PLAYER_ONE is a constant
```

Note how we use uppercase letters and underscore characters (_) in the names of constants. This is a convention that professional Java programmers follow, and its purpose is to make it easy to distinguish the constants from the variables in a program. This makes the program easier to read and understand.

JAVA PROGRAMMING TIP

Readability. To make your programs more readable, use uppercase font for constant identifiers.

Another way that named constants improve the readability of a program is by replacing the reliance on literal values. For example, for the OneRowNim class, compare the following two if conditions:

```
if (num < 1 || num > 3 || num > nSticks) ...
if (num < 1 || num > MAX_PICKUP || num > nSticks) ...
```

Clearly, the second condition is easier to read and understand. In the first condition, we have no good idea what the literal value 3 represents. In the second, we know that MAX_PICKUP represents the most sticks a player can pick up.

Thus, to make OneRowNim more readable, we should replace all occurrences of the literal value 3 with the constant MAX_PICKUP. This same principle would apply to some of the other literal values in the program. Thus, instead of using 1 and 2 to represent the two players, we could use PLAYER_ONE and PLAYER_TWO to make methods such as the following easier to read and understand:

```
public int getPlayer()
{   if (onePlaysNext)
        return PLAYER_ONE;
    else return PLAYER_TWO;
} // getPlayer()
```

JAVA PROGRAMMING TIP

Readability. To make your programs more readable, use named constants instead of literal values.

Another advantage of named constants (over literals) is that their use makes the program easier to modify and maintain. For example, suppose that we decide to change OneRowNim so that the maximum number of sticks that can be picked up is 4 instead

of 3. If we used literal values, we would have to change all occurrences of 4 that were used to represent the maximum pick up. If we used a named constant, we need only change its declaration to:

```
public static final int MAX_PICKUP = 4;
```

JAVA EFFECTIVE DESIGN

Maintainability. Constants should be used instead of literal values in a program. This will make the program easier to modify and maintain.

So far, all of the examples we have presented show why named constants (but not necessarily class constants) are useful. Not all constants are class constants. That is, not all constants are declared static. However, the idea of associating constants with a class makes good sense. In addition to saving memory resources, by creating just a single copy of the constant, constants such as MAX_STICKS and PLAYER_ONE make more conceptual sense to associate with the class itself rather than with any particular OneRowNim instance.

Class constants are used extensively in the Java class library. For example, as we saw in Chapter 2, Java's various built-in colors are represented as constants of the java.awt.Color class— Color.blue and Color.red. Similarly, java.awt.Label uses int constants to specify how a label's text should be aligned: Label.CENTER.

Another advantage of class constants is that they can be used *before* instances of the class exist. For example, a class constant (as opposed to an instance constant) may be used during object instantiation:

```
OneRowNim game = new OneRowNim(OneRowNim.MAX_STICKS);
```

Note how we use the name of the class to refer to the class constant. Of course, MAX_STICKS has to be a public variable in order to be accessible outside the class. To use MAX_STICKS as a constructor argument it has to be a class constant because at this point in the program there are no instances of OneRowNim. A new version of OneRowNim that uses class constants is shown in Figure 5.8.

```
public class OneRowNim
{   public static final int PLAYER_ONE = 1;
    public static final int PLAYER_TWO = 2;
    public static final int MAX_PICKUP = 3;
    public static final int MAX_STICKS = 11;
    public static final boolean GAME_OVER = false;

     private int nSticks = MAX_STICKS;
     private boolean onePlaysNext = true;

     public OneRowNim()
     {
     } //OneRowNim() constructor1
     public OneRowNim(int sticks)
     {   nSticks = sticks;
     }  // OneRowNim() constructor2
     public OneRowNim(int sticks, int starter)
     {   nSticks = sticks;
         onePlaysNext = (starter == PLAYER_ONE);
     }  // OneRowNim() constructor3
     public boolean takeSticks(int num)
     {   if (num < 1 || num > MAX_PICKUP || num > nSticks)
             return false;               // Error
         else                            // Valid move
         {   nSticks = nSticks - num;
             onePlaysNext = !onePlaysNext;
             return true;
         } //else
     }//takeSticks()
     public int getSticks()
     {   return nSticks;
     } //getSticks()
     public int getPlayer()
     {   if (onePlaysNext)
             return PLAYER_ONE;
         else return PLAYER_TWO;
     } //getPlayer()
     public boolean gameOver()
     {   return (nSticks <= 0);
     } // gameOver()
     public int getWinner()
     {   if (nSticks < 1)
             return getPlayer();
         else return 0;         // Game is not over
     } // getWinner()
     public String report()
     {   return ("Number of sticks left: " + getSticks()
          + "\nNext turn by player " + getPlayer() + "\n");
     }   // report()
} // OneRowNim class
```

Figure 5.8: This version of OneRowNim uses named constants.

It is important to note that Java also allows class constants to be referenced through an instance of the class. Thus, once we have instantiated game, we can refer to MAX_STICKS with either OneRowNim.MAX_STICKS or game.MAX_STICKS.

? Exercise 5.5.3

Implement a command-line interface class named KBTestOneRowNim, that uses our new version of OneRowNim. Make use of the MAX_STICKS and MAX_PICKUP in the user interface.

OBJECT-ORIENTED DESIGN: Information Hiding

The fact that our new versions of OneRowNim—we've developed two new versions in this chapter—are *backward compatible* with the previous version is due in large part to the way we have divided up its public and private elements. Because the new versions still present the same public interface, programs that use the OneRowNim class, such as the OneRowNimApp from Chapter 4 (Fig. 4.24), can continue to use the class without changing a single line of their own code. To confirm this, see the Self-Study Exercise at the end of this section.

Although we have made significant changes to the underlying representation of OneRowNim, the implementation details—its data and algorithms—are hidden from other objects. As long as OneRowNim's public interface remains compatible with the old version, changes to its private elements won't cause any inconvenience to those objects that were dependent on the old version. This ability to change the underlying implementation without affecting the outward functionality of a class is one of the great benefits of the information hiding principle.

JAVA EFFECTIVE DESIGN

Information Hiding. In designing a class, other objects should be given access just to the information they need and nothing more.

The lesson to be learned here is that the public parts of a class should be restricted to just those parts that must be accessible to other objects. Everything else should be private. Things work better, in Java programming and in the real world, when objects are designed with the principle of information hiding in mind.

? Exercise 5.5.4

To confirm that our new version of OneRowNim still works correctly with the user interfaces we developed in Chapter 4, compile and run it with OneRowNimApp (Fig. 4.24).

Example: A Winning Algorithm for One Row Nim

Now that we have access to numeric data types and operators, lets develop an algorithm that can win the One Row Nim game. Recall that in Chapter 4 we left things such that when the computer moves, it always takes 1 stick. Let's replace that strategy with a more sophisticated approach.

If you have played One Row Nim, you have probably noticed that in a game with 21 sticks, you can always win the game if you leave your opponent with 1, 5, 9, 13, 17, or 21 sticks. This is obvious for the case of 1 stick. For the case where you leave your opponent 5 sticks, no matter what the opponent does, you can make a move that leaves the other player with 1 stick. For example, if your opponent takes 1 stick, you can take 3; if your opponent takes 2, you can take 2; and, if your opponent takes 3, you can take 1. In any case, you can win the game by making the right move, if you have left your opponent with 5 sticks. The same arguments apply for the other values: 9, 13, 17, and 21.

What relationship is common to the numbers in this set? Notice that if you take the remainder after dividing each of these numbers by 4 you always get 1:

```
 1 % 4 == 1
 5 % 4 == 1
 9 % 4 == 1
13 % 4 == 1
17 % 4 == 1
21 % 4 == 1
```

Thus, we can base our winning strategy on the goal of leaving the opponent with a number of sticks, N, such that N % 4 equals 1.

To determine how many sticks to take in order to leave the opponent with N, we need to use a little algebra. Let's suppose that sticksLeft represents the number of sticks left before our turn. The first thing we have to acknowledge is that if sticksLeft % 4 == 1, then we have been left with 1, 5, 9, 13, and so on, sticks, so we cannot force a win. In that case, it doesn't matter how many sticks we pick up. Our opponent should win the game.

So, let's suppose that sticksLeft % 4 != 1, and let M be the number of sticks to pickup in order to leave our opponent with N, such that N % 4 == 1. Then we have the following two equations:

```
sticksLeft – M == N
N % 4 == 1
```

We can combine these into a single equation, which can be simplified as follows:

```
(sticksLeft – M) % 4 == 1
```

If *sticksLeft* - M leaves a remainder of 1 when divided by 4, that means that sticksLeft - M is equal some integer quotient, Q times 4 plus 1:

```
(sticksLeft – M) == (Q * 4 ) + 1
```

By adding M to both sides and subtracting 1 from both sides of this equation, we get:

```
(sticksLeft – 1 ) == (Q * 4 ) + M
```

This equation is saying that (sticksLeft - 1) % 4 == M. That is, that when you divide sticksLeft-1 by 4, you will get a remainder of M, which is the number of sticks you should pick up. Thus, to decide how many sticks to take, we want to compute:

```
M == (sticksLeft –1) % 4
```

To verify this, let's look at some examples:

sticksLeft Before	(sticksLeft −1) % 4	sticksLeft After
9	(9 −1) % 4 == 0	Illegal Move
8	(8 −1) % 4 == 3	5
7	(7 −1) % 4 == 2	5
6	(6 −1) % 4 == 1	5
5	(5 −1) % 4 == 0	Illegal Move

The examples in this table show that when we use (sticksLeft-1 % 4) to calculate our move, we always leave our opponent with a losing situation. Note that when sticksLeft equals 9 or 5, we can't apply this strategy because it would lead to an illegal move.

Let’s now convert this algorithm into Java code. In addition to incorporating our winning strategy, this move() method makes use of two important Math class methods:

```
public int move( )
```

This page titled 5.5: Numeric Processing Examples is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **5.5: Numeric Processing Examples** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

5.6: From the Java Library- java.text.NumberFormat

5.6: From the Java Library- java.text.NumberFormat is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

5.7: Character Data and Operators

5.7: Character Data and Operators is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

5.8: Example- Character Conversions

5.8: Example- Character Conversions is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

5.9: Problem Solving = Representation + Action

5.9: Problem Solving = Representation + Action is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

CHAPTER OVERVIEW

6: Control Structures

Learning Objectives

After studying this chapter, you will

- Be able to solve problems involving repetition.
- Understand the differences among various loop structures.
- Know the principles used to design effective loops.
- Improve your algorithm design skills.
- Understand the goals and principles of structured programming.

6.1: Introduction
6.2: Flow of Control- Repetition Structures
6.3: Counting Loops
6.4: Example- Car Loan
6.5: Graphics Example- Drawing a Checkerboard
6.6: Conditional Loops
6.7: Example- Computing Averages
6.8: Example- Data Validation
6.9: Principles of Loop Design
6.10: The switch Multiway Selection Structure
6.11: Object-Oriented Design- Structured Programming
6.12: Special Topic- What Can Be Computed?

This page titled 6: Control Structures is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

6.1: Introduction

As we learned in Chapter 3, a control structure is a language element that changes the flow of control of a program. Thus far, we have used the if and if/else statements to select between two or more alternate paths in a program. We have used the while-loop structure to repeat statements. And we have used method-call-and-return to invoke methods that carry out certain well-defined tasks in a program.

In this chapter we will extend our repertoire of control structures. We will introduce the *for* and *do-while* statements, both of which are used in programs that require calculations to be repeated. We will also introduce the switch statement, which will give us another way, in addition to if/else, to select from among several alternate paths in a program.

We begin by introducing the idea of a *counting loop*, which is used for repetitive tasks when you know beforehand exactly how many repetitions are necessary. This type of loop is most often implemented using a for statement.

We then distinguish two kinds of *conditional loops*, which are used for performing repetitive tasks where the number of repetitions depends on some kind of non-counting condition. These kinds of loops are usually implemented using Java's while and do-while statements. We give examples of several kinds of *loop bounds* and use them to identify several useful principles of loop design. Finally, we introduce some of the key principles of the *structured programming* approach, a disciplined design approach that preceded the object-oriented approach.

6.1: Introduction is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

- **6: Control Structures** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

6.2: Flow of Control- Repetition Structures

As we saw in Chapter 3, a **repetition structure** is a control structure that repeats a statement or sequence of statements. Many programming tasks require a repetition structure. Consider some examples.

- You're working for the National Security Agency trying to decipher secret messages intercepted from foreign spies, and you want to count the number of times a certain letter, "a," occurs in a document containing *N* characters. In this case, you would want to employ something like the following (pseudocode) algorithm:

```
initialize totalAs to 0
for each character in the document
    if the character is an 'a'
        add 1 to totalAs
return totalAs as the result
```

- You're working for a caterer who wants to number the invitations to a client's wedding, so you need to print the numbers between 1 and 5000 on the invitation cards (it's a big wedding)! In this case, you want to go through each number, 1 to 5000, and simply print it out:

```
for each number, N, from 1 to 5000
    print N on the invitation card
```

- You are helping the forest service in Alaska to track the number of black bear sightings, and you want to compute the average number of sightings per month. Suppose the user enters each month's count at the keyboard, and uses a special number, say, 9999, to signify the end of the sequence. However, 9999 should not be figured into the average. This example differs a bit from the preceding ones, because here you don't know exactly how many numbers the user will input:

```
initialize sumOfBears to 0
initialize numOfMonths to 0
repeat the following steps
    read a number from the keyboard
    if the number is NOT 9999
        add it to the sumOfBears
        add 1 to numOfMonths
until the number read is 9999
divide sumOfBears by numOfMonths giving average
return average as the result
```

We repeat the process of reading numbers and adding them to a running total "until the number read is 9999."

- Student records are stored in a file and you want to calculate Erika Wilson's current GPA. Here we need to perform a repetitive process—searching through the file for Erika Wilson's record—but again we don't know exactly how many times to repeat the process:

```
repeat the following steps
    read a record from the file
until Erika Wilson's record is read
compute Erika Wilson's GPA
return gpa as the result
```

As these examples suggest, two types of loops are used: counting loops and non-counting loops. Counting loops are used whenever you know in advance exactly how many times an action must be performed. Non-counting loops are used when the number of

repetitions depends on some condition—for example, the number of data items input from the keyboard or the input of a particular record from a file.

This page titled 6.2: Flow of Control- Repetition Structures is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **6.2: Flow of Control- Repetition Structures** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

6.3: Counting Loops

A **counting loop**, or *counter-controlled loop*, is a loop in which you know beforehand how many times it will be repeated. Among the preceding examples, the first two are counting loops.

Because you know the exact number of times the loop repeats beforehand, a counting loop can be made dependent on the value of a counter. For example, if you want to print the word “Hello” 100 times, you can use the following while structure:

```
int k = 0;
while (k < 100) {
    System.out.println("Hello");
    k++;
}
```

In this case, the counter is the variable *k*, which counts from 0 through 99—that is, it counts 100 times. Note that we start counting from 0 instead of 1. Counting from 0 is known as **zero indexing** and it is a common programming convention for counting loops. Although it doesn’t really make any practical difference in this case, later on we will use loops to process structures, such as strings and arrays, which use zero indexing. It will be easier to process these structures if our loop counter also starts at 0.

The variable *k* in this example is called a or *loop counter*. Although it is certainly possible to name the counter variable anything we like, it is customary to use single letters like *i, j*, and *k* as loop counters. The fundamental feature of a counting loop is that we must know beforehand exactly how many iterations the loop will take.

JAVA EFFECTIVE DESIGN

Loop Design. A counting loop can be used whenever you know exactly how many times a process must be repeated.

Although we can use a while-structure to code a counting loop, Java’s for statement is ideally suited for this purpose. For example, the following for statement will also print the word “Hello” 100 times:

```
for (int k = 0; k < 100; k++)
    System.out.println("Hello");
```

In fact, this for statement is equivalent to the preceding while structure.

The for statement has the following syntax:

JAVA LANGUAGE RULE

For Statement. The for statement has the following syntax:

for (initializer ; loop entry condition ; updater)

for loop body ;

The for statement begins with the keyword for, which is followed by a parenthesized list of three expressions separated by semicolons: an **initializer**, a **loop entry condition**, and an **updater**. Following the parenthesized list is the for **loop body**, which is either a single statement or a sequence of statements contained in curly brackets, `{ … }`.

The For Structure

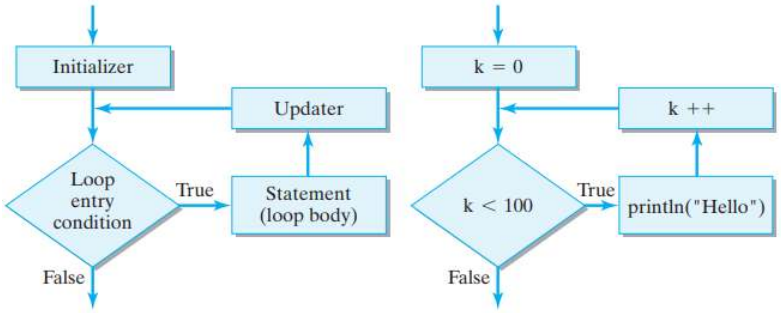


Figure 6.1: Flowchart of the for statement.

Figure 6.1 shows how the for statement works. It might be useful to compare this flowchart with the flowchart for the the while structure (Fig. 6.2), which was introduced in Chapter 3. You see that it has exactly the same structure. First, the initializer is evaluated. Thus, the initializer sets the integer variable *k* to 0. Then the loop entry condition, which must be a boolean expression, is evaluated. If it is true, the body of the loop is executed; if it is false, the body of the loop is skipped and control passes to the next statement following the for statement. The updater is evaluated after the loop body is executed. After completion of the updater, the loop entry condition is reevaluated and the loop body is either executed again or not, depending on the truth value of the loop entry condition. This process is repeated until the loop entry condition becomes false.

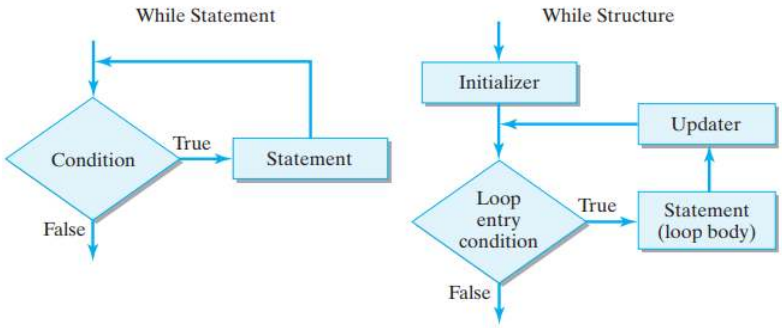


Figure 6.2: Flowchart of the while statement and while structure.

Tracing the order in which the for loop components are evaluated gives this sequence:

```
evaluate initializer
evaluate loop entry condition ==> True
execute for loop body;
evaluate updater
evaluate loop entry condition ==> True
execute for loop body;
evaluate updater
evaluate loop entry condition ==> True
execute for loop body;
evaluate updater
.
.
.
evaluate loop entry condition ==> False
```

As this trace shows, the loop entry condition controls entry to the body of the loop and will, therefore, be the last thing done before the loop terminates.

We have followed the standard convention of declaring the counter variable in the header of the for statement. This restricts the variable's *scope* to the for statement itself. It would be a syntax error to use *k* outside the scope of the for loop, as in this example:

```
for (int k = 0; k < 100; k++)
  System.out.println("Hello");
        // Syntax error, k undeclared
System.out.println("k = " + k);
```

For some problems, it might be necessary to use the loop variable outside the scope of the for statement, in which case the variable should be declared before the for statement. For example, if we want to print the value of the loop variable, k, after the loop completes, we have to declare it before the loop:

```
int k = 0;          // Declare the loop variable here
for (k = 0; k < 100; k++)
    System.out.println("Hello");
System.out.println("k = " + k); // To use it here
```

In this example, the loop will exit when k becomes 100, so "k = 100" will be printed.

Loop Bounds

A counting loop starts at some initial value and counts 0 or more iterations. A **loop bound** is a value that controls how many times a loop is repeated. A loop will repeat until its loop bound is reached. In a counting loop, the *loop entry condition* should be a boolean expression that tests whether the loop's bound has been reached. Similarly, in a counting loop, the *updater* should modify the loop counter so that it makes progress toward reaching its bound. Counting loops often increment or decrement their counter by 1, depending on whether the loop is counting forward or backward. The following method contains a countdown loop, which prints 10 9 8 7 6 5 4 3 2 1 BLASTOFF. In this case, progress toward the loop bound is made by decrementing the loop counter:

```
public void countdown() {
    for (int k = 10; k > 0; k--)
        System.out.print(k + " ");
    System.out.println("BLASTOFF");
} // countdown()
```

Note in this case that we are using **unit indexing** instead of zero indexing, because countdowns repeat, or *iterate*, from 10 down to 1, not from 10 down to 0.

Infinite Loops

If the loop bound is never reached, the loop entry condition will never become false and the loop will repeat forever. This is known as an **infinite loop**. Can you see why each of the following statements will result in an infinite loop?

```
for (int k = 0; k < 100; k--)       // Infinite loop
    System.out.println("Hello");

for (int k = 1; k != 100; k+=2)     // Infinite loop
    System.out.println("Hello");

for (int k = 98; k < 100; k = k / 2)// Infinite loop
    System.out.println("Hello");
```

In the first example, *k* starts out at 0 and is decremented on each iteration, taking on values $-1, -2, -3$, and so on, so *k* will never reach its loop bound.

In the second example, *k* starts out at 1 and is incremented by 2 on each iteration, taking on the values 3, 5, 7, and so on. Because all these values are odd, *k* will never equal 100. A much safer loop bound in this case would be \(k <\)= 100.

In the third example, *k* starts out at 98 and is halved on each iteration, taking on the values 49, 24, 12, 6, 3, 1, 0, 0, and so on, forever. Thus, it too will be stuck in an infinite loop.

Encountering an unintended infinite loop when developing a program can be very frustrating. If the program is stuck in a loop that generates output, it will be obvious that it is looping, but if no output is being generated, the computer will appear to "freeze," no longer responding to your keyboard or mouse commands. Some programming environments allow you to break out of a looping program by typing a special keyboard command such as CONTROL-C or CTRL-ALT-DELETE or CONTROL-APPLE-ESCAPE, but if that doesn't work you will have to reboot the computer, possibly causing a loss of data. The best way to avoid infinite loops is to determine that the loop's updater expression will cause the loop to eventually reach its bound.

JAVA EFFECTIVE DESIGN

Loop Design. To guard against infinite loops, make sure that the loop bound will eventually be reached.

Loop Indentation

Note how indentation is used to distinguish the loop body from both the heading and from the statement that follows the loop:

```
for (int k = 10; k > 0; k--)     // Loop heading
    System.out.print (k + " ");  //  Indent the body
System.out.println( "BLASTOFF" ) // After the loop
```

Indenting the loop body is a stylistic convention intended to make the code more readable. However, the indentation itself has no effect on how the code is interpreted by Java. Each of the following code segments would still produce the same countdown:

```
for (int k = 10; k > 0; k--)
System.out.print (k + " ");
System.out.println("BLASTOFF");

for (int k = 10; k > 0; k--) System.out.print(k +" ");
System.out.println("BLASTOFF");

for
(int k = 10; k > 0; k--)
System.out.print (k + " ");
System.out.println("BLASTOFF");
```

In each case the statement, `System.out.println("BLASTOFF")` , is not part of the for loop body and is executed only once when the loop terminates.

JAVA PROGRAMMING TIP

Loop Indentation. To make loops more readable, indent the loop body to set it off from the heading and to highlight which statement(s) will be repeated.

JAVA DEBUGGING TIP

Loop Indentation. Loop indentation has no effect on how Java interprets the loop. The loop body is determined entirely by the syntax of the for statement.

Note that up to this point the loop body has consisted of a single statement, such as a println() statement. But the loop body may consist of any Java statement, including an if or if-else statement or a compound statement, which contains a sequence of

statements enclosed within braces. Consider the following examples. The first example prints the sequence 0, 5, 10, 15, ... 95. Its loop body consists of a single if statement:

```
for (int k = 0; k < 100; k++)// Print 0 5 10...95
  if (k % 5 == 0) // Loop body: single if statement
     System.out.println("k= " + k);
```

The next example prints the lowercase letters of the alphabet. In this case, the loop counter is of type char, and it counts the letters of the alphabet. The loop body consists of a single print() statement:

```
for (char k = 'a'; k <= 'z'; k++)// Print 'a' 'b'...'z'
  System.out.print (k + " ");  // Loop body: single print()
```

The next example prints the sequence 5, 10, 15, ... 50, but it uses several statements within the loop body:

```
for (int k = 1; k <= 10; k++) {// Print 5 10 15...50
   int m = k * 5;                  // Begin body
   System.out.print (m + " ");
}                                  // End body
```

In this example, the scope of the local variable *m*, declared within the loop body, is limited to the loop body and cannot be used outside of that scope.

JAVA LANGUAGE RULE

Loop Body. The body of a for statement consists of the statement that immediately follows the for loop heading. This statement can be either a simple statement or a compound statement—a sequence of statements enclosed within braces, {. . . }.

Although braces are not required when the body of a for loop consists of a single statement, some coding styles recommend that braces should always be used for the body of a loop statement. For example, it is always correct to code the for loop as

```
for (int k = 1; k <= 10; k++) {// Print 1 2 ... 10
  System.out.print (k + " ");     // Begin body
}                                // End body
```

Another advantage of this coding style is that you can easily place additional statements in the loop body by placing them within the braces.

JAVA DEBUGGING TIP

Missing Braces. A common programming error for novices is to forget to use braces to group the statements they intend to put in the loop body. When braces are not used only the first statement after the loop heading will be iterated.

Exercise 6.3.1

Identify the syntax error in the following for loop

a.
```
for (int k = 5, k < 100, k++)
    System.out.println(k);
```

b.
```
for (int k = 0; k < 12 ; k--;)
    System.out.println(k);
```

? Exercise 6.3.2

Identify those statements that result in infinite loops.

a.
```
for (int k = 0; k < 100; k = k )
    System.out.println(k)
```

b.
```
for (int k = 1; k == 100; k = k + 2 )
    System.out.println(k);
```

c.
```
for (int k = 1; k >= 100; k = k - 2 )
    System.out.println(k);
```

? Exercise 6.3.3

Suppose you're helping your little sister learn to count by fours. Write a for loop that prints the following sequence of numbers: 1, 5, 9, 13, 17, 21, 25.

? Exercise 6.3.4

What value will *j* have when the following loop

```
for (int i = 0; i < 10; i++) {
    int j;
    j = j + 1;
}
```

Nested Loops

A **nested loop** is a structure in which one loop is contained inside the body of another loop, such as when a for loop body contains a for loop. For example, suppose you are working for Giant Auto Industries, and your boss wants you to print a table for buyers to figure the cost of buying multiple quantities of a certain part. The cost of individual parts ranges from $1 to $9. The cost of *N* items is simply the unit price times the quantity. Thus, you'll want to print something like the following table of numbers, where the prices per unit are listed in the top row, and the prices for 2, 3 and 4 units are listed in subsequent rows:

```
1  2  3  4  5  6  7  8  9
2  4  6  8  10 12 14 16 18
3  6  9  12 15 18 21 24 27
4  8  12 16 20 24 28 32 36
```

To produce this multiplication table, we could use the following nested for loops:

```
for (int row = 1; row <= 4 ; row++) { // For each of 4 rows
  for (int col = 1; col <= 9; col++) // For each of 9 columns
    System.out.print(col * row + "\t" );      // Print number
  System.out.println();              // Start a new row
} // for row
```

Note how indenting is used here to distinguish the levels of nesting and to make the code more readable. In this example, the *outer loop* controls the number of rows in the table, hence, our choice of row as its loop counter. The println() statement is executed after the *inner loop* is done iterating, which allows us to print a new row on each iteration of the outer loop. The inner loop prints the nine values in each row by printing the expression *colrow*. Obviously, the value of this expression depends on both loop variables.

Let's dissect this example a bit. How many times is the for statement in line 2 executed? The inner loop is executed once for each iteration of the outer loop. Thus, it is executed four times, which is the same number of times that line 4 is executed. How many times is the statement on line 3 executed? The body of the inner loop is executed 36 times—9 times for each execution of line 2.

Sometimes it is useful to use the loop variable of the outer loop as the bound for the inner loop. For example, consider the following pattern:

```
# # # # #
# # # #
# # #
# #
#
```

Note that the number of # symbols in each row varies inversely with the row number. In row 1, we have five symbols; in row 2 we have four; and so on down to row 5, where we have one #.

To produce this kind of two-dimensional pattern, we need two counters: one to count the row number, and one to count the number of # symbols in each row. Because we have to print each row's symbols before moving on to the next row, the outer loop will count row numbers, and the inner loop will count the symbols in each row. But note that the inner loop's bound will depend on the row number. Thus, in row 1 we want five symbols; in row 2 we want four symbols; and so on. If we let row be the row number, then in each row we want to print 6 – *row* symbols. The following table shows the relationship we want:

Row	Bound (6-row)	Number of Symbols
1	6-1	5
2	6-2	4
3	6-3	3
4	6-4	2
5	6-5	1

If we let *j* be the counter for the inner loop, then *j* will be bound by the expression 6 – *row*. This leads to the following nested loop structure:

```
for (int row = 1; row <= 5; row++) { // For each row
  for (int j = 1; j <= 6 - row; j++)// Print the row
    System.out.print('#');
  System.out.println();    // Start a new line
} // for row
```

Note that the bound of the inner loop varies according to the value of *row*, the loop counter for the outer loop.

Exercise 6.3.5

As the engineer hired to design ski jumps, write a nested for loop to print the following pattern:

```
#
# #
# # #
# # # #
# # # # #
```

This page titled 6.3: Counting Loops is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **6.3: Counting Loops** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

6.4: Example- Car Loan

Recall the self-study exercise from Chapter 5 that calculated the value of a bank CD (a) given its initial principle (p), interest rate (r), and number of years (n), using the formula $a = p(1 + r)^n$. This section explains how to use the same formula to calculate the cost of a car loan for different interest rates over various time periods.

Problem Description

For example, suppose you are planning on buying a car that costs $20,000. You find that you can get a car loan ranging anywhere from 8 to 11 percent, and you can have the loan for periods as short as two years and as long as eight years. Let's use our loop constructs to create a table to show what the car will actually cost you, including financing. In this case, *a* will represent the total cost of the car, including the financing, and *p* will represent the price tag on the car ($20,000):

```
             8%          9%          10%         11%
Year 2  $23,469.81  $23,943.82  $24,427.39  $24,920.71
Year 3  $25,424.31  $26,198.42  $26,996.07  $27,817.98
Year 4  $27,541.59  $28,665.32  $29,834.86  $31,052.09
Year 5  $29,835.19  $31,364.50  $32,972.17  $34,662.19
Year 6  $32,319.79  $34,317.85  $36,439.38  $38,692.00
Year 7  $35,011.30  $37,549.30  $40,271.19  $43,190.31
Year 8  $37,926.96  $41,085.02  $44,505.94  $48,211.60
```

Algorithm Design

The key element in this program is the nested for loop that generates the table. Because the table contains seven rows, the outer loop should iterate seven times, through the values $2, 3, \ldots 8$:

```
for (int years = 2; years <= 8; years++)
```

The inner loop should iterate through each of the interest rates, 8 through 11:

```
for (int years = 2; years <= 8; years++) {
   for (int rate = 8; rate <= 11; rate++) {
   } // for rate
} // for years
```

The financing calculation should be placed in the body of the inner loop together with a statement to print one cell (not row) of the table. Suppose the variable we use for *a* in the formula is carPriceWithLoan, and the variable we use for the actual price of the car is carPrice. Then our inner loop body is

```
carPriceWithLoan = carPrice *
     Math.pow(1 + rate/100.0/365.0, years * 365.0);
System.out.print(dollars.format(carPriceWithLoan) +"\t");
```

Note that the rate is divided by both 100.0 (to make it a percentage) and by 365.0 (for daily compounding), and the year is multiplied by 365.0 before these values are passed to the Math.pow() method. It is important here to use 100.0 and not 100 so that the resulting value is a double and not the int 0.

Implementation

The program must also contain statements to print the row and column headings. Printing the row headings should be done within the outer loop, because it must be done for each row. Printing the column headings should be done before the outer loop is entered. Finally, our program should contain code to format the dollar and cents values properly. For this we use the java.text.NumberFormat class that was described in Chapter 5. The complete program is shown in Figure 6.3.

```
import java.text.NumberFormat;   // For formatting $nn.dd or n%

public class CarLoan {
  public static void main(String args[]) {
    double carPrice = 20000;  // Car's actual price
    double carPriceWithLoan;  // Total cost of the car plus financing
                                                // Number formatting code
    NumberFormat dollars = NumberFormat.getCurrencyInstance();
    NumberFormat percent = NumberFormat.getPercentInstance();
    percent.setMaximumFractionDigits(2);
                                              // Print the table
    for (int rate = 8; rate <= 11; rate++)  // Print the column heading
      System.out.print("\t" + percent.format(rate/100.0) + "\t" );
    System.out.println();

    for (int years = 2; years <= 8; years++) {  // For years 2 through 8
      System.out.print("Year " + years + "\t"); // Print row heading
      for (int rate = 8; rate <= 11; rate++) {  // Calc and print CD value
        carPriceWithLoan = carPrice *
                      Math.pow(1 + rate / 100.0 / 365.0, years * 365.0);
        System.out.print( dollars.format(carPriceWithLoan)  + "\t");
      } // for rate
      System.out.println();                    // Start a new row
    } // for years
  } // main()
} // CarLoan
```

Figure 6.3: The CarLoan application.

This page titled 6.4: Example- Car Loan is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **6.4: Example- Car Loan** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

6.5: Graphics Example- Drawing a Checkerboard

In this section we will combine some of the graphics methods we have learned with the nested for-loop structure to draw a checkerboard with checkers on it (Fig. 6.4). For this example, we will just concentrate on drawing the checkerboard. We will not worry about providing a full checkerboard representation, of the sort we would need if we were writing a program to play the game of checkers. So, our design will not involve instance variables for whose turn it is, how many checkers each player has, where the checkers are located on the board, and other elements of the game's state. Still, our visible representation of the board should be designed so that it will eventually be useful when we do develop a checkers game in Chapter 8.

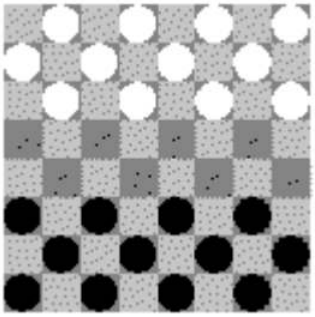


Figure 6.4: A checkerboard with checkers.

Problem Description and Specification

The specification for this problem is to develop a program that will draw a checkerboard and place upon it the checkers in their appropriate starting positions. As with many of our programs, our design will involve two classes, one which defines the user interface and the other which represents the computational object. For this problem, the computational object will be defined in the CheckerBoard class. The details of its design are described in the next section.

Because the purpose of this example is to focus on how we use loops and drawing methods, we will employ a very simple JFrame interface, whose implementation is given in Figure 6.5.

```
import java.awt.*;
import javax.swing.*;

public class CheckerBoardFrame extends JFrame {
   private CheckerBoard theBoard;

   public CheckerBoardFrame() {
       theBoard = new CheckerBoard();
   }
   public  void paint(Graphics g) {
       super.paint(g);
       theBoard.draw(g);
   } // paint()

   public static void main(String[] args){
        CheckerBoardFrame c = new CheckerBoardFrame();
        c.setSize(500,500);
        c.setVisible(true);
    }
} // CheckerBoardFrame
```

Figure 6.5: The CheckerBoardFrame class.

As shown there, the program simply creates a CheckerBoard instance in its CheckerBoardFrame() constructor, and then invokes the CheckerBoard's draw method in its paint() method. The reason we invoke the draw() method in paint() is because we need to have access to the JFrame's Graphics context, which is passed as an argument to the draw() method. Recall that the main() method is invoked when CheckerBoardFrame is run,and then the paint() methods are invoked automatically by calling the setVisible(true) method in main(). Thus, the action taken by this program is simply to draw a visual representation of the checkerboard.

Class Design: CheckerBoard

Because the program will invoke its draw() method, this method must be part of the CheckerBoard's interface. Hence, it must be declared public. The task of drawing a checkerboard involves two distinct subtasks: (1) drawing the board itself, which will involve drawing a square with smaller squares of alternating colors; and, (2) drawing the checkers on the checkerboard. A good design for the draw() method would be simply to invoke helper methods that encapsulate these two subtasks. This is good method design because it results in relatively small methods, each of which performs a very well-defined task. Let's call these methods drawBoard() and drawCheckers(), respectively. Their signatures are shown in Figure 6.6, which summarizes the design of the CheckerBoard class.



Figure 6.6: Design of the CheckerBoard class.

Before gettinginto the details of the drawBoard and drawCheckers() methods, we must first discuss CheckerBoard's several instance variables. The first two variables LEFT_X and UPPER_Y, give the absolute position of the upper left corner of the checkerboard on the JFrame's drawing panel. The SQ_SIDE variable gives the size of the checkerboard's individual squares. N_ROWS and N_COLS give the number of rows and columns in the checkerboard (typically, 8 by 8). All of these variables are integers. The final four variables, SQ_COLOR1, SQ_COLOR2, CHECKER_COLOR1, and CHECKER_COLOR2, specify the colors of the checkerboard and the checkers.

Note that the names of all the instance variables are written in uppercase letters. This is to identify them as symbolic constants—that is, as final variables whose values do not chage once they are initialized. Because their actual values are not important, we do not show them in the UML diagram and we won't discuss them here. Recall that the advantage of defining class constants, rather than sprinkling literal values throughout the program, is that they make it easy to modify the program if we decide to change the size, location, or color of the checkerboard.

Method Design

Returning now to the design of CheckerBoard's instance methods, the complete definition of the CheckerBoard class is given in Figure 6.7. Note how simple its draw() method is. As we noted earlier, in order to using Java's drawing commands, it is necessary to have a reference to a Graphics object. This is passed to the draw() method when the draw() method is invoked in the program. Because the draw() method delegates the details of the drawing algorithm to its helper methods, drawBoard() and drawCheckers(), it has to pass them a reference to the Graphics object.

```
import java.awt.*;

public class CheckerBoard {
             // Default values for a standard checkerboard
  private final int LEFT_X = 10;        // Position of left
  private final int UPPER_Y = 10;           // upper corner
  private final int SQ_SIDE = 40;     //Size of each square
  private final int N_ROWS = 8;         // Checkerboard rows
  private final int N_COLS = 8;     // Checkerboard columns
  private final Color SQ_COLOR1 = Color.lightGray; //Colors
  private final Color SQ_COLOR2 = Color.gray;  //of squares
  private final Color CHECKER_COLOR1 = Color.white;  // and
  private final Color CHECKER_COLOR2 = Color.black; //checkers

  private void drawBoard(Graphics g) {
    for(int row = 0; row < N_ROWS; row++)        // For each row
      for(int col = 0; col < N_COLS; col++) { // For each square
        if ((row + col) % 2 == 0)            // Alternate colors
          g.setColor(SQ_COLOR1);                       // Light
        else
          g.setColor(SQ_COLOR2);                     // or dark
        g.fillRect(LEFT_X+col*SQ_SIDE,
              UPPER_Y+row*SQ_SIDE,SQ_SIDE,SQ_SIDE);
      } //for
  } //drawBoard()
  private void drawCheckers(Graphics g) {   // Place checkers
    for(int row = 0; row < N_ROWS; row++)   // For each row
      for(int col = 0; col < N_COLS; col++)// For each square
        if ((row + col)%2 == 1) {// One player has top 3 rows
          if (row < 3) {
              g.setColor(CHECKER_COLOR1);
              g.fillOval(LEFT_X+col*SQ_SIDE,
                  UPPER_Y+row*SQ_SIDE,SQ_SIDE-2,SQ_SIDE-2);
          }//if
          if (row >= N_ROWS - 3) { // Other has bottom 3 rows
              g.setColor(CHECKER_COLOR2);
              g.fillOval(LEFT_X+col*SQ_SIDE,
                   UPPER_Y+row*SQ_SIDE,SQ_SIDE-2,SQ_SIDE-2);
          }//if
        }//if
  }//drawCheckers()
  public void draw(Graphics g) { // Draw board and checkers
      drawBoard(g);
      drawCheckers(g);
  }//draw()
} //CheckerBoard
```

Figure 6.7: The CheckerBoard class.

The drawBoard() method uses a nested for loop to draw an 8×8 array of rectangles of alternating colors. The loop variables, row and col, both range from 0 to 7. The expression used to determine alternating colors tests whether the sum of the row and column subscripts is even: $((row + col)\%2 == 0)$. If their sum is even, we use one color; if odd, we use the other color.

As the table below shows for a 4×4 board, the sum of a board's row and column subscripts alternates between even and odd values. Thus, in row 2 column 3, the sum of the subscripts is 5.

	0	1	2	3
0	0	1	2	3
1	1	2	3	4
2	2	3	4	5
3	3	4	5	6

To switch from one color to the other, we use the Graphics setColor() method to alternate between the two colors designated for the checkerboard, SQ_COLOR1 and SQ_COLOR2. We then use the following method call to draw the colored squares:

```
g.fillRect(LEFT_X+col*SQ_SIDE,UPPER_Y+row*SQ_SIDE,
                                        SQ_SIDE,SQ_SIDE);
```

Note how we use the loop variables, row col, together with the constants specifying the top left corner of the board (UPPER_Y and LEFT_X) and the size of the squares (SQ_SIDE) to calculate the location and size of each square. The calculation here is illustrated in Figure 6.8. The first two parameters in fillRect(left,top,width,height) specify the coordinates for the rectangle's top-left corner. These are calculated as a function of the rectangle's row and column position within the checkerboard and the rectangle's width and height, which are equal for the squares of a checkerboard.

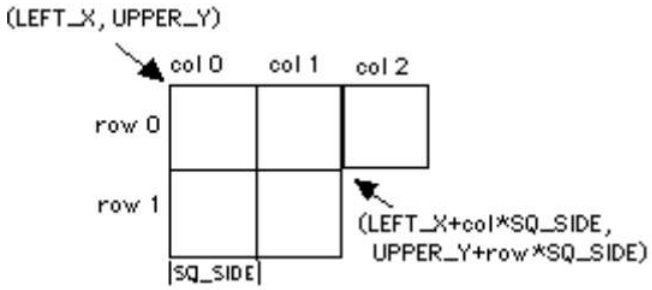


Figure 6.8: Calculating the locations of the checkerboard squares.

The drawCheckers() method also uses a nested for loop to trace through the checkerboard's rows and columns. In this case, however, we draw checkers on just the dark-colored squares—that is, those that satisfy the expression $(row + col)\%2 == 1$)—on the first three rows of each player's side of the board. So, each player's checkers initially are located in the first three rows and last three rows of the checker board:

```
if ((row + col)%2 == 1) {// One player has top 3 rows
    if (row < 3){
        g.setColor(CHECKER_COLOR1);
        g.fillOval(LEFT_X+col*SQ_SIDE,
           UPPER_Y+row*SQ_SIDE,SQ_SIDE-2,SQ_SIDE-2);
    }//if
    if (row >= N_ROWS - 3) { // Other has bottom 3 rows
        g.setColor(CHECKER_COLOR2);
        g.fillOval(LEFT_X+col*SQ_SIDE,
           UPPER_Y+row*SQ_SIDE,SQ_SIDE-2,SQ_SIDE-2);
    }//if
}//if
```

Because the checkers are circles, we use the fillOval() method to draw them. Note that the parameters for fillOval(left, top, width, height) are identical to those for fillRect(). The parameters specify an enclosing rectangle in which the oval is inscribed. In this case, of course, the enclosing rectangle is a square, which causes fillOval() to draw a circle.

Our design of the CheckerBoard class illustrates an important principle of method design. First, rather than placing all of the commands for drawing the checkerboard and the checkers into one method, we broke up this larger task into distinct subtasks. This resulted in small methods, each of which has a well defined purpose.

JAVA EFFECTIVE DESIGN

Method Decomposition. Methods should be designed to have a clear focus. If you find a method becoming too long, you should break its algorithm into subtasks and define a separate method for each subtask.

This page titled 6.5: Graphics Example- Drawing a Checkerboard is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **6.5: Graphics Example- Drawing a Checkerboard** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

6.6: Conditional Loops

Unlike the problems in the previous sections, not all loops can be coded as counting loops. Here's a problem that can't be solved by a counting loop.

Mathematicians, especially number theorists, have found that certain operations on numbers lead to interesting sequences. For example, the $3N + 1$ problem is a conjecture in number theory, which says that if N is any positive integer, then the sequence generated by the following rules will always terminate at 1.

```
Case          Operation
----          ---------
N is odd      N = 3 * N + 1
N is even     N = N / 2
```

In other words, start with any positive integer, N. If N is odd, multiply it by 3 and add 1. If N is even, divide it by 2. In either case, assign the result back to N. The conjecture states that N will eventually equal 1. For example, if N is initially 26, then the sequence generated is 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1.

The $3N + 1$ problem is an example of a noncounting loop. Because for any given N we don't know how long the $3N + 1$ sequence will be, we need a loop that terminates when the loop variable reaches a certain value, called a sentinel value—when N equals 1. This is an example of a loop that is terminated by a **sentinel bound**. With the exception of infinite loops, all loops are bounded by some condition, which is why they are sometimes referred to as **conditional loop** structures. The count and sentinel bounds are just special cases of the conditional loop structure.

The While Structure, Revisited

Consider the following pseudocode algorithm for the $3N + 1$ problem:

```
Algorithm for computing the 3N+1 sequence
    While N is not equal to 1, do: {
        Print N.
        If N is even, divide it by 2.
        If N is odd, multiply N by 3 and add 1.
    }
    Print N
```

In this structure, the body of the loop prints N and then updates N's value, using the $3N + 1$ rules. Suppose N equals 5 when this code segment begins. It will print the following sequence: 5, 16, 8, 4, 2, 1. Note that the loop body is entered as long as N is not equal to 1. So the loop entry condition in this case is $N! = 1$. Conversely, the loop will terminate when N equals 1. Also note that in this code segment the loop bound is tested before the body of the loop is executed.

We can implement this algorithm using Java's **while statement**, whose flowchart is shown in Figure 6.2:

```
while (N != 1) {         // While N not equal to 1
  System.out.print(N + " ");// Print N
  if (N % 2 == 0)         // If N is even
      N = N / 2;          //  divide it by 2
  else                    // If N is odd
      N = 3 * N + 1;      //  multiply by 3 and add 1
}
System.out.println(N);    // Print N
```

Recall that unlike the for statement, the while statement does not contain syntax for the initializer and the updater. These must be coded separately. As we pointed out in Chapter 3, the while structure (as opposed to the while statement) is a segment of code built by the programmer that satisfies the following design principle:

JAVA EFFECTIVE DESIGN

Loop Structure. A properly designed loop structure must include an initializer, a loop-entry condition, and an updater. The updater should guarantee that the loop-entry condition will eventually become false, thereby causing the loop to terminate.

The while structure has the following form:

```
InitializerStatements;          // Initializer
while (loop entry condition) { // Bound test
    Statements;                 // Loop body
    UpdaterStatements;          // Updater
}
```

As its form suggests, the while structure is designed so that on some conditions the loop body will never be executed. Because it tests for the loop bound *before* the loop body, it is possible that the loop body is never executed. We might say that it is designed to perform 0 or more iterations.

For example, going back to the $3N + 1$ problem, what if N equals 1 initially? In that case, the loop body will be skipped, because the loop entry condition is false to begin with. No iterations will be performed, and the algorithm will simply print the value 1.

The while structure would be an appropriate control structure for the following type of problem:

```
write the problems on the assignment sheet // Initializer
while there are problems on the sheet      // Bound test
    do a problem                            // Loop body
    cross it off the assignment sheet         // Updater
```

It is possible that the assignment sheet contains no homework problems to begin with. In that case, there's no work for the body of the loop to do and it should be skipped.

Exercise 6.6.1

Identify the syntax error in the following while structures:

a.
```
int k = 5;
while (k < 100) {
    System.out.println(k);
    k++
}
```

b.
```
int k = 0;
while (k < 12 ;) {
   System.out.println(k);
   k++;
}
```

Exercise 6.6.2

Determine the output and/or identify the error in each of the following while structures:

a.
```
int k = 0;
while (k < 100)
    System.out.println(k);
```

b.
```
while (k < 100) {
    System.out.println(k);
    k++;
}
```

? Exercise 6.6.3

Your younger sister is now learning how to count by sixes. Write a while loop that prints the following sequence of numbers: 0, 6, 12, 18, 24, 30, 36.

? Exercise 6.6.4

Here's another number theory problem. Start with any positive integer, N. If N is even, divide it by 2. If N is odd, subtract 1 and then divide it by 2. This will generate a sequence that is guaranteed to terminate at 0. For example, if N is initially 15, then you get the sequence: 15, 7, 3, 1, 0. Write a method that implements this sequence using a while statement.

The Do-While Structure

Here's another problem that can't be solved with a counting loop. Your father has been fretting about the bare spots on the front lawn and is considering hiring the ChemSure Lawn Service to fertilize. However, your scientifically minded younger sister wants to reassure him that at the rate the grass is dying, there will be enough to last through the summer. Using techniques she learned in biology, your sister estimates that the grass is dying at the rate of 2 percent per day. How many weeks will it take for half the lawn to disappear?

One way to solve this problem would be to keep subtracting 2 percent from the current amount of grass until the amount dipped below 50 percent, all the while counting the number of iterations required. Consider the following pseudocode algorithm:

```
Algorithm for calculating grass loss
    Initialize amtGrass to 100.0
    Initialize nDays to 0
    Repeat the following statements
        amtGrass -= amtGrass * 0.02;
        ++nDays;
    As long as amtGrass > 50.0
    Print nDays / 7
```

We begin by initializing amtGrass to 100.0, representing 100 percent. And we initialize our counter, nDays to 0. Then we repeatedly subtract 2 percent of the amount and increment the counter until the amount drops below 50 percent. In other words, in this case, we repeat the loop body as long as the amount of grass remains above 50 percent of the original. When the loop finishes, we report the number of weeks it took by dividing the number of days by 7.

The loop bound in this case is known as a **limit bound**. The loop will terminate when a certain limit has been reached—in this case, when the amount of grass dips below 50 percent of the original amount. Note that in this case the loop bound is tested *after* the loop body. This is appropriate for this problem, because we know in advance that the loop will iterate at least once. We can implement this algorithm using Java's *do-while statement*:

```
public int losingGrass(double perCentGrass) {
  double amtGrass = 100.0; // Initialize amount grass
  int nDays = 0;            // Initialize day counter
  do {                              // Repeat
    amtGrass -= amtGrass * LOSSRATE; // Update amount
    ++nDays;               //   Increment the counter
  } while (amtGrass > perCentGrass);
                  // As long as enough grass remains
  return nDays / 7;     // Return the number of weeks
} // losingGrass()
```

The **do-while statement** is a loop statement in which the loop entry condition occurs after the loop body. It has the following general form:

JAVA LANGUAGE RULE

Do-while Statement. The do-while statement has the following syntax:

do

loop body

while (loop entry condition) ;

Note, again, that unlike the for statement, the do-while statement does not contain syntax for the initializer and the updater. These must be coded separately.

To further highlight the difference between a loop statement and a loop structure, the do-while structure takes the following form:

```
InitializerStatements1;     // Initializer
do {                        // Beginning of loop body
  InitializerStatements2;   //   Another initializer
  Statements;               //   Loop body
  UpdaterStatements         //   Updater
} while (loop entry condition); // Bound test
```

Note that initializer statements may be placed before the loop body, at the beginning of the loop body, or in both places, depending on the particular problem. Like the other loop structures, updater statements occur within the body of the loop. A flowchart of the do-while structure is shown in Figure 6.9.

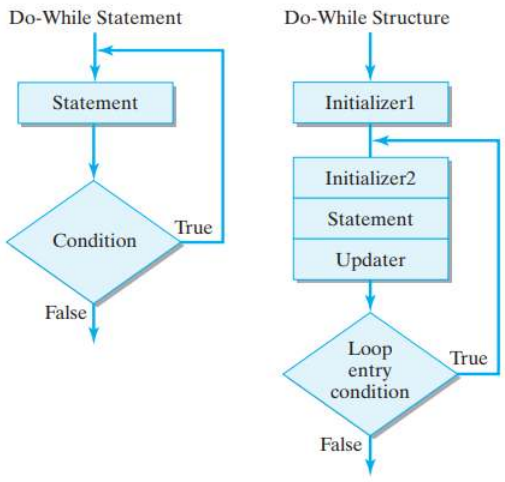


Figure 6.9: Flowchart of the do-while statement and do-while structure.

The do-while structure would be an appropriate control structure for the following type of problem:

```
do
  dial the desired telephone number  // Initializer
  if you get a busy signal
    hang up                          // Updater
while there is a busy signal         // Bound test
```

In this case, you want to perform the actions in the body of the loop at least once and possibly more than once (if you continue to receive a busy signal).

JAVA EFFECTIVE DESIGN

Do-While Loops. The do-while loop is designed for solving problems in which at least one iteration must occur.

JAVA EFFECTIVE DESIGN

While versus Do-While Structures. For problems where a noncounting loop is required, the while loop structure is more general and, therefore, preferable to the do-while structure. Use do-while only when at least one iteration must occur.

Exercise 6.6.5

Identify the syntax error in the following do-while structures:

a.
```
int k = 0;
do while (k < 100)
{    System.out.println(k);
     k++
}
```

b.
```
int k = 0;
do {
    System.out.println(k);
    k++;
} while (k < 12)
```

Exercise 6.6.6

Your sister has moved on to counting by sevens. Write a do-while loop that prints the following sequence of numbers: 1, 8, 15, 22, 29, 36, 43.

Exercise 6.6.7

As the owner of Pizza Heaven, every night at the close of business you quickly enter the price of every pizza ordered that day. You take the data from the servers' receipts. Pizzas cost $8, $10, or (the Heavenly Special) $15. You enter the data without dollar signs, and use 99 to indicate you're finished for the day. Write a Java method to input and validate a single pizza data item. If an incorrect price is entered, the program should print an error message and prompt for corrected input. Correct input is used to compute a daily total.

Exercise 6.6.8

Because the pizza prices in the previous exercise are fixed, change the method so you can save time on keyboarding. Instead of entering the price, you'll enter codes of 1, 2, or 3 (corresponding to the $8, $10, and $15 pizzas), and 0 to indicate that you're finished. Validate that the data value entered is correct and then convert it to the corresponding price before returning it.

This page titled 6.6: Conditional Loops is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **6.6: Conditional Loops** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

6.7: Example- Computing Averages

Suppose you want to compute the average of your exam grades in a course. Grades, represented as real numbers, will be input from the keyboard using our KeyboardReader class. To signify the end of the list, we will use a sentinel value—9999 or −1 or some other value that won't be confused with a legitimate grade. Because we do not know exactly how many grades will be entered, we will use a noncounting loop in this algorithm. Because there could be no grades to average, we will use a while structure so it is possible to skip the loop entirely in the case that there are no grades to average.

The algorithm should add each grade to a running total, keeping track of the number of grades entered. Thus, this algorithm requires two variables: one to keep track of the running total and the other to keep track of the count. Both should be initialized to 0. After the last grade has been entered, the total should be divided by the count to give the average. In pseudocode, the algorithm for this problem is as follows:

```
initialize runningTotal to 0         // Initialize
initialize count to 0
prompt and read the first grade      // Priming read
while the grade entered is not 9999 {// Sentinel test
    add it to the runningTotal
    add 1 to the count
    prompt and read the next grade   // Update
}
if (count > 0)           // Guard against divide by 0
    divide runningTotal by count
output the average as the result
```

Note that in this problem our loop variable, grade, is read before the loop test is made. This is known as a **priming read**. It is necessary in this case, because the loop test depends on the value that is read. Within the loop's body, the updater reads the next value for grade. This is a standard convention for coding while structures that involve input, as this problem does. Note also that we must make sure that count is not 0 before we attempt to compute the average because dividing by 0 would cause a divide-by-zero error.

Translating the pseudocode algorithm into Java raises several issues. Suppose we store each grade that is input in a double variable named grade. The loop will terminate when grade equals 9999, so its entry condition will be (grade != 9999). Because this condition uses grade, it is crucial that the grade variable be initialized before the bound test is made. This requires a priming read. By reading the first value of grade before the loop entry condition is tested, ensures that the loop will be skipped if the user happens to enter the sentinel (9999) on the very first prompt. In addition to reading the first exam score, we must initialize the variables used for the running total and the counter. Thus, for our initialization step, we get the following code:

```
double runningTotal = 0;
int count = 0;
reader.prompt("Input a grade (e.g., 85.3) " +
      "or 9999 to indicate the end of the list >> ");
double grade =
     reader.getKeyboardDouble(); // Priming input
```

Within the body of the loop we must add the grade to the running total and increment the counter. Since these variables are not tested in the loop entry condition, they will not affect the loop control. Our loop updater in this case must read the next grade. Placing the updater statement at the end of the loop body will ensure that the loop terminates immediately after the user enters the sentinel value:

```
while (grade != 9999) {    // Loop test: sentinel
  runningTotal += grade;
  count++;
  reader.prompt("Input a grade (e.g., 85.3) " +
      "or 9999 to indicate the end of the list >> ");
  grade = reader.getKeyboardDouble(); // Update:input
} // while
```

You can see that it is somewhat redundant to repeat the same statements needed to do the initializating and the updating of the grade variable. A better design would be to encapsulate these into a method and then call the method both before and within the loop. The method should take care of prompting the user, reading the input, converting it to double, and returning the input value. The method doesn't require a parameter:

```
private double promptAndRead() {
  reader.prompt("Input a grade (e.g., 85.3) " +
     "or 9999 to indicate the end of the list >> ");
  double grade = reader.getKeyboardDouble();
                          // Confirm input
  System.out.println("You input " + grade + "\n");
  return grade;
}
```

Note that we have declared this as a private method. It will be used to help us perform our task but won't be available to other objects. Such private methods are frequently called *helper methods*.

This is a much more modular design. In addition to cutting down on redundancy in our code, it makes the program easier to maintain. For example, there is only one statement to change if we decide to change the prompt message. It also makes the program easier to debug. Input errors are now localized to the promptAndRead() method.

JAVA EFFECTIVE DESIGN

Modularity. Encapsulating code in a method is a good way to avoid redundancy in a program.

JAVA DEBUGGING TIP

Localization. Encapsulating code in a method removes the need to have the same code at several locations in a program. By localizing the code in this way, you make it easier to modify and debug.

Another advantage of encapsulating the input task in a separate method is that it simplifies the task of calculating the average. This task should also be organized into a separate method:

```
public double inputAndAverageGrades() {
  double runningTotal = 0;
  int count = 0;
  double grade = promptAndRead();// Priming initializer
  while (grade != 9999) {      // Loop test: sentinel
    runningTotal += grade;
    count++;
    grade = promptAndRead();// Update: get next grade
  } // while
  if (count > 0)      // Guard against divide-by-zero
    return runningTotal / count;// Return the average
  else
    return 0;         // Special (error) return value
}
```

Note that we have declared this as a public method. This will be the method you call to calculate your course average.

Because we have decomposed the problem into its subtasks, each subtask is short and simple, making it easier to read and understand. As we saw in the checkerboard example, the use of small, clearly-focused methods is a desireable aspect of designing a program.

The complete Average.java application is shown in Figure 6.10. Its overall design is similar to application programs we designed in previous chapters. The only instance variable it uses is the KeyboardReader variable. The other variables are declared locally, within the methods. In this case, declaring them locally makes the algorithms easier to read.

```
import java.io.*;
public class Average {                              // Console I/O
  private KeyboardReader reader = new KeyboardReader();

  private double promptAndRead() {
    reader.prompt("Input a grade (e.g., 85.3) " +
           "or 9999 to indicate the end of the list >> ");
    double grade = reader.getKeyboardDouble();
    System.out.println("You input " + grade + "\n");// Confirm input
    return grade;
  }
  public double inputAndAverageGrades() {
    double runningTotal = 0;
    int count = 0;
    double grade = promptAndRead();  // Initialize: priming input
    while (grade != 9999) {          // Loop test: sentinel
      runningTotal += grade;
      count++;
      grade = promptAndRead();       // Update: get next grade
    } // while

    if (count > 0)         // Guard against divide-by-zero
      return runningTotal / count;   // Return the average
    else
      return 0;            // Special (error) return value
 }
 public static void main(String argv[]) {
   System.out.println("This program calculates average grade.");
   Average avg = new Average();
   double average = avg.inputAndAverageGrades();
   if (average == 0)                              // Error check
      System.out.println("You didn't enter any grades.");
   else
      System.out.println("Your average is " + average);
 } // main()
} // Average
```

Figure 6.10: A program to compute average grade using a while structure.

One final point about this program is to note the care taken in the design of the user interface to explain the program to the user, to prompt the user before a value is input, and to confirm the user's input after the program has read it.

JAVA EFFECTIVE DESIGN

User Interface. Whenever you ask a user for input, the user should know why you are asking and what you are asking for. Prompts should be used for this purpose. It is also a good idea to confirm that the program has received the correct input.

This page titled 6.7: Example- Computing Averages is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **6.7: Example- Computing Averages** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

6.8: Example- Data Validation

One frequent programming task is *data validation*. This task can take different forms depending on the nature of the program. One use for data validation occurs when accepting input from the user.

In the program in the preceding section, suppose the user types -10 by mistake when asked to input an exam grade. Obviously this is not a valid exam grade and should not be added to the running total. How should a program handle this task?

Because it is possible that the user may take one or more attempts to correct an input problem, we should use a do-while structure for this problem. The program should first input a number from the user. The number should then be checked for validity. If it is valid, the loop should exit and the program should continue computing the before getting the input average grade. If it is not valid, the program should print an error message and input the number again. A flowchart for this algorithm is shown in Figure 6.11.

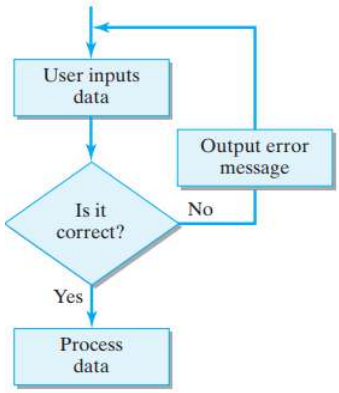


Figure 6.11: Do-while is a good structure for the data validation algorithm.

For example, suppose only numbers between 0 and 100 are considered valid. The data validation algorithm would be as follows:

```
do
 Get the next grade // Initialize: priming input
 if the grade < 0 or grade > 100 and grade != 9999
   print an error message // Error case
            // Sentinel test
while the grade < 0 or grade > 100 and grade != 9999
```

Note here that initialization and updating of the loop variable are performed by the same statement. This is acceptable because we must update the value of grade on each iteration *before* checking its validity. Note also that for this problem the loop-entry condition is also used in the if statement to check for an error. This allows us to print an appropriate error message if the user makes an input error.

Let's incorporate this data validation algorithm into the promptAndRead() method that we designed in the previous section (Fig. 6.10). The revised method will handle and validate all input and return a number between 0 and 100 to the calling method. To reflect its expanded purpose, we will change the method's name to getAndValidateGrade(), and incorporate it into a revised application, which we name Validate (Fig. 6.12).

```
import java.io.*;
public class Validate {                              // Console input
  private KeyboardReader reader = new KeyboardReader();

  private double getAndValidateGrade() {
    double grade = 0;
    do {
       reader.prompt("Input a grade (e.g., 85.3) " +
                     "or 9999 to indicate the end of the list >> ");
       grade = reader.getKeyboardDouble();
       if ((grade != 9999) && ((grade < 0) || (grade > 100))) // If error
          System.out.println("Error: grade must be between 0 and 100 \n");
       else
          System.out.println("You input " + grade + "\n");  // Confirm input
    }  while ((grade != 9999) && ((grade < 0) || (grade > 100)));
    return grade;
  }
  public double inputAndAverageGrades() {
    double runningTotal = 0;
    int count = 0;
    double grade = getAndValidateGrade();  // Initialize: priming input
    while (grade != 9999) {                 // Loop test: sentinel
       runningTotal += grade;
       count++;
       grade = getAndValidateGrade();     // Update: get next grade
    } // while

    if (count > 0)                         // Guard against divide-by-zero
       return runningTotal / count;        // Return the average
    else
       return 0;                           // Special (error) return value
  }
  public static void main(String argv[]) {
    System.out.println("This program calculates average grade."); // Explain
    Average avg = new Average();
    double average = avg.inputAndAverageGrades();
    if (average == 0)                            // Error check
       System.out.println("You didn't enter any grades.");
    else
       System.out.println("Your average is " + average);
  } // main()
} // Validate
```

Figure 6.12: A program to compute average grade using a while structure. This version validates the user's input.

This page titled 6.8: Example- Data Validation is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **6.8: Example- Data Validation** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

6.9: Principles of Loop Design

Before moving on, it will be useful to summarize the main principles involved in correctly constructing a loop.

- A *counting loop* can be used whenever you know in advance exactly how many iterations are needed. Java's *for statement* is an appropriate structure for coding a counting loop.
- A *while structure* should be used when the problem suggests that the loop body may be skipped entirely. Java's *while statement* is specially designed for the while structure.
- A *do-while structure* should be used only when a loop requires one or more iterations. Java's *do-while statement* is specially designed for the do-while structure.
- The *loop variable* is used to specify the *loop-entry condition*. It must be initialized to an appropriate initial value, and it must be updated on each iteration of the loop.
- A loop's *bound* may be a *count*, a *sentinel*, or, more generally, a *conditional bound*. It must be correctly specified in the loop-entry expression, and progress toward the bound must be made in the *updater*.
- An *infinite loop* may result if the initializer, loop-entry expression, or updater expression is not correctly specified.

The loop types are also summarized in Table 6.1.

Table 6.1: A summary of the design decisions required when coding a loop

Use	If	Java Statement
Counting loop	Number of iterations known in advance	for
While structure	Number of iterations not known Loop may not be entered at all	while
Do-while structure	Number of iterations not known Loop must be entered at least once	do-while

? Exercise 6.9.1

For each of the following problems, decide whether a counting loop structure, a while structure, or a do-while structure should be used, and write a pseudocode algorithm.

- Print the names of all visitors to your Web site.
- Validate that a number input by the user is positive.
- Change all the backslashes (\) in a Windows Web page address to the slashes (/) used in a Unix Web page address.
- Find the car with the best miles-per-gallon ratio among the cars in the *Consumer Reports* database.

This page titled 6.9: Principles of Loop Design is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **6.9: Principles of Loop Design** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

6.10: The switch Multiway Selection Structure

Another selection structure to add to our repertoire is the **switch/break structure**. It is meant to provide a shorthand way of coding the following type of multiway selection structure:

```
if (integralVar == integralValue1)
     // some statements
else if (integralVar == integralValue2)
     // some statements
else if (integralVar == integralValue3)
     // some statements
else // some statements
```

Note that each of the conditions in this case involves the equality of an integral variable and an integral value. This type of structure occurs so frequently in programs that most languages contain statements specially designed to handle it. In Java, we use a combination of the switch and break statements to implement multiway selection.

The switch is designed to select one of several actions depending on the value of some integral expression:

```
switch (integralExpression)
{  case integralValue1:
       // some statements
   case integralValue2:
       // some statements
   case integralValue3:
       // some statements
   default:
       some statements
}
```

The *integralExpression* must evaluate to a primitive integral value of type byte, short, int, char, or boolean. It may not be a long, float, double, or a class type. The *integralValues* must be literals or final variables. They serve as labels in the one or more case clauses that make up the switch statement body. The default clause is optional, but it is a good idea to include it.

A switch statement is executed according to the following rules:

- **Rule 1**. The *integralExpression* is evaluated.
- **Rule 2**. Control passes to the statements following the case label whose value equals the *integralExpression* or, if no cases apply, to the default clause.
- **Rule 3**. Beginning at the selected label or at the default, all of the statements up to the end of the switch are executed.

Consider the following example:

```
int m = 2;
switch (m)
{  case 1:
      System.out.print(" m = 1");
   case 2:
      System.out.print(" m = 2");
   case 3:
      System.out.print(" m = 3");
   default:
      System.out.print(" default case");
}
```

In this case, because \(m\) equals 2, the following output would be produced:

```
 m = 2 m = 3 default case
```

Obviously, this output does not match the following if-else multiway selection structure, which would output, simply, \(m = 2\):

```
int m = 2;
if (m == 1)
     System.out.print(" m = 1");
else if (m == 2)
     System.out.print(" m = 2");
else if (m == 3)
     System.out.print(" m = 3");
else
     System.out.print(" default case");
```

The reason for this disparity is that the switch executes *all* statements following the label that matches the value of the *integralExpression* (see again Rule 3 on the previous page).

In order to use the switch as a multiway selection, you must force it to break out of the case clause after executing that clause's statements:

```
int m = 2;
switch (m)
{  case 1:
       System.out.print(" m = 1");
       break;
   case 2:
       System.out.print(" m = 2");
       break;
   case 3:
       System.out.print(" m = 3");
       break;
   default:
       System.out.print(" default case");
}
```

In this example, the break statement causes control to pass to the end of the switch, with the effect being that one and only one case will be executed within the switch. Thus, the output of this code segment will be simply \(m = 2\), matching exactly the behavior of the multiway if-else selection structure (Fig. 6.13).

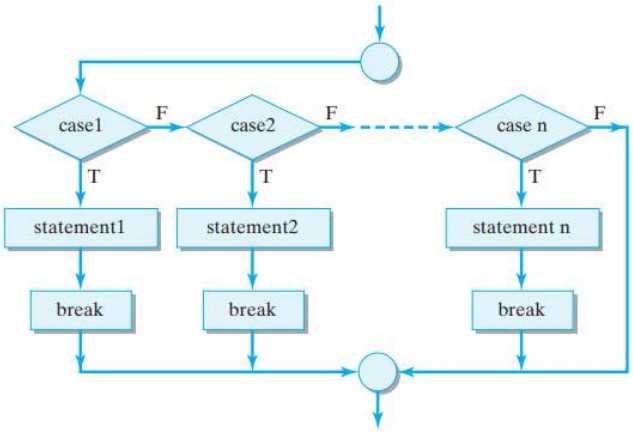


Figure 6.13: Flowchart of the multiway switch structure. Note that because of the break statement, one and only one case is executed.

JAVA PROGRAMMING TIP

Multiway Selection. A typical use for the switch statement is to use it together with break to code a multiway selection structure.

JAVA LANGUAGE RULE

break. The break statement transfers control out of its enclosing block, where a block is any sequence of statements contained within curly brackets { and }.

JAVA DEBUGGING TIP

Switch without break. A common error in coding the switch-based multiway selection is forgetting to put a break statement at the end of each clause. This may cause more than one case to be executed.

? Exercise 6.10.1

Identify any errors in the following switch structures (if there is no error, specify the output):

a.
```
(a) int k = 0;
    switch (k)
    case 0:
        System.out.println("zero");
        break;
    case 1:
        System.out.println("one");
        break;
    default:
        System.out.println("default");
        break;
```

b.
```
(b) int k = 0;
    switch (k + 1)
    {   case 0:
            System.out.println("zero");
            break;
        case 1:
            System.out.println("one");
            break;
        default:
            System.out.println("default");
            break;
    }
```

c.
```
(c) int k = 6;
    switch (k / 3.0)
    {   case 2:
            System.out.println("zero");
            break;
        case 3:
            System.out.println("one");
            break;
        default:
            System.out.println("default");
            break;
    }
```

? Exercise 6.10.2

Flavors of ice cream are represented as integers where 0 is vanilla, 1 is chocolate, and 2 is strawberry. Write a switch statement that checks an integer variable flavor and prints out the name of the ice cream flavor or prints “Error” in the default case.

? Exercise 6.10.3

Modify your solution to the previous exercise to use constants (final variables) to represent the ice cream flavors.

This page titled 6.10: The switch Multiway Selection Structure is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **6.10: The switch Multiway Selection Structure** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

6.11: Object-Oriented Design- Structured Programming

Structured programming is the practice of writing programs that are built up from a small set of predefined control structures. As an overall approach to programming, structured programming has largely been superseded by the object-oriented approach. Nevertheless, its design principles are still relevant to the design of the algorithms and methods that make up a program's objects.

The principles of structured programming seem so obvious today that it may be difficult to appreciate their importance. In the 1960s and 1970s, one of the main controls used in programs was the infamous *go to* statement, which could be used to transfer control of a program to any arbitrary location within it, and from there to any other arbitrary location, and so on. This led to incredibly complex and ill-formed programs—so called "spaghetti code"—that were almost impossible to understand and modify.

Structured programming evolved in reaction to the unstructured software development practices of the 1960s, which were fraught with budget overruns, costly delays, and failed products. One of the classic research results of that era was a 1966 paper by Boehm and Jacopini that showed that any program using go to's could be represented by an equivalent program that used a sequence of two types of controls: if/else and while structures. Another influential paper by Edgar Dikjstra ("GoTo Statement Considered Harmful") pointed out the various ways in which the go to statement could lead to impossibly complex programs.

The Pascal language, introduced by Nicklaus Wirth in 1971, was designed to promote structured programming techniques and became the language of choice within academic institutions because of its suitability as a teaching language. In Pascal, the go to was replaced with the four structures that control the flow of execution in a program (Fig. 6.14):

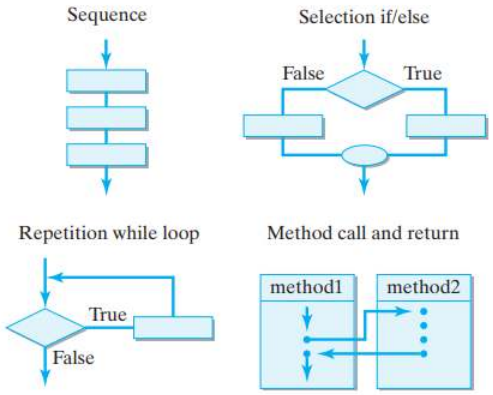


Figure 6.14: Flowcharts of the four types of control structures. Each small rectangle represents a single executable statement.

- *Sequence*—The statements in a program are executed in sequential order unless their flow is interrupted by one of the following control structures.
- *Selection*—The if, if/else, and switch statements are *branching* statements that allow choice through the forking of the control path into two or more alternatives.
- *Repetition*—The for, while, and do-while statements are *looping* statements that allow the program to repeat a sequence of statements.
- *Method Call*—Invoking a method transfers control temporarily to a named method. Control returns to the point of invocation when the method is completed.

No matter how large or small a program you write, its flow of control can be constructed as a combination of these four basic types of structures.

Preconditions and Postconditions

The Java language supplies us with a good collection of control structures, and Java's syntax constrains the way we can use them. One of the features of the four control structures is that each has a single entry point and exit (Fig. 6.14). This is an extremely important property. To grasp its importance, consider the following debugging problem:

```
k = 0;                        // 1. Unstructured code
System.out.println("k= " + k);// 2. k should equal 0 here
goto label1;                  // 3.
label2:
System.out.println("k= " + k);// 4. k should equal 1 here
```

In this example a *goto* statement is used to jump to label1, a label that marks a section of code somewhere else in the program. Suppose we're trying to determine how *k* has acquired an erroneous value and that its value is correct in line 2 of this sequence. Given the go to statement on line 3, there's no guarantee that control will ever return to the println() statement on line 4. Thus, in unstructured code it is very difficult to narrow the scope of an error to a fixed segment of code. Because the go to statement can transfer control anywhere in the program, with no guarantee of return, any segment of code can have multiple entry points and multiple exits.

Now contrast the unstructured code with the following well-structured code:

```
k = 0;                         // 1. Structured code
System.out.println("k= " + k); // 2. k should equal 0 here
someMethod();                  // 3.
System.out.println("k= " + k); // 4. k should equal 1 here
```

In this case, we can be certain that control will eventually return to line 4. If *k*'s value is erroneous on line 4, we can trace through someMethod() to find the error. Because any segment of a structured program has a single entry and exit, we can use a pair of println() statements in this way to converge on the location of the program bug.

An important implication of the single-entry/single-exit property is that we can use **preconditions** and **postconditions** to help us design and debug our code. The previous example provided a simple example: The precondition is that *k* should equal 0 on line 2, and the postcondition is that *k* should equal 1 on line 4. Figure 6.15 shows some additional examples.

```
  int k = 0;   // Precondition: k == 0
  k = 5;       // Assignment to k
// Postcondition: k == 5
```

```
  int k = 0;           // Precondition: k == 0
  while (k < 100) {    // While loop
      k = 2 * k + 2;
  }
  // Postcondition: k >= 100
```

```
/*
 * factorial(n):
 *   factorial(n) is 1 if n is 0
 *   factorial(n) is n * n-1 * n-2 * ... * 1 if n > 0
 * Precondition:  n >= 0
 * Postcondition:
 *  factorial(n) = 1 if n = 0
 *               = n * n-1 * n-2 * ... * 1 if n > 0
 */
public int factorial(int n) {
    if (n == 0)
        return 1;
    else {
        int f = 1;         // Init a temporary variable
        for (int k = n; k >= 1; k--) // For n down to 1
            f = f * k;      //   Accumulate the product
        return f;              // Return the factorial
    } // else
} // factorial()
```

Figure 6.14: Flowcharts of the four types of control structures. Each small rectangle represents a single executable statement.

In the first example, we use pre- and postconditions to define the semantics of an assignment statement. No matter what value *k* has before the assignment, the execution of the assignment (k = 5) will make the postcondition (k == 5) true.

In the second example, the postcondition follows from the semantics of the while loop. Because the loop-entry condition is `k < 100` , when the loop exits the postcondition (`k >= 100`) must be true.

The third example shows how pre- and postconditions can be used to design and document methods. The *factorial(n)* is defined for $n \geq 0$ as follows:

```
factorial(n) is 1, if n == 0
factorial(n) is n * n-1 * n-2 * ... * 1, if n > 0
```

In other words, the factorial of *N* is defined as the cumulative product of multiplying 1 times 2, times 3, and so on up to *N*. For example, if *N* is 5, then factorial(5) is 1 * 2 * 3 * 4 * 5 = 120.

Note how the factorial computation is done in the method. The variable *f*, which is used to accumulate the product, is initialized to 1. Then on each iteration of the for loop, *f* is multiplied by *k* and the product is assigned back to *f*. This is similar to the way we accumulate a sum, except in this case we are accumulating a product.

The precondition on the factorial() method represents the condition that must be true in order for the method to work correctly. Factorial is undefined for $n < 0$, so it is important that *n* be greater than or equal to 0 whenever this method is called. Given that the precondition holds, the postcondition gives a precise specification of what must be true when the method is finished.

Design: Defensive Programming

The pre- and postconditions for a method can be used to design defensive code—that is, code that guards against errors. For example, what action should factorial() take if its precondition fails to hold? In Java, the best way to handle this situation is to *throw* an IllegalArgumentException, as the following example illustrates:

```
public int factorial(int n) {
  if (n < 0)                  // Precondition failure
   throw new IllegalArgumentException("Factorial:"+ n);
  if (n == 0)
    return 1;
  else {
    int f = 1;           // Init a temporary variable
    for (int k = n; k >= 1; k--)   // For n down to 1
      f = f * k;        //     Accumulate the product
    return f;                 // Return the factorial
  }
} // factorial()
```

An **exception** is an erroneous condition (an error) that arises during the running of a program. An Exception is an object that encapsulates information about the erroneous condition. A program can *throw* an Exception, thereby stopping the program, when an erroneous condition is detected. In this example, we create a new IllegalArgumentException that would report the illegal value of *n* with something like the following error message:

```
Exception in thread "main" java.lang.IllegalArgumentException:
          Factorial: -1
   at Test.factorial(Param.java:5)
   at Test.main(Param.java:18)
```

You have undoubtedly already encountered thrown exceptions during program development. Java has an extensive hierarchy of Exceptions, which we will cover in some depth in Chapter 11. For now, however, we just note how to use the IllegalArgumentException. As its name implies, an IllegalArgumentException is used when an argument in a method call is not legal.

Rather than continuing the program with an erroreous data value, throwing an exception causes the program to stop and print an error message. Determining whether an argument is legal or illegal is an important use of the method's preconditions. The failure of the precondition in factorial() points to a problem elsewhere in the program, because it is doubtful that the program deliberately passed a negative value to factorial(). The discovery of this error should lead to modifications in that part of the program where factorial() was invoked—perhaps to some validation of the user's input:

```
int num = Integer.parseInt(textIn.getText());
if (num >= 0)    // If factorial() precondition valid
    factNum = factorial(num);   // Compute factorial
else
    System.out.println("Error");// Report input error}
```

This would be the traditional way to handle this kind of error.

Using and Postconditions

The use of preconditions and postconditions in the ways we've described can help improve a program's design at several distinct stages of its development:

- Design stage: Using pre- and postconditions in design helps to clarify the design and provides a precise measure of correctness.
- Implementation and testing stage: Test data can be designed to demonstrate that the preconditions and postconditions hold for any method or code segment.
- Documentation stage: Using pre- and postconditions to document the program makes the program more readable and easier to modify and maintain.
- Debugging stage: Using the pre- and postconditions provides precise criteria that can be used to isolate and locate bugs. A method is incorrect if its precondition is true and its postcondition is false. A method is improperly invoked if its precondition is

false.

Like other programming skills and techniques, learning how to use pre- and postconditions effectively requires practice. One way to develop these skills is to incorporate pre- and postconditions into the documentation of the methods you write for laboratories and programming exercises. Appendix A provides guidelines on how to incorporate pre- and postconditions into your program's documentation. However, it would be a mistake to get in the habit of leaving the identification of pre- and postconditions to the documentation stage. The method's documentation, including its pre- and postconditions, should be developed during the design stage and should play a role in all aspects of program development.

Effective Program Design

What we're really saying here is that using pre- and postconditions forces you to analyze your program's logic. It is not enough to know that a single isolated statement within a program works correctly at the present time. You have to ask yourself: Will it continue to work if you change some other part of the program? Will other parts of the program continue to work if you revise it? No matter how clever you are, it is not possible to keep an entire model of a good-sized program in your head at one time. It is always necessary to focus on a few essential details and leave aside certain others. Ideally, what you hope is that the details you've left aside for the moment aren't the cause of the current bug you're trying to fix. Using pre- and postconditions can help you determine the correctness of the details you choose to set aside.

JAVA EFFECTIVE DESIGN

Pre- and Postconditions. Pre- and postconditions are an effective way of analyzing the logic of your program's loops and methods. They should be identified at the earliest stages of design and development. They should play a role in the testing and debugging of the program. Finally, they should be included, in a systematic way, in the program's documentation.

JAVA PROGRAMMING TIP

Develop your program's documentation at the same time that you develop its code and include the pre- and postconditions in the documentation.

As the programs you write become longer and more complex, the chances that they contain serious errors increase dramatically. There's no real way to avoid this complexity. The only hope is to try to manage it. In addition to analyzing your program's structure, another important aspect of program design is the attempt to reduce its complexity.

JAVA EFFECTIVE DESIGN

Reducing Complexity. Design your programs with an aim toward reducing their complexity.

Perhaps the best way to reduce complexity is to build your programs using a small collection of standard structures and techniques. The basic control structures (Fig. 6.14) help reduce the potential complexity of a program by constraining the kinds of branching and looping structures that can be built. The control structures help to manage the complexity of your program's algorithms. In the same way, the following practices can help reduce and manage the complexity in a program.

JAVA PROGRAMMING TIP

Standard Techniques. Acquire and use standard programming techniques for standard programming problems. For example, using a temporary variable to swap the values of two variables is a standard technique.

JAVA PROGRAMMING TIP

Encapsulation. Use methods wherever appropriate in your own code to encapsulate important sections of code and thereby reduce complexity.

JAVA PROGRAMMING TIP

Code Reuse. Instead of reinventing the wheel, use library classes and methods whenever possible. These have been carefully designed by experienced programmers. Library code has been subjected to extensive testing.

Exercise 6.11.1

Identify the pre- and postconditions on *j* and *k* where indicated in the following code segment:

```
int j = 0; k = 5;
do {
    if (k % 5 == 0)  {
                        // Precondition
        j += k;
        k--;
    }
    else k *= k;
} while (j <= k);
                       // Postcondition
```

Exercise 6.11.2

Identify the pre- and postconditions for the following method, which computes x^n for $n \geq 0$:

```
public double power(double x, int n) {
    double pow = 1;
    for (int k = 1; k <= n; k++)
        pow = pow * x;
    return pow;
} // power()
```

This page titled 6.11: Object-Oriented Design- Structured Programming is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **6.11: Object-Oriented Design- Structured Programming** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

6.12: Special Topic- What Can Be Computed?

Did you ever wonder whether there are problems that cannot be solved by a computer, no matter what kind of control structures are used? Well, back in 1939, in his seminal paper titled “On Computable Numbers,” Alan Turing proved that indeed there are an infinite number of unsolvable problems. Prior to this, mathematicians and logicians thought all problems could be solved. So Turing’s proof was quite a blow!

To help him prove this point, Turing defined an abstract computer, which has come to be known as a Turing machine. A Turing machine has an alphabet of symbols; a read/write head; an infinitely long tape on which the read/write head can write symbols, and from which it can also read symbols; and a control unit, which controls the movement and action of the read/write head. Note that the Turing machine elements correspond to key components of a real computer—although Turing invented this concept a decade before the first computers were developed. The read/write head corresponds to a computer’s central processing unit (CPU). The tape corresponds to the computer’s memory. And the control unit corresponds to the computer program.

A Turing machine represents a purely abstract concept of computation. It represents the pure idea of an algorithmic solution to a problem. Equipped with this concept, Turing was able to prove that there are unsolvable problems—that is, problems for which no algorithm can arrive at a solution.

One such problem is the *halting problem*. This problem asks whether an algorithm can be devised to determine whether an arbitrary program will eventually halt. If there were such an algorithm, it could be used to detect programs that contain infinite loops, a service that might be really helpful in an introductory computing lab, among other places! But, alas, there can be no such algorithm.

Here’s an outline of a proof that shows that the halting problem is unsolvable. (This particular version of the proof was suggested by J. Glenn Brookshear in *Computer Science: An Overview*, Benjamin-Cummings, 1985.)

Suppose you had a program, *P*, that solves the halting problem. That is, whenever *P* is given a self-halting program, it sets a variable *isTerminating* to true, and otherwise it sets *isTerminating* to false. Now let’s create a new version of *P*, named $P\prime$, which is identical to *P* except that right after where *P* sets *isTerminating* to true or false, $P\prime$ contains the following loop:

```
while (isTerminating == true);  // Infinite if isTerminating true
```

In other words, if the input to $P\prime$ is a self-terminating program, then $P\prime$ will enter an infinite loop and it won’t terminate. Otherwise, if a non-self-terminating program is input to $P\prime$, $P\prime$ will skip the loop and will terminate.

Now what if we give a representation of $P\prime$ to itself. Will it halt? The answer generates a contradiction: If $P\prime$ is a self-terminating program, then when it is input to itself, it will not terminate. And if $P\prime$ is not self-terminating, when it is input to itself, it will terminate. Because our assumption that *P* solves the halting problem has led to a contradiction, we have to conclude that it wasn’t a very good assumption in the first place. Therefore, there is no program that can solve the halting problem.

The topic of computability is a fundamental part of the computer science curriculum, usually taught in a sophomore- or junior-level theory of computation course.

Chapter Summary

conditional loop

counting loop

do-while statement

infinite loop

initializer

limit bound

loop body

loop bound

loop entry condition

nested loop

postcondition

precondition

priming read

repetition structure

sentinel bound

unit indexing

updater

while statement

zero indexing

A *repetition structure* is a control structure that allows a statement or sequence of statements to be repeated.

All loop structures involve three elements—an *initializer*, a *loop entry condition* or a *loop boundary condition*, and an *updater*.

When designing a loop, it is important to analyze the loop structure to make sure that the loop bound will eventually be satisfied.

The for statement has the following syntax:

```
for ( initializer ; loop entry condition ; updater )
for loop body ;
```

The while statement takes the following form:

```
while ( loop entry condition )
loop body ;
```

The do-while statement has the following general form:

```
do
loop body ;
while ( loop entry condition ) ;
```

When designing a loop, it is important to analyze the loop structure to make sure that the loop bound will eventually be satisified. Table 6.2 summarizes the types of loop bounds that we have identified.

Structured programming is the practice of writing programs that are built up from a small set of predefined control structures—the *sequence*, *selection*, *repetition*, and *method-call* structures. An important feature of these structures is that each has a single entry and exit.

A *precondition* is a condition that must be true before a certain code segment executes. A *postcondition* is a condition that must be true when a certain code segment is finished. Preconditions and postconditions should be used in the design, coding, documentation, and debugging of algorithms and methods.

Identify the syntax error in the following for loop statements:

1. Commas are used instead of semicolons in the header.

```
for (int k = 5; k < 100; k++)
    System.out.println(k);
```

2. There shouldn't be 3 semicolons in the header

```
for (int k = 0; k < 12 ; k--)
    System.out.println(k);
```

Identify those statements that result in infinite loops:

1. Infinite loop because k is never incremented.
2. Infinite loop because k is always odd and thus never equal to 100.

Your sister is learning to count by fours. Write a for loop that prints the following sequence of numbers: 1, 5, 9, 13, 17, 21, 25.

```
for (int k = 1; k <= 25; k = k+4)
    System.out.print(k + " ");
```

What value will *j* have when the following loop terminates? *Answer*: *j* will be undefined when the loop terminates. It is a local variable whose scope is limited to the loop body.

```
for (int i = 0; i < 10; i++)
{
   int j;
   j = j + 1;
}
```

Write a nested for loop to print the following geometric pattern:

```
#
# #
# # #
# # # #
# # # # #

for (int row = 1; row <= 5; row++)  {    // For each row
  for (int col = 1; col <= row; col++) // Columns per row
    System.out.print('#');
  System.out.println();               // New line
} // row
```

Identify the syntax error in the following while structures:

1.
```
int k = 5;
while (k < 100) {
    System.out.println(k);
    k++                     << Missing semicolon
}
```

2.
```
int k = 0;
while (k < 12;) {           << Extra semicolon
    System.out.println(k);
    k++;
}
```

Determine the output and/or identify the error in each of the following while structures.

1.
```
int k = 0;
while (k < 100)
  System.out.println(k);  << Missing updater in loop body
```

   Output: infinite loop prints 0 0 0 0 0...

2.
```
while (k < 100) {           << Missing initializer
  System.out.println(k);
  k++;
}
```

Output: unpredictable since k's initial value is not known

Your younger sister is now learning how to count by sixes. Write a while loop that prints the following sequence of numbers: 0, 6, 12, 18, 24, 30, 36.

```
int k = 0;                  // Initializer
while (k <= 36) {           // Loop-entry condition
    System.out.println(k);
    k += 6;                   // Updater
}
```

If *N* is even, divide it by 2. If *N* is odd, subtract 1 and then divide it by 2. This will generate a sequence that is guaranteed to terminate at 0. For example, if *N* is initially 15, then you get the sequence 15, 7, 3, 1, 0. Write a method that implements this sequence using a while statement.

```
public static void sub1Div2(int N) {
    while(N != 0) {
        System.out.print(N + " ");
        if (N % 2 == 0)
            N = N / 2;
        else
            N = (N - 1) / 2;
    }
    System.out.println( N );
} // sub1Div2()
```

Identify the syntax error in the following do-while structures:

1.
```
int k = 0;
do while (k < 100) << Misplaced condition
{
     System.out.println(k);
     k++;
}                   << Belongs here
```

2.
```
int k = 0;
do {
    System.out.println(k);
    k++;
} while (k < 12) << Missing semicolon
```

Your sister has moved on to counting by sevens. Write a do-while loop that prints the following sequence of numbers: 1, 8, 15, 22, 29, 36, 43.

```
n = 1;                          // Initializer
do {
    System.out.print(n + " ");
    n += 7;                     // Updater
} while (n <= 43);               // Loop entry condition
```

Write a method to input and validate pizza sales.

```
public int getAndValidatePizzaPrice() {// Uses KeyboardReader
  int pizza = 0;
  do {
    reader.prompt("Input a pizza price (8, 10, or 15) ");
    reader.prompt("or 99 to end the list >> ");
    pizza = reader.getKeyboardInteger();
    if ((pizza != 99) && (pizza != 8) && (pizza != 10) &&
                        (pizza != 15))
      System.out.println("Error: you've entered an "
       + "invalid pizza price\n");    // Error input
    else                               // OK input
        System.out.println("You input " + pizza + "\n");
  } while ((pizza != 99) && (pizza != 8) &&
                  (pizza != 10) && (pizza != 15));
  return pizza;
} // getAndValidatePizzaPrice()
```

Write a method to input and validate pizza sales using the numbers 1, 2, and 3 to represent pizzas at different price levels.

```
public int getAndValidatePizzaPrice() { // Uses KeyboardReader
  int pizza = 0;
  do {
    reader.prompt("Input a 1,2 or 3 to indicate pizza"
              + "price ( 1($8), 2($10), or 3($15) ) ");
    reader.prompt("or 0 to end the list >> ");
    pizza = reader.getKeyboardInteger();
    if ((pizza < 0) || (pizza > 3))  // Error check
        System.out.println("Error: you've entered an "
                                + "invalid value\n");
    else                                 // OK input
        System.out.println("You input " + pizza + "\n");
  } while ( (pizza < 0) || (pizza > 3) );
  if (pizza == 1)
    return 8;
  else if (pizza == 2)
    return 10;
  else if (pizza == 3)
    return 15;
  else
    return 0;
} // getAndValidatePizzaPrice()
```

For each of the following problems, decide whether a counting loop structure, a while structure, or a do-while structure should be used, and write a pseudocode algorithm.

Printing the names of all the visitors to a Web site could use a counting loop because the exact number of visitors is known.

```
for each name in the visitor's log
    print the name
```

Validating that a user has entered a positive number requires a do-while structure in which you repeatedly read a number and validate it.

```
do
    read a number
    if number is invalid, print error message
while number is invalid
```

Changing all the backslashes (\) in a Windows Web page address, to the slashes (/) used in a Unix Web page address.

```
for each character in the Web page address
    if it is a backslash replace it with slash
```

Finding the largest in a list of numbers requires a while loop to guard against an empty list.

```
initialize maxMPG to smallest possible number
while there are more cars in the database
    if current car's MPG is greater than maxMPG
        replace maxMPG with it
```

Identify any errors in the following switch structures (if there is no error, specify the output):

1.
```
int k = 0;
switch (k)  // Syntax error: missing braces
case 0:
    System.out.println("zero");
    break;
case 1:
    System.out.println("one");
    break;
default:
    System.out.println("default");
    break;
```

2.
```
int k = 0;
switch (k + 1)
{
    case 0:
        System.out.println("zero");
        break;
    case 1:
        System.out.println("one"); // Output "one"
        break;
    default:
        System.out.println("default");
        break;
}
```

3.
```
int k = 6;
switch (k / 3.0) // Syntax error: not an integral value
{
    case 2:
        System.out.println("zero");
        break;
    case 3:
        System.out.println("one");
        break;
    default:
        System.out.println("default");
        break;
}
```

A switch statement to print ice cream flavors:

```
switch (flavor)
{
    case 1:
        System.out.println("Vanilla");
        break;
    case 2:
        System.out.println("Chocolate");
        break;
    case 3:
        System.out.println("Strawberry");
        break;
    default:
        System.out.println("Error");
}
```

```
public final int VANILLA = 0,
                 CHOCOLATE = 1,
                 STRAWBERRY = 2;
switch (flavor)
{
    case VANILLA:
        System.out.println("Vanilla");
        break;
    case CHOCOLATE:
        System.out.println("Chocolate");
        break;
    case STRAWBERRY:
        System.out.println("Strawberry");
        break;
    default:
        System.out.println("Error");
}
```

Identify the pre- and postconditions on *j* and *k* where indicated in the following code segment:

```
int j = 0; k = 5;
do {
    if (k % 5 == 0) {
                    // Precondition: j <= k
        j += k;
        k--;
    }
    else k *= k;
} while (j <= k);
                    // Postcondition: j > k
```

Identify the pre- and postconditions for the following method, which computes x^n for $n >= 0$.

```
// Precondition: N >= 0
// Postcondition: power(x,n) == x to the n
public double power(double x, int n ) {
    double pow = 1;
    for (int k = 1; k <= n; k++)
        pow = pow * x;
    return pow;
} // power()
```

Explain the difference between the following pairs of terms:

Counting loop and *conditional loop*.

For statement and *while statement*.

While statement and *do-while statement*.

Zero indexing and *unit indexing*.

Sentinel bound and *limit bound*.

Counting bound and *flag bound*.

Loop initializer and *updater*.

Named constant and *literal*.

Compound statement and *null statement*.

Fill in the blank.

=14pt

The process of reading a data item before entering a loop is known as a

_____.

A loop that does nothing except iterate is an example of

_____.

A loop that contains no body is an example of a

_____ statement.

A loop whose entry condition is stated as $(k < 100 \mid\mid k >= 0)$ would be an example of an

_____ loop.

A loop that should iterate until the user types in a special value should use a

_____ bound.

A loop that should iterate until its variable goes from 5 to 100 should use a

_____ bound.

A loop that should iterate until the difference between two values is less than 0.005 is an example of a

_____ bound.

=11pt

Identify the syntax errors in each of the following:

for (int k = 0; k < 100; k++) System.out.println(k)

for (int k = 0; k < 100; k++); System.out.println(k);

int k = 0 while k < 100 System.out.println(k); k++;

int k = 0; do System.out.println(k); k++; while k < 100 ;

Determine the output and/or identify the error in each of the following code segments:

for (int k = 1; k == 100; k += 2) System.out.println(k);

int k = 0; while (k < 100) System.out.println(k); k++;

for (int k = 0; k < 100; k++) ; System.out.println(k);

Write pseudocode algorithms for the following activities, paying particular attention to the *initializer*, *updater*, and *boundary condition* in each case.

a softball game

a five-question quiz

looking up a name in the phone book

Identify the pre- and postconditions for each of the statements that follow. Assume that all variables are int and have been properly declared.

int result = x / y;

int result = x

int x = 95; do x /= 2; while(x >= 0);

Write three different loops—a for loop, a while loop, and a do-while loop—to print all the multiples of 10, including 0, up to and including 1,000.

Write three different loops—a for loop, a while loop, and a do-while loop—to print the following sequence of numbers: 45, 36, 27, 18, 9, 0, -9, -18, -27, -36, -45.

Write three different loops—a for loop, a while loop, and a do-while loop—to print the following ski-jump design:

```
#
# #
# # #
# # # #
# # # # #
# # # # # #
# # # # # # #
```

The Straight Downhill Ski Lodge in Gravel Crest, Vermont, gets lots of college students on breaks. The lodge likes to keep track of repeat visitors. Straight Downhill's database includes an integer variable, *visit*, which gives the number of times a guest has stayed at the lodge (1 or more). Write the pseudocode to catch those visitors who have stayed at the lodge at least twice and to send them a special promotional package (pseudocode = send promo). (*Note:* The largest number of stays recorded is eight. The number nine is used as an end-of-data flag.)

Modify your pseudocode in the previous exercise. In addition to every guest who has stayed at least twice at the lodge receiving a promotional package, any guest with three or more stays should also get a $40 coupon good for lodging, lifts, or food.

Write a method that is passed a single parameter, *N*, and displays all the even numbers from 1 to *N*.

Write a method that is passed a single parameter, *N*, that prints all the odd numbers from 1 to *N*.

Write a method that is passed a single parameter, *N*, that prints all the numbers divisible by 10 from *N* down to 1.

Write a method that is passed two parameters—a char *Ch* and an int *N*—and prints a string of *N Chs*.

Write a method that uses a nested for loop to print the following multiplication table:

```
   1  2  3  4  5  6  7  8  9
1  1
2  2  4
3  3  6  9
4  4  8 12 16
5  5 10 15 20 25
6  6 12 18 24 30 36
7  7 14 21 28 35 42 48
8  8 16 24 32 40 48 56 64
9  9 18 27 36 45 54 63 72 81
```

Write a method that uses nested for loops to print the patterns that follow. Your method should use the following statement to print the patterns: `System.out.print('#')` .



Write a program that asks the user for the number of rows and the number of columns in a box of asterisks. Then use nested loops to generate the box.

Write a Java application that lets the user input a sequence of consecutive numbers. In other words, the program should let the user keep entering numbers as long as the current number is one greater than the previous number.

Write a Java application that lets the user input a sequence of integers terminated by any negative value. The program should then report the largest and smallest values that were entered.

How many guesses does it take to guess a secret number between 1 and *N*? For example, I'm thinking of a number between 1 and 100. I'll tell you whether your guess is too high or too low. Obviously, an intelligent first guess would be 50. If that's too low, an intelligent second guess would be 75. And so on. If we continue to divide the range in half, we'll eventually get down to one number. Because you can divide 100 seven times (50, 25, 12, 6, 3, 1, 0), it will take at most seven guesses to guess a number between 1 and 100. Write a Java Swing program that lets the user input a positive integer, *N*, and then reports how many guesses it would take to guess a number between 1 and *N*.

Suppose you determine that the fire extinguisher in your kitchen loses *X* percent of its foam every day. How long before it drops below a certain threshold (*Y* percent), at which point it is no longer serviceable? Write a Java Swing program that lets the user input the values *X* and *Y* and then reports how many weeks the fire extinguisher will last.

Leibnitz's method for computing π is based on the following convergent series:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots \tag{6.12.1}$$

How many iterations does it take to compute π to a value between 3.141 and 3.142 using this series? Write a Java program to find out.

Newton's method for calculating the square root of *N* starts by making a (nonzero) guess at the square root. It then uses the original guess to calculate a new guess, according to the following formula:

```
guess = (( N / guess) + guess) / 2;
```

No matter how wild the original guess is, if we repeat this calculation, the algorithm will eventually find the square root. Write a square root method based on this algorithm. Then write a program to determine how many guesses are required to find the square roots of different numbers. Uses Math.sqrt() to determine when to terminate the guessing.

Your employer is developing encryption software and wants you to develop a Java Swing Program that will display all of the primes less than *N*, where *N* is a number to be entered by the user. In addition to displaying the primes themselves, provide a count of how many there are.

Your little sister asks you to help her with her multiplication and you decide to write a Java application that tests her skills. The program will let her input a starting number, such as 5. It will generate multiplication problems ranging from from 5×1 to 5×12. For each problem she will be prompted to enter the correct answer. The program should check her answer and should not let her advance to the next question until the correct answer is given to the current question.

Write an application that prompts the user for four values and draws corresponding bar graphs using an ASCII character. For example, if the user entered 15, 12, 9, and 4, the program would draw

```
******************
**********
*********
****
```

Revise the application in the previous problem so that the bar charts are displayed vertically. For example, if the user inputs 5, 2, 3, and 4, the program should display

```
 **
 **         **
 **     **  **
 **  **  **  **
 **  **  **  **
--------------
```

The Fibonacci sequence (named after the Italian mathematician Leonardo of Pisa, ca. 1200) consists of the numbers $0, 1, 1, 2, 3, 5, 8, 13, ...$ in which each number (except for the first two) is the sum of the two preceding numbers. Write a method fibonacci(N) that prints the first *N* Fibonacci numbers.

The Nuclear Regulatory Agency wants you to write a program that will help determine how long certain radioactive substances will take to decay. The program should let the user input two values: a string giving the substance's name and its half-life in years. (A substance's half-life is the number of years required for the disintegration of half of its atoms.) The program should report how many years it will take before there is less than 2 percent of the original number of atoms remaining.

Modify the CarLoan program so that it calculates a user's car payments for loans of different interest rates and different loan periods. Let the user input the amount of the loan. Have the program output a table of monthly payment schedules.

This page titled 6.12: Special Topic- What Can Be Computed? is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **6.12: Special Topic- What Can Be Computed?** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

CHAPTER OVERVIEW

7: Strings and String Processing

Learning Objectives

After studying this chapter, you will

- Be more familiar with Java Strings.
- Know how to solve problems that involve manipulating strings.
- Be able to use loops in designing string-processing algorithms.

Introduction

You have already had an introduction to Strings in the early chapters of this text. In Chapter 2, we introduced the String data type and showed how to create String objects and use String methods, such as length(), concat(), and equals().

We have seen Strings used for GUI I/O operations when used Strings as the contents of JTextFields and other text components, as the values of JLabels, as the labels for JButtons, and so on. Strings are also used extensively in command-line interfaces.

Another important task that Strings are used for are as a standard way of presenting or displaying information about objects. As we saw in Chapter 2, one of the key conventions of the Java class hierarchy is that every class inherits the Object.toString() method, which can be used to provide a string representation of any object. For example, converts an int to a String, so that it can be used in JTextFields or JLabels.

Programmers often have to work with strings. Think of some of the tasks performed by a typical word processor, such as cut, paste, copy, and insert. When you cut and paste text from one part of the document to another, the program has to move one string of text, the cut, from one location in the document and insert it in another.

Strings are also important because they are our first look at a *data structure*. A **data structure** is a collection of data that is organized (structured) in some way. A **string** is a collection of character (char) data. Strings are important data structures in a programming language, and they are used to represent a wide variety of data.

The main purpose of this chapter is to provide a detailed discussion of Java's string-related classes, including the String, StringBuffer, and StringTokenizer classes. These are the important classes for writing string-processing applications. Our goal is to introduce the important String methods and illustrate common string-processing algorithms. We will review how to create strings from scratch and from other data types. We will learn how to find characters and substrings inside bigger strings. We will learn how to take strings apart and how to rearrange their parts. Finally, we will learn how to apply these string-processing skills in a program that plays the game of Hang Man.

7.1: Introduction
7.2: String Basics
7.3: Finding Things Within a String
7.4: Example- Keyword Search
7.5: From the Java Library- StringBuffer
7.6: Retrieving Parts of Strings
7.7: Example- Processing Names and Passwords
7.8: Processing Each Character in a String
7.9: Comparing Strings
7.10: From the Java Library- StringTokenizer
7.11: Handling Text in a Graphics Context (Optional)

This page titled 7: Strings and String Processing is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

7.1: Introduction

7.1: Introduction is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

7.2: String Basics

Before we cover the new material on Strings, let's first review what we know about this topic. In Java, Strings are considered full-fledged objects. A String object is a sequence of the characters that make up the string, plus the methods that are used to manipulate the string. The java.lang.String class (Fig. [fig-p349f1]) is a direct subclass of Object, and it contains many public methods that can be used to perform useful operations on strings (such as concatenation). We will discuss a selection of the more commonly used methods, but for a full listing and description of the String methods see

```
http://java.sun.com/j2se/1.5.0/docs/api/
```

Like other object variables, String variables serve as *references* to their respective objects. However, unlike other Java objects, Strings have certain characteristics in common with the primitive data types. For example, as we have already seen, Java allows for literal strings. A **string literal** is a sequence of zero or more characters contained in double quotes, such as "Socrates" and "" (the **empty string**). Java allows us to perform operations on literal strings, such as concatenation. As we have already seen, the expression "Hello" + "world" results in the string "Helloworld". Java also allows us to use string literals to initialize String variables with an assignment statement. These exceptional features greatly simplify the use of Strings in our programs. Given how much we use Strings, incorporating these features into Java seems like a good design decision.

Constructing Strings

To create String objects, the String class provides many constructors, including the following:

```
public String();        // Creates an empty string
   // Copy constructor: Creates a copy of a string
public String(String initial_value);
```

When we create an object using the first constructor, as in

```
String name = new String();
```

Java will create a String object and make name the reference to it. Figure [fig-emptystring] shows a hypothetical representation of a String object. In addition to storing the sequence of characters that make up the string, Java also stores an integer value representing the number of characters in the string. We have chosen to represent these two elements as the private instance variables, value, for the sequence of characters, and count for the number of characters. In fact, we don't know exactly how Java stores the sequence of characters. That information is hidden. As Figure [fig-emptystring] illustrates, when we use the default constructor, the value of the is the empty string and its count is 0.

The second constructor is the *copy constructor* for the String class. A **copy constructor** is a constructor that makes a duplicate, sometimes called a clone, of an object. Many Java classes have copy constructors. Consider the following statements:

```
String s1 = new String("Hello"););
String s2 = new String(s1);
```

These two statements would result in two distinct String objects, both storing the word "Hello".

Note that in the first of the preceding statements, we used the literal string "Hello" in the constructor. When Java encounters a new literal string in a program, it constructs an object for it. For example, if your program contained the literal "Socrates," Java would create an object for it and treat the literal itself as a reference to the object (Fig. [fig-literalstr]).

We often use a string literal to assign a value to a String variable:

```
String s;      // The value of s is initially null
s = "Socrates";// s now refers to "Socrates" object
```

In this case, the reference variable *s* is initially null—that is, it has no referent, no object, to refer to. However, after the assignment statement, *s* would refer to the literal object "Socrates," which is depicted in Figure [fig-literalstr]. Given these two statements, we still have only one object, the String object containing the word "Socrates.". But now we have two references to it: the literal string "Socrates," and the reference variable *s*.

Assignment statements can also be used as initializers when declaring a String variable:

```
String name1 = "";  // Reference to the empty string
String name2 = "Socrates";// References to "Socrates"
String name3 = "Socrates";
```

In this example, Java does not construct new String objects. Instead, as Figure [fig-literalrefs] shows, it simply makes the variables name1, name2, and name3 serve as references to the same objects that are referred to by the literal strings "" and "Socrates." This is a direct consequence of Java's policy of creating only one object to serve as the referent of a literal string, no matter how many occurrences there are of that literal in the program. Thus, these declarations result in no new objects, just new references to existing objects. The justification for this policy is that it saves lots of memory in our programs. Instead of creating a String object for each occurrence of the literal "Socrates," Java creates one object and lets all occurrences of "Socrates" refer to that object.

Finally, consider the following declarations, which do invoke the String constructors:

```
String name4 = new String();    // Creates an object
String name5 = new String("Socrates");
String name6 = name4;
```

In this case, as shown in Figure [fig-strobjects], Java creates two new objects and sets name4 to refer to the first and name5 to refer to the second. It gives name4 the empty string as its value, and it gives name5 "Socrates" as its value. But these two objects must be distinguished from the objects corresponding to the literals ("" and "Socrates") themselves. The declaration of name6 just creates a second reference to the object referred to by name4.

Concatenating Strings

Another way to build a String object is to concatenate two other strings. Recall from Chapter [chapter-objects] that there are two ways to perform string concatenation in Java: We can use the concat() method or the concatenation operator, +.

```
String lastName = "Onassis";
String jackie =
   new String("Jacqueline " + "Kennedy " + lastName);
System.out.println("Jacqueline".concat(lastName));
```

The second of these statements uses the *concatenation operator*, +, to create the String "Jacqueline Kennedy Onassis." The third statement uses the String method, concat(), to print "JacquelineOnassis."

Using the + symbol as the string concatenation operator is another example of *operator overloading*—using the same operator for two or more different operations—which we encountered in Chapter 5.

Note that primitive types are automatically promoted to Strings when they are mixed with concatenation operators. Thus, the statement

```
System.out.println("The sum of 5 and 5 = "+ (5 + 5));
```

will print the string "The sum of 5 and 5 = 10." Note that the integer addition—(5 + 5)—is performed first, before the integer result is converted into a String. If we had left off the parentheses around the addition operation, the second plus sign would also be interpreted as a concatenation operator. Thus,

-2pc

```
System.out.println("The concatenation of 5 and 5 = " + 5 + 5);
```

would print “The concatenation of 5 and 5 = 55.”

What will be printed by each of the following segments of code?

```
String s1 = "silly"; System.out.println(s1);
```

```
String s2 = s1; System.out.println(s2);
```

```
String s3 = new String (s1 + " stuff");
System.out.println(s3);
```

Write a String declaration that satisfies each of the following descriptions:

Initialize a String variable, *str1*, to the empty string.

Instantiate a String object, *str2*, and initialize it to the word *stop*.

Initialize a String variable, *str*, to the concatenation of *str1* and *str2*.

Evaluate the following expressions:

```
int M = 5, N = 10;
String s1 = "51", s2 = "75";
```

3

```
M + N
```

```
M + s1
```

```
s1 + s2
```

Draw a picture, similar to Figure 7.5, showing the objects and references that are created by the following declarations:

```
String s1, s2 = "Hello", s3 = "Hello";
String s4 = "hello";
String s5 = new String("Hello");
String s6 = s5;
String s7 = s3;
```

Indexing Strings

Programmers often need to take strings apart or put them together or rearrange them. Just think of the many word-processing tasks, such as cut and paste, that involve such operations. To help simplify such operations, it is useful to know how many characters a string contains and to number, or *index*, the characters that make up the string.

The number of characters in a string is called its *length*. The instance method, length(), returns an integer that gives the String’s length. For example, consider the following String declarations and the corresponding values of the length() method for each case:

```
String string1 = "";           string1.length()  ==> 0
String string2 = "Hello";      string2.length()  ==> 5
String string3 = "World";      string3.length()  ==> 5
String string4 = string2 + " "
    + string3;                 string4.length()  ==> 11
```

The position of a particular character in a string is called its **string index**. All Strings in Java are **zero indexed**—that is, the index of the first character is zero. (Remember, zero indexing is contrasted with **unit indexing**, in which we start counting at 1.) For example, in “Socrates,” the letter *S* occurs at index 0, the letter *o* occurs at index 1, *r* occurs at index 3, and so on. Thus, the String

"Socrates" contains eight characters indexed from 0 to 7 (Fig. [fig-zeroindex]). Zero indexing is customary in programming languages. We will see other examples of this when we talk about arrays and vectors.

Converting Data to Strings

The String.valueOf() method is a *class method* that is used to convert a value of some primitive type into a String object. For example, the expression, String.valueOf(128) converts its int argument to the String "128."

There are different versions of valueOf(), each of which has the following type of signature:

```
static public String valueOf(Type);
```

where Type stands for any primitive data type, including boolean, char, int, double, and so on.

The valueOf() method is most useful for initializing Strings. Because valueOf() is a class method, it can be used as follows to instantiate new String objects:

```
String number = String.valueOf(128); // Creates "128"
String truth = String.valueOf(true); // Creates "true"
String bee = String.valueOf('B');    // Creates "B"
String pi = String.valueOf(Math.PI);  // Creates "3.14159"
```

We have already seen that Java automatically promotes primitive type values to String where necessary, so why do we need the valueOf() methods? For example, we can initialize a String to "3.14159" as follows:

```
String pi = new String(""+Math.PI);// Creates "3.14"
```

In this case, because it is part of a concatenation expression, the value of Math.PI will automatically be promoted to a String value. The point of the valueOf() method is twofold. First, it may be the method that the Java compiler relies on to perform string promotions such as this one. Second, using it in a program—even when it is not completely necessary—makes the promotion operation explicit rather than leaving it implicit. This helps to make the code more readable. (Also, see Exercise 7.9.)

Evaluate each of the following expressions:

2

```
String.valueOf (45)
String.valueOf (128 - 7)
String.valueOf ('X')
```

Write an expression to satisfy each of the following

Convert the integer value 100 to the string "100".

Convert the character 'V' to the string "V".

Initialize a new String object to X times Y.

This page titled 7.2: String Basics is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **7.2: String Basics** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

7.3: Finding Things Within a String

Programmers often have to find the location of a particular character or substring in a string. For example, user names and passwords are sometimes stored in a single string in which the name and password are separated from each other by a special character, such as a colon (username:password). In order to get the name or password from such a string, it is convenient to have methods that will search the string and report the index of the colon character.

The indexOf() and lastIndexOf() methods are instance methods that can be used to find the index position of a character or a substring within a String. There are several versions of each:

```
public int indexOf(int character);
public int indexOf(int character, int startingIndex);
public int indexOf(String string);
public int indexOf(String string, int startingIndex);

public int lastIndexOf(int character);
public int lastIndexOf(int character, int startingIndex);
public int lastIndexOf(String string);
public int lastIndexOf(String string, int startingIndex);
```

The indexOf() method searches from left to right within a String for either a character or a substring. The lastIndexOf() method searches from right to left for a character or substring. To illustrate, suppose we have declared the following Strings:

```
String string1 = "";
String string2 = "Hello";
String string3 = "World";
String string4 = string2 + " " + string3;
```

Recalling that Strings are indexed starting at 0, searching for *o* in the various strings gives the following results:

```
string1.indexOf('o') ==> -1  string1.lastIndexOf('o') ==> -1
string2.indexOf('o') ==>  4  string2.lastIndexOf('o') ==>  4
string3.indexOf('o') ==>  1  string3.lastIndexOf('o') ==>  1
string4.indexOf('o') ==>  4  string4.lastIndexOf('o') ==>  7
```

Because string1 is the empty string, “”, it does not contain the letter *o*. Therefore, indexOf() returns -1, a value that cannot be a valid index for a String. This convention is followed in indexOf() and lastIndexOf(). Because string2 and string3 each contain only one occurrence of the letter *o*, both indexOf() and lastIndexOf() return the same value when used on these Strings. Because string4 contains two occurrences of *o*, indexOf() and lastIndexOf() return different values in this case. As Figure [fig-helloindex] shows, the first *o* in “Hello, World!” occurs at index 4, the value returned by indexOf(). The second *o* occurs at index 7, which is the value returned by lastIndexOf().

By default, the single-parameter versions of indexOf() and lastIndexOf() start their searches at their respective (left or right) ends of the string. The two-parameter versions of these methods allow you to specify both the direction and starting point of the search. The second parameter specifies the starting index. Consider these examples:

```
string4.indexOf('o', 5)     ==> 7
string4.lastIndexOf('o', 5) ==> 4
```

If we start searching in both cases at index 5, then indexOf() will miss the *o* that occurs at index 4. The first *o* it finds will be the one at index 7. Similarly, lastIndexOf() will miss the *o* that occurs at index 7 and will find the *o* that occurs at index 4.

The indexOf() and lastIndexOf() methods can also be used to find substrings:

-2pc

```
string1.indexOf("or") ==> -1  string1.lastIndexOf("or") ==> -1
string2.indexOf("or") ==> -1  string2.lastIndexOf("or") ==> -1
string3.indexOf("or") ==>  1  string3.lastIndexOf("or") ==>  1
string4.indexOf("or") ==>  7  string4.lastIndexOf("or") ==>  7
```

The substring “or” does not occur in either string1 or string2. It does occur beginning at location 1 in string3 and beginning at location 7 in string4. For this collection of examples, it doesn’t matter whether we search from left to right or right to left.

Suppose the String variable s has been initialized to “mom.” Evaluate each of the following expressions:

3

```
s.indexOf("m");
s.indexOf("o");
s.indexOf("M");
```

Evaluate the expressions given the String declaration `String s1 = "Java, Java, Java";`

=0pt

2

```
s1.length()
String.valueOf(s1.length())
s1.indexOf('a')
s1.lastIndexOf('a')
s1.indexOf("av")
s1.lastIndexOf("av")
s1.indexOf('a', 5)
s1.lastIndexOf('a', 5)
s1.indexOf("av", s1.length() - 10)
s1.lastIndexOf("av",
s1.length() - 4)
s1.indexOf("a", s1.indexOf("va"))
```

Evaluate the following expression:

```
String tricky = "abcdefg01234567";
tricky.indexOf(String.valueOf( tricky.indexOf("c")));
```

This page titled 7.3: Finding Things Within a String is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **7.3: Finding Things Within a String** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

7.4: Example- Keyword Search

One of the most widely used Web browser functions is the search utility. You probably know how it works. You type in a keyword and click on a button, and it returns with a list of Web pages that contain the keyword.

Suppose you were writing a browser in Java. How would you implement this function? Of course, we don't know yet how to read files or Web pages, and we won't cover that until Chapter 11. But, for now, we can write a method that will search a string for all occurrences of a given keyword. That's at least part of the task that the browser's search engine would have to do.

So we want a method, keywordSearch(), that takes two parameters, one for the string that's being searched, and the other representing the keyword. Let's have the method return a String that lists the number of keyword occurrences, followed by the index of each occurrence. For example, if we asked this method to find all occurrences of *is* in "This is a test," it should return the string "2: 2 5" because there are two occurrences of *is*, one starting at index 2 and the other at index 5 in the string.

The algorithm for this method will require a loop, because we want to know the location of every occurrence of the keyword in the string. One way to do this would be to use the indexOf() method to search for the location of substrings in the string. If it finds the keyword at index *N*, it should record that location and then continue searching for more occurrences starting at index $N+1$ in the string. It should continue in this way until there are no more occurrences.

```
Suppose S is our string and K is the keyword.
Initialize a counter variable and result string.
Set Ptr to the indexOf() the first occurrence of K in S.
While (Ptr != -1)
    Increment the counter
    Insert Ptr into the result string
    Set Ptr to the next location of the keyword in S
Insert the count into the result string
Return the result string as a String
```

As this pseudocode shows, the algorithm uses a while loop with a *sentinel bound*. The algorithm terminates when the indexOf() method returns a -1, indicating that there are no more occurrences of the keyword in the string.

Translating the pseudocode into Java gives us the method shown in Figure [fig-keywordsearch]. Note how string concatenation is used to build the . Each time an occurrence is found, its location (ptr) is concatenated to the right-hand side of the resultStr. When the loop terminates, the number of occurrences (count) is concatenated to the left-hand side of the resultStr.

```
/**
 * Pre:  s and keyword are any Strings
 * Post: keywordSearch() returns a String containing the
 *  number of occurrences of keyword in s, followed
 *  by the starting location of each occurrence
 */
public String keywordSearch(String s, String keyword) {
  String resultStr = "";
  int count = 0;
  int ptr = s.indexOf(keyword);
  while (ptr != -1) {
    ++count;
    resultStr = resultStr + ptr + " ";
    ptr = s.indexOf(keyword, ptr + 1);// Next occurrence
  }
  resultStr = count + ": " + resultStr;// Insert the count
  return resultStr;                    // Return as a String
} // keywordSearch()
```

Testing and Debugging

What test data should we use for the keywordSearch() method? One important consideration in this case is to test that the method works for all possible locations of the keyword within the string. Thus, the method should be tested on strings that contain keyword occurrences at the beginning, middle, and end of the string. We should also test the method with a string that doesn't contain the keyword. Such tests will help verify that the loop will terminate properly in all cases. Given these considerations, Table 7.1 shows the tests that were made. As you can see from these results, the method did produce the expected outcomes. While these tests do not guarantee its correctness, they provide considerable evidence that the algorithm works correctly.

ll
Test Performed & Expected Result

```
keywordSearch("this is a test","is")              &          2:          2          5
keywordSearch("able was i ere i saw elba","a")    &    4:    0    6    18    24
keywordSearch("this is a test","taste")  & 0:
```

This page titled 7.4: Example- Keyword Search is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **7.4: Example- Keyword Search** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

7.5: From the Java Library- StringBuffer

From the Java Library: java.lang.StringBuffer

with the keywordSearch() method is that it is not very efficient because a String in Java is a **read-only** object. This means that once it has been instantiated, a String cannot be changed. You cannot insert new characters or delete existing characters from it.

Given this fact, how is it possible that the resultStr in the keywordSearch() ends up with the correct value? he answer is that every time we assign a new value to resultStr, Java has to create a new String object. Figure 7.9 illustrates the process. Thus, given the statement

```
resultStr = resultStr + ptr + " ";
```

Java will evaluate the right-hand side, which creates a new String object whose value would be the concatenation of the right-hand-side elements, `resultStr + ptr + " "` (Fig. 79a). It would then assign the new object as the new referent of resultStr (Fig. 7.9b). This turns the previous referent of resultStr into an **orphan object**—that is, into an object that no longer has any references to it. Java will eventually dispose of these orphaned objects, removing them from memory in a process known as **garbage collection**. However, creating and disposing of objects is a task that consumes the computer's time.

The fact that this assignment statement occurs within a loop means that several new objects are created and later garbage collected. Because object creation is a relatively time-consuming and memory-consuming operation, this algorithm is somewhat wasteful of Java's resources.

Of course, except for the inefficiency of doing it this way, no real harm is done by this algorithm used in the keywordSearch() method. Java's garbage collector will automatically reclaim the memory used by the orphaned object. However, this algorithm does consume more of Java's resources than other algorithms we might use.

A more efficient way to write the keywordSearch() method would make use of a StringBuffer to store and construct the resultStr. Like the String class, the java.lang.StringBuffer class also represents a string of characters. However, unlike the String class, a StringBuffer can be modified, and it can grow and shrink in length as necessary. As Figure 7.10 shows, the StringBuffer class contains several of the same kind of methods as the String class, for example, charAt() and length(). But it also contains methods that allow characters and other types of data to be inserted into a string, such as append(), insert(), and setCharAt(). Most string-processing algorithms use StringBuffers instead of Strings as their preferred data structure.

The StringBuffer class provides several methods that are useful for string processing. The constructor method, StringBuffer(String), makes it easy to convert a String into a StringBuffer. Similarly, once you are done processing the buffer, the toString() method makes it easy to convert a StringBuffer back into a String.

The typical way to use a StringBuffer is shown in the following revised version of the keywordSearch() method:

```
public String keywordSearch(String s, String keyword) {
                                  // Create StringBuffer
  StringBuffer resultStr = new StringBuffer();
  int count = 0;
  int ptr = s.indexOf(keyword);
  while (ptr != -1) {
    ++count;
    resultStr.append(ptr + " ");  // Append to buffer
    ptr = s.indexOf(keyword, ptr + 1);
  }
  resultStr.insert(0, count + ": ");
  return resultStr.toString(); // Convert buffer to String
} // keywordSearch()
```

We declare resultStr as a StringBuffer instead of a String. Then, instead of concatenating the ptr and reassigning the resultStr, we append() the ptr to the resultStr for each occurrence of a keyword. Similarly, after the loop exits, we insert() the count at the front

(index 0) of the resultStr. Finally, we convert resultStr into a String by using the toString() method before returning the method's result.

One advantage of the StringBuffer class is that there are several versions of its insert() and append() methods. These make it possible to insert any type of data—int, double, Object, and so on—into a StringBuffer. The method itself takes care of converting the data into a string for us.

To summarize, String objects in Java are *immutable*. So when a String is "modified," this really means that a new String object is created and the old String object must be garbage collected. This is somewhat inefficient, especially if done repeatedly within a loop. To avoid these inefficiencies, use a StringBuffer instead of a String in such contexts.

This page titled 7.5: From the Java Library- StringBuffer is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **7.5: From the Java Library- StringBuffer** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

7.6: Retrieving Parts of Strings

Programmers often need to retrieve an individual character or a part of a string from a string, as, for example, in a word processing program when a part of a string is copied or deleted. In this section we look at methods that help us with these kinds of tasks.

The charAt(int index) method is a String instance method that can be used to retrieve the character stored at a certain index. The several varieties of the substring() method can be used to retrieve a substring of characters from a String. These methods are defined as follows:

```
public char charAt(int index)
public String substring(int startIndex)
public String substring(int startIndex, int endIndex)
```

The charAt() method returns the character located at the index supplied as its parameter. Thus, str.charAt(0) retrieves the first character in str, while str.charAt(str.length()-1) retrieves the last character.

The substring() methods work in a similar way, except that you need to specify both the starting and the ending index of the substring you wish to retrieve. The first version of substring(int startIndex) takes a single parameter and returns a String consisting of all the characters beginning with startIndex and continuing up to the end of the String. For example, if the str is “HelloWorld”, then str.substring(5) would return “World” and str.substring(3) would return “loWorld”:

```
String str = "HelloWorld";
str.substring(5)            ==> "World"
str.substring(3)            ==> "loWorld"
```

The substring(int, int) version requires that you specify both the starting and ending index of the substring. The second index always points to the character that is one beyond the last character in the String you want to retrieve. For example,

```
//   INDEX:   0123456789
String str = "HelloWorld";
str.substring(5,7)              ==> "Wo"
str.substring(0,5)              ==> "Hello"
str.substring(5, str.length())  ==> "World"
```

Note here that when we want to retrieve “Wo” from str, we specify its substring as indexes 5 and 7; the 7 points to the character just beyond “Wo.” Similarly, substring(0,5), picks out the first five characters (“Hello”). In the third example, the length() method specifies the substring beginning at index 5 and extending to the end of the string. This is equivalent to str.substring(5):

```
//   INDEX:   0123456789
String str = "HelloWorld";
str.substring(5, str.length())  ==> "World"
str.substring(5)                ==> "World"
```

The fact that the second parameter in substring() refers to the character one beyond the desired substring may seem a bit confusing at first, but it is actually a very useful way to designate a substring. For example, many string-processing problems have to do with retrieving substrings from a **delimited string**, which is a string that contains special characters that separate the string into certain substrings. For example, consider the string “substring1:substring2,” in which the *delimiter* is the colon, ’:’. The following code retrieves the substring preceding the delimiter:

```
String str = "substring1:substring2";
int n = str.indexOf(':');
str.substring(0,n)              ==> "substring1"
```

Thus, by making the second index of substring() refer to the character one beyond the last character in the desired substring, we can use indexOf() and substring() together to process delimited strings. Note that it is not necessary to use a temporary variable *n* to store the index of the delimiter, because the two method calls can be nested:

```
String str = "substring1:substring2";
str.substring(0,str.indexOf(':'))  ==> "substring1"
```

Given the String declaration

```
String s = "abcdefghijklmnopqrstuvwxyz";
```

evaluate each of the following expressions:

2

```
s.substring(20)
s.substring(1, 5)
s.substring(23)
s.substring(23, 25)
s.substring(s.indexOf('x'))
```

Given the preceding declaration of s, evaluate each of the following expressions:

```
s.substring(20, s.length())
s.substring(s.indexOf('b'), s.indexOf('f'))
s.substring(s.indexOf("xy"))
s.substring(s.indexOf(s.charAt(23)))
s.substring(s.length() - 3)
```

This page titled 7.6: Retrieving Parts of Strings is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **7.6: Retrieving Parts of Strings** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

7.7: Example- Processing Names and Passwords

Many computer systems store user names and passwords as delimited strings, such as

```
smith:bg1s5xxx
mccarthy:2ffo900ssi
cho:biff4534ddee4w
```

Obviously, if the system is going to process passwords, it needs some way to take apart these name-password pairs.

Let's write methods to help perform this task. The first method will be passed a name-password pair and will return the name. The second method will be passed a name-password pair and will return the password. In both cases, the method takes a single String parameter and returns a String result:

```
String getName(String str);
String getPassword(String str);
```

To solve this problem we can make use of two String methods. We use the indexOf() method to find the location of the *delimiter*—which is the colon, ":"—in the name-password pair and then we use substring() to take the substring occurring before or after the delimiter. It may be easier to see this if we take a particular example:

```
 INDEX:           1         2
 INDEX: 012345678901234567890
        jones:b34rdffg12    // (1)
        cho:rtf546          // (2)
```

In the first case, the delimiter occurs at index position 5 in the string. Therefore, to take the name substring, we would use substring(0,5). To take the password substring, we would use substring(6). Of course, in the general case, we would use variables to indicate the position of the delimiter, as in the following methods:

```
public static String getName(String str) {
  int posColon = str.indexOf(':');  // Find the delimiter
  String result = str.substring(0, posColon); // Get name
  return result;
}

public static String getPassword(String str) {
  int posColon = str.indexOf(':');  // Find the delimiter
  String result = str.substring(posColon + 1); // Get passwd
  return result;
}
```

Note in both of these cases we have used local variables, posColon and , to store the intermediate results of the computation—that is, the index of the ":" and the name or password substring.

An alternative way to code these operations would be to use nested method calls to reduce the code to a single line:

```
return str.substring(0, str.indexOf(':'));
```

In this line, the result of str.indexOf(' : ') is passed immediately as the second argument to str.substring(). This version dispenses with the need for additional variables. And the result in this case is not unreasonably complicated. But whenever you are faced with a trade-off of this sort—nesting versus additional variables—you should opt for the style that will be easier to read and understand.

This page titled 7.7: Example- Processing Names and Passwords is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **7.7: Example- Processing Names and Passwords** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

7.8: Processing Each Character in a String

Many string-processing applications require you to process each character in a string. For example, to encrypt the string “hello” into “jgnnq”, we have to go through each letter of the string and change each character to its substitute.

These types of algorithms usually involve a counting loop bounded by the length of the string. Recall that the length() method determines the number of characters in a String and that strings are zero indexed. This means that the first character is at index 0, and the last character is at index length()-1. For example, to print each character in a string on a separate line, we would step through the string from its first to its last character and print each character::

```
// Precondition:  str is not null
// Postcondition: the letters in str will have been printed
public void printLetters(String str) {
  for (int k = 0; k < str.length(); k++)// For each char
    System.out.println(str.charAt(k));  //  Print it
}
```

Note that our loop bound is `k < str.length()` , since the index of the last character of any String is length()-1. Note also the use of str.charAt(k) to retrieve the *k*th character in str on each iteration of the loop.

Note the use of pre- and postconditions in the method’s comment block. The precondition states that str has been properly initialized—that is, it is not null. The postcondition merely states the expected behavior of the method.

Off-by-One Error

A frequent error in coding counter-controlled loops is known as the **off-by-one error**, which can occur in many different ways. For example, if we had coded the loop boundary condition as `k <= str.length()` , this would cause an off-by-one error, because the last character in str is at location length()-1. This would lead to a Java IndexOutOfBoundsException, which would be reported as soon as the program executed this statement.

The only way to avoid off-by-one errors is to check your loop bounds whenever you code a loop. Always make sure you have the loop counter’s initial and final values correct.

Example: Counting Characters

As another example of an algorithm that processes each character in a string, consider the problem of computing the frequency of the letters in a given document. Certain text analysis programs, such as programs that analyze encrypted data and spam filters, perform this type of function.

The countChar() method will count the number of occurrences of any particular character in a String (Fig. [fig-countchar]). This method takes two parameters: a String parameter that stores the string being searched and a char parameter that stores the character being counted.

```
// Precondition: Neither str nor ch are null
// Postcondition: countchar() == the number of ch in str
public int countChar(String str, char ch) {
  int counter = 0;              // Initialize a counter
  for (int k = 0; k < str.length(); k++)// For each char
    if (str.charAt(k) == ch)           //  If it's a ch
      counter++;                          //   count it
  return counter;                  // Return the result
}
```

Begin by initializing the local variable, counter, to 0. As in the previous example, the for loop here will iterate through each character of the String—from 0 to length()-1. On each iteration a check is made to see if the character in the *k*th position

(str.charAt(k)) is the character being counted. If so, counter is incremented. The method ends by returning counter, which, when the method completes, will store an integer representing the number of *ch*'s in str.

Example: Reversing a String

Another interesting method that processes each character in a string is the reverse() method. This is a method that reverses the letters in a string. For example, the reverse of "java" is "avaj".

The algorithm for the reverse() method should use a simple counting loop to reverse the letters in its String parameter. In this case, however, we can process the string from right to left, beginning at its last character and ending with its first character. That way we can just append each character, left to right, in the result string:

```
 /*
  * Pre:  s is any non null string
  * Post: s is returned in reverse order
  */
public String reverse(String s) {
    StringBuffer result = new StringBuffer();
    for (int k = s.length()-1; k >= 0; k--) {
        result.append(s.charAt(k));
    } //for
    return result.toString();
} // reverse()
```

Note that as in the other string-manipulation algorithms—for example, keywordSearch()—we should us a StringBuffer to store the method's result. Thus we declare the result StringBuffer at the beginning of the method and convert it back into a String at the end of the method.

Example: Capitalizing the First Letter

Another string manipulation method is the capitalize() method, which returns a String whose initial letter is capitalized but whose other letters are lowercase – for example, "Hello". We use the static toUpperCase() and toLowerCase() methods from the Character class to convert individual letters. We could also have used the methods of the same name that we wrote in Section 5.8. The algorithm converts the first letter to upper case and then loops through the remaining letters converting each to lowercase:

```
/*
 * Pre:  s is any non null string
 * Post: s is returned with only its first letter capitalized
 */
public String capitalize(String s) {
  if (s.length() == 0)    // Special case: empty string
    return s;
  StringBuffer result = new StringBuffer();
  result.append(Character.toUpperCase(s.charAt(0)));
                          // Convert the first letter
  for (int k = 1; k < s.length(); k++) { // And the rest
    result.append(Character.toLowerCase(s.charAt(k)));
  } //for
  return result.toString();
} // capitalize()
```

Write a Java program to test the methods described in this section. Organize the methods themselves into a single class, named StringProcessor, and design a second class to serve as the user interface. Because these methods are similar to the utility methods

of the Math class, it would be useful to declare them static. The user interface should prompt the user to input a string and should then print out the result of passing that string to each of the methods we developed.

Add a method to the StringProcessor class that will remove all blanks from a string. It should take a String parameter and should return a String result.

Miscellaneous String Methods

In addition to the several String class methods we have discussed—valueOf(), equals(), indexOf(), lastIndexOf(), charAt(), substring()—Table 7.2 shows some of the other useful methods in the String class. Note that because of what we said about the read-only nature of Strings, methods such as toUpperCase(), toLowerCase(), and trim() do not change their string. Instead they produce a new string. If you want to use one of these methods to convert a string, you must reassign its result back to the original string:

```
String s = new String("hello world");
s = s.toUpperCase(); // s now equals "HELLO WORLD"
```

ll
Method Signature &Example

boolean endsWith(String suffix) & `"Perfection".endsWith("tion")` ⇒ true & `"Perfection".startsWith("Per")` ⇒ true & `"Perfection".startsWith("fect",3)` ⇒ true & `"Perfection".toUpperCase()` ⇒ `"PERFECTION"` & `"Perfection".toLowerCase()` ⇒ `"perfection"` & `"Perfection".trim()` ⇒ `"Perfection"`

This page titled 7.8: Processing Each Character in a String is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **7.8: Processing Each Character in a String** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

7.9: Comparing Strings

Comparing strings is another important task. For example, when a word processor performs a search and replace operation, it needs to identify strings in the text that match the target string.

Strings are compared according to their *lexicographic order*—that is, the order of their characters. For the letters of the alphabet, lexicographic order just means alphabetical order. Thus, *a* comes before *b* and *d* comes after *c*. The string “hello” comes before “jello” because *h* comes before *j* in the alphabet.

For Java and other programming languages, the definition of lexicographic order is extended to cover all the characters that make up the character set. We know, for example, that in Java’s Unicode character set the uppercase letters come before the lowercase letters (Table 5.13). So, the letter *H* comes before the letter *h* and the letter *Z* comes before the letter *a*.

Lexicographic order can be extended to include strings of characters. Thus, “Hello” precedes “hello” in lexicographic order because its first letter, *H*, precedes the first letter, *h*, in “hello.” Similarly, the string “Zero” comes before “aardvark,” because *Z* comes before *a*. To determine lexicographic order for strings, we must perform a character-by-character comparison, starting at the first character and proceeding left to right. As an example, the following strings are arranged in lexicographic order:

-4.5pc

```
"" "!" "0" "A" "Andy" "Z" "Zero" "a" "an" "and" "andy" "candy" "zero"
```

We can define **lexicographic order** for strings as follows:

Perhaps a more precise way to define lexicographic order is to define a Java method:

```
public boolean precedes(String s1, String s2) {
                            // Pick shorter length
  int minlen = Math.min(s1.length(), s2.length());
               // For each char in shorter string}
  for (int k=0; k < minlen; k++) {
    if (s1.charAt(k) != s2.charAt(k))// If chars unequal
       //  return true if s1's char precedes s2's
      return s1.charAt(k) < s2.charAt(k);
  }
            // If all characters so far are equal
        //  then s1 < s2 if it is shorter than s2
  return s1.length() < s2.length();
} // precedes()}
```

This method does a character-by-character comparison of the two strings, proceeding left to right, starting at the first character in both strings. Its for loop uses a counting bound, which starts at *k* equal to zero and counts up to the length of the shorter string. This is an important point in designing this algorithm. If you don’t stop iterating when you get past the last character in a string, your program will generate a StringIndexOutOfBounds exception. To prevent this error, we need to use the shorter length as the loop bound.

Note that the loop will terminate early if it finds that the respective characters from *s1* and *s2* are unequal. In that case, *s1* precedes *s2* if *s1*’s *k*th character precedes *s2*’s. If the loop terminates normally, that means that all the characters compared were equal. In that case, the shorter string precedes the longer. For example, if the two strings were “alpha” and “alphabet,” then the method would return true, because “alpha” is shorter than “alphabet.”

Arrange the following strings in lexicographic order:

```
zero bath bin alpha Alpha Zero Zeroes a A z Z
```

Modify the precedes() method so that it will also return true when *s1* and *s2* are equal—for example, when *s1* and *s2* are both "hello".

Object Identity Versus Object Equality

Java provides several methods for comparing Strings:

```
public boolean equals(Object anObject); // Overrides Object.equals()
public boolean equalsIgnoreCase(String  anotherString);
public int compareTo(String  anotherString);
```

The first comparison method, equals(), overrides the Object.equals() method. Two Strings are equal if they have the exact same letters in the exact same order. Thus, for the following declarations,

```
String s1 = "hello";
String s2 = "Hello";
```

is false, but `s1.equals("hello")` is true.

You have to be careful when using Java's equals() method. According to the default definition of equals(), defined in the Object class, "equals" means "identical." Two Objects are equal only if their names are references to the same object.

This is like the old story of the morning star and the evening star, which were thought to be different objects before it was discovered that both were just the planet Venus. After the discovery, it was clear that "the morning star" and "the evening star" and "Venus" were just three different references to one and the same object (Fig. [fig-venus]).

We can create an analogous situation in Java by using the following JButton definitions:

```
JButton b1 = new Button("a");
JButton b2 = new Button("a");
JButton b3 = b2;
```

Given these three declarations, b1.equals(b2) and b1.equals(b3) would be false, but b2.equals(b3) would be true because *b2* and *b3* are just two names for the same object (Fig. 7.13). So, in this case, "equals" really means "identical."

Moreover, in Java, when it is used to compare two objects, the equality operator (==) is interpreted in the same way as the default Object.equals() method. So, it really means object identity. Thus, b1 == b2 would be false, because b1 and b2 are different objects, but b2 == b3 would be true because b2 and b3 refer to the same object.

These points are illustrated in the program shown in Figure [fig-equals]. This program uses methods isEquals()

```
import java.awt.*;

public class TestEquals {
  static Button b1 = new Button ("a");
  static Button b2 = new Button ("b");
  static Button b3 = b2;

  private static void isEqual(Object o1, Object o2) {
    if (o1.equals(o2))
      System.out.println(o1.toString() + " equals " + o2.toString());
    else
      System.out.println(o1.toString() + " does NOT equal " +
                                                    o2.toString());
  } // isEqual()

  private static void isIdentical(Object o1, Object o2) {
    if (o1 == o2)
      System.out.println(o1.toString() + " is identical to " +
                                                    o2.toString());
    else
      System.out.println(o1.toString() + " is NOT identical to " +
                                                    o2.toString());
  } // isIdentical()

  public static void main(String argv[]) {
    isEqual(b1, b2);         // not equal
    isEqual(b1, b3);         // not equal
    isEqual(b2, b3);         // equal

    isIdentical(b1, b2);     // not identical
    isIdentical(b1, b3);     // not identical
    isIdentical(b2, b3);     // identical
  } // main()
} // TestEquals
```

and isIdentical() to perform the comparisons and print the results. This program will produce the following output:

```
java.awt.Button[button0,0,0,0x0,invalid,label=a]
   does NOT equal java.awt.Button[button1,0,0,0x0,invalid,label=b]
java.awt.Button[button0,0,0,0x0,invalid,label=a]
   does NOT equal java.awt.Button[button1,0,0,0x0,invalid,label=b]
java.awt.Button[button1,0,0,0x0,invalid,label=b]
   equals java.awt.Button[button1,0,0,0x0,invalid,label=b]
java.awt.Button[button0,0,0,0x0,invalid,label=a]
   is NOT identical to java.awt.Button[button1,0,0,0x0,invalid,label=b]
java.awt.Button[button0,0,0,0x0,invalid,label=a]
   is NOT identical to java.awt.Button[button1,0,0,0x0,invalid,label=b]
java.awt.Button[button1,0,0,0x0,invalid,label=b]
   is identical to java.awt.Button[button1,0,0,0x0,invalid,label=b]
```

String Identity Versus String Equality

In comparing Java Strings, we must be careful to distinguish between object identity and string equality. Thus, consider the following declarations, which create the situation shown in Figure [fig-strings].

```
String s1 = new String("hello");
String s2 = new String("hello");
String s3 = new String("Hello");
String s4 = s1;            // s1 and s4 are now identical
String s5 = "hello";
String s6 = "hello";
```

Given these declarations, we would get the following results if we compare the equality of the Strings:

```
s1.equals(s2) ==> true  s1.equalsIgnoreCase(s3)  ==> true
s1.equals(s3) ==> false s1.equals(s5)            ==> true
s1.equals(s4) ==> true  s1.equals(s6)            ==> true
```

and the following results if we compare their identity:

```
s1 == s2  ==> false           s1 == s3  ==> false
s1 == s4  ==> true            s1 == s5  ==> false
s5 == s6  ==> true
```

The only true identities among these Strings are *s1* and *s4*, and *s5* and *s6*. In the case of *s5* and *s6*, both are just references to the literal string, “hello”, as we described in Section 2. The program in Figure [fig-equals2] illustrates these points.

```
import java.awt.*;

public class TestStringEquals {
  static String s1 = new String("hello"); // s1 and s2 are equal, not identical
  static String s2 = new String("hello");
  static String s3 = new String("Hello"); // s1 and s3 are not equal
  static String s4 = s1;                  // s1 and s4 are identical
  static String s5 = "hello";             // s1 and s5 are not identical
  static String s6 = "hello";             // s5 and s6 are identical

  private static void testEqual(String str1, String str2) {
    if (str1.equals(str2))
      System.out.println(str1 + " equals " + str2);
    else
      System.out.println(str1 + " does not equal " + str2);
  } // testEqual()

  private static void testIdentical(String str1, String str2) {
    if (str1 == str2)
      System.out.println(str1 + " is identical to " + str2);
    else
      System.out.println(str1 + " is not identical to " + str2);
  } // testIdentical()

  public static void main(String argv[]) {
    testEqual(s1, s2);        // equal
    testEqual(s1, s3);        // not equal
    testEqual(s1, s4);        // equal
    testEqual(s1, s5);        // equal
    testEqual(s5, s6);        // equal

    testIdentical(s1, s2);    // not identical
    testIdentical(s1, s3);    // not identical
    testIdentical(s1, s4);    // identical
    testIdentical(s1, s5);    // not identical
    testIdentical(s5, s6);    // identical
  } // main()
}// TestStringEquals
           ------Program Output-----
           hello equals hello
           hello does not equal Hello
           hello equals hello
           hello equals hello
           hello equals hello
           hello is not identical to hello
           hello is not identical to Hello
           hello is identical to hello
           hello is not identical to hello
           hello is identical to hello
```

Given the String declarations,

```
String s1 = "java", s2 = "java", s3 = "Java";
String s4 = new String(s2);
String s5 = new String("java");
```

evaluate the following expressions:

3

`s1 == s2`

`s1.equals(s2)`

`s1 == s3`

`s1.equals(s3)`

`s2 == s3`

`s2.equals(s4)`

`s2 == s4`

`s1 == s5`

`s4 == s5`

Why are the variables in TestStringEquals declared static?

Given the following declarations,

```
String s1 = "abcdefghijklmnopqrstuvwxyz";
String s2 = "hello world";
```

write Java expressions to carry out each of the following operations:

Swap the front and back half of s1 giving a new string.

Swap "world" and "hello" in s2 giving a new string.

Combine parts of s1 and s2 to create a new string "hello abc".

This page titled 7.9: Comparing Strings is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **7.9: Comparing Strings** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

7.10: From the Java Library- StringTokenizer

From the Java Library: java.util.StringTokenizer

most widespread string-processing tasks is that of breaking up a string into its components, or **tokens**. For example, when processing a sentence, you may need to break the sentence into its constituent words, which are considered the sentence tokens. When processing a name-password string, such as “boyd:14irXp”, you may need to break it into a name and a password. Tokens are separated from each other by one or more characters which is known as **delimiters**. Thus, for a sentence, *white space*, including blank spaces, tabs, and line feeds, serve as the delimiters. For the password example, the colon character serves as a delimiter.

Java’s java.util.StringTokenizer class is specially designed for breaking strings into their tokens (Fig. [fig-p377f1]). When instantiated with a String parameter, a StringTokenizer breaks the string into tokens, using white space as delimiters. For example, if we instantiated a StringTokenizer as in the code

```
StringTokenizer sTokenizer
  = new StringTokenizer("This is an English sentence.");
```

it would break the string into the following tokens, which would be stored internally in the StringTokenizer in the order shown:

```
This
is
an
English
sentence.
```

Note that the period is part of the last token (“sentence.”). This is because punctuation marks are not considered delimiters by default.

If you wanted to include punctuation symbols as delimiters, you could use the second StringTokenizer() constructor, which takes a second String parameter (Fig. 7.17). The second parameter specifies a string of those characters that should be used as delimiters. For example, in the instantiation,

```
StringTokenizer sTokenizer
    = new StringTokenizer("This is an English sentence.",
                          "\b\t\n,;.!");
```

various punctuation symbols (periods, commas, and so on) are included among the delimiters. Note that escape sequences (`\b\t\n`) are used to specify blanks, tabs, and newlines.

The hasMoreTokens() and nextToken() methods can be used to process a delimited string, one token at a time. The first method returns true as long as more tokens remain; the second gets the next token in the list. For example, here’s a code segment that will break a standard URL string into its constituent parts:

```
String url = "http://java.trincoll.edu/~jjj/index.html";
StringTokenizer sTokenizer = new StringTokenizer(url,":/");
while (sTokenizer.hasMoreTokens()) {
    System.out.println(sTokenizer.nextToken());
}
```

This code segment will produce the following output:

```
http
java.trincoll.edu
~jjj
index.html
```

The only delimiters used in this case were the ":" and "/" symbols. And note that nextToken() does not return the empty string between ":" and "/" as a token.

This page titled 7.10: From the Java Library- StringTokenizer is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **7.10: From the Java Library- StringTokenizer** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

7.11: Handling Text in a Graphics Context (Optional)

In order to create attractive GUIs, it is often necessary to be able to select and control the font that is used. Even a simple drawing task, such as being able to center a message in a panel, requires that we know the font's dimensions and be able to manipulate them. In this section, we learn how to work with Java's fonts and font control methods.

Each graphics context has an associated Font and FontMetrics object, and the Graphics class (Fig. [fig-fontmethods]) provides several methods to access them. A FontMetrics is an object that encapsulates important data about a font, such as its height and width. Java assigns a default font to each Graphics object. For example, this is the font used by the drawString() method, which we used in our very first Java programs back in Chapter [ch:intro2]. The particular font used is system dependent, but to override the default one can simply invoke the setFont() method:

```
g.setFont(new Font("TimesRoman", Font.ITALIC, 12));
```

In this case, the Font() constructor is used to specify a 12-point, italicized, *TimesRoman* font. Once the font is set, it will be used in all subsequent drawings.

The Font and FontMetrics Classes

The Font class (Fig. 7.19) provides a platform-independent representation of an individual font. A font is distinguished by its name, size, and style, and the Font class includes protected instance variables for these properties, as well as a constructor method that allows these three characteristics to be specified.

In order to understand how fonts work, it is necessary to distinguish between a *character*, which is a symbol that represents a certain letter or digit, and a **glyph**, which is a shape used to display the character. When you display a string, such as "Hello", Java maps each individual character into a corresponding shape, as defined by the particular font that is selected.

Java distinguishes between *physical* and *logical* fonts. A **physical font** is an actual font library that contains the data and tables needed to associate the correct glyph with a given character. Generally speaking, a given platform (host computer plus operating system) will have a collection of such fonts available on it.

A **logical font** is one of five font families that are supported by the Java runtime environment. These include Serif, SansSerif, Monospaced, Dialog, and DialogInput. Java also supports the following font styles: PLAIN, BOLD, ITALIC, and BOLD+ITALIC. Whereas the physical fonts are platform dependent, the logical fonts and styles are platform independent. When used in a program, they are mapped to real fonts available on the host system. If the host system does not have an exact match for the specified font, it will supply a substitute. For example, if you specify a 48-point, italic, Monospaced font,

```
Font myFont = new Font("Monospaced", Font.ITALIC, 48);
```

the system may map this to a 24-point, italic Courier font, if that is the largest fixed-spaced font available.

The Font() constructor is designed to work with any set of arguments. Thus, if you supply the name of a font that is not available, the system will supply a default font as a substitute. For example, on my system, specifying a nonexistent font named Random,

```
g.setFont(new Font("Random", Font.ITALIC, 12) );
g.drawString("Hello, World!! (random, italic, 12)", 30, 45);
```

produces the same font used as the mapping for a font named *Dialog*.

The Component.setFont() method can be used to assign a specific font to a button or window or other graphics component. All AWT and JFC components have an associated font, which can be accessed using the Component.setFont() and Component.getFont() methods. For example, the following code could be used to override a Button's font:

```
Button b = new Button("Label");
b.setFont(new Font("Times", Font.ITALIC, 14));
```

If 14-point, italic, Times font is not available on the host system, a substitute will be supplied.

Font Metrics

To illustrate how to use the FontMetrics class, let's write a "Hello, World!" application that centers its message both horizontally and vertically in its window. The message should be centered regardless of the size of the application window. Thus, we will have to position the text relative to the window size, which is something we learned in positioning geometric shapes. The message should also be centered no matter what font is used. This will require us to know certain characteristics of the font itself, such as the height and width of its characters, whether the characters have a fixed or variable width, and so on. In order to get access to these properties, we will use the FontMetrics class.

Figure 7.20 illustrates the various properties that are associated with a font. The **baseline** of a font refers to the line on which the bottom of most characters occurs. When drawing a string, the *x*- and *y*-coordinates determine the baseline of the string's first character. Thus, in

```
g.drawString("Hello, World!", 10, 40);
```

the bottom left of the *H* in "Hello, World!" would be located at (10, 40).

All characters ascend some distance above the baseline. This is known as the character's **ascent**. Some characters, such as *y*, may extend below the baseline, into what's known as the *descent*. Each font has a *maximum descent*. Similarly, some characters, such as accent characters, may extend above the *maximum ascent* into a space known as the *leading*.

The *height* of a font is defined as the sum (in pixels) of the ascent, descent, and leading values. The height is a property of the font itself rather than of any individual character. Except for fixed-width fonts, in which the width of all characters is the same, the characters that make up a font have varying widths. The width of an individual character is known as its *advance*.

The FontMetrics class (Fig. [fig-fontmetrics2]) provides methods for accessing a font's properties. These can be useful to control the layout of text on a GUI. For example, when drawing multiple lines of text, the getHeight() method is useful for determining how much space should be left between lines. When drawing character by character, the charWidth() method can be used to determine how much space must be left between characters. Alternatively, the stringWidth() method can be used to determine the number of pixels required to draw the entire string.

Example: Centering a Line of Text

Given this background, let's take on the task of centering a message in an application window. In order for this application to work for any font, we must take care not to base its design on characteristics of the particular font that we happen to be using. To underscore this point, let's design it to work for a font named Random, which, as we noted earlier, will be mapped to some font by the system on which the application is run. In other words, we will let the system pick a font for this application's message. An interesting experiment would be to run the application on different platforms to see what fonts are chosen.

The only method we need for this application is the paint() method. Let's begin by setting the font used by the graphics context to a random font. To get the characteristics of this font, we create a FontMetrics object and get the font metrics for the font we just created:

```
g.setFont(new Font("Random", Font.BOLD, 24));
FontMetrics metrics = g.getFontMetrics();
```

The next step is to determine the JFrame's dimensions using the getSize() method. This method returns an object of type Dimension. The java.awt.Dimension class (Fig. 7.22) represents the size (width and height) of a GUI component. A Dimension makes it possible to manipulate an object's width and height as a single entity. Note that the height and width variables are defined as public, which is an exception from the usual convention of defining instances variables as private or protected. The justification for this exception is probably to simplify the syntax of referring to an object's width and height. For example, the following syntax can be used to refer to a component's dimensions:

```
Dimension d = new Dimension(100, 50);
System.out.println("width = " + d.width +
                   " height = " + d.height);
```

Note the redundancy built into the Dimension class. For example, in addition to being able to set a Dimension's instance variables directly, public access methods are provided. Also, by defining more than one version of some access methods, the class achieves a higher level of flexibility. The same can be said for providing several different constructors, including a copy constructor. Finally, note how it overrides the equals() and toString() methods. These are all examples of good object-oriented design.

The Dimension object is used to calculate the *x*- and *y*-coordinates for the string. In order to center the string horizontally, we need to know its width, which is supplied by the metrics object. If the JFrame is d.width pixels wide, then the following expression subtracts the width of the string from the width of the JFrame and then divides the leftover space in half:

```
                            // Calculate coordinates
int x = (d.width - metrics.stringWidth(str)) / 2;
```

Similarly, the following expression adds the height of the string to the height of the JFrame and divides the leftover space in half:

```
int y = (d.height + metrics.getHeight()) / 2;
```

Taken together, these calculations give the coordinates for the lower left pixel of the first character in "Hello, World!!" The only remaining task is to draw the string (Fig. [fig-centertext]). Because the paint() method is called automatically whenever the JFrame is resized, this application, whose output is shown in Figure [fig-centertextscreen], will re-center its message whenever it is resized by the user.

```
import java.awt.*;
import javax.swing.*;

public class CenterText extends JFrame {
                // Print hello world! in center of frame
public void paint(Graphics g) {
  String str = "Hello, World!";
  g.setFont(new Font("Random", Font.PLAIN, 24)); // Random font
  FontMetrics metrics = g.getFontMetrics(); //  And its metrics

  Dimension d = getSize();       // Get the frame's size
            // Clear the frame
  g.setColor(getBackground());
  g.fillRect(0,0,d.width,d.height);
  g.setColor(Color.black);
                               // Calculate coordinates
  int x = (d.width - metrics.stringWidth(str)) / 2;
  int y = (d.height + metrics.getHeight()) / 2;

  g.drawString( str, x, y );          // Draw the string
} // paint()

public static void main(String args[]) {
  CenterText ct = new CenterText();
  ct.setSize(400,400);
  ct.setVisible(true);
}
} // CenterText
```

ascent

baseline

concatenation

copy constructor

data structure

delimited string

delimiter

empty string

garbage collection

glyph

lexicographic order

logical font

off-by-one error

orphan object

physical font

read only

string

string index

string literal

token

unit indexed

zero indexed

A String literal is a sequence of 0 or more characters enclosed within double quotation marks. A String object is a sequence of 0 or more characters, plus a variety of class and instance methods and variables.

A String object is created automatically by Java the first time it encounters a *literal string*, such as “Socrates,” in a program. Subsequent occurrences of the literal do not cause additional objects to be instantiated. Instead, every occurrence of the literal “Socrates” refers to the initial object.

A String object is created whenever the new operator is used in conjunction with a String() constructor—for example, new String("hello").

The String concatenation operator is the overloaded + symbol; it is used to combine two Strings into a single String: “hello” + “world” ==> “helloworld”.

Strings are indexed starting at 0. The indexOf() and lastIndexOf() methods are used for finding the first or last occurrence of a character or substring within a String. The valueOf() methods convert a nonstring into a String. The length() method determines the number of characters in a String. The charAt() method returns the single character at a particular index position. The various substring() methods return the substring at particular index positions in a String.

The overloaded equals() method returns true if two Strings contain the same exact sequence of characters. The == operator, when used on Strings, returns true if two references designate the same String object.

String objects are *immutable*. They cannot be modified.

A StringBuffer is a string object that can be modified using methods such as insert() and append().

A StringTokenizer is an object that can be used to break a String into a collection of *tokens* separated by *delimiters*. The whitespace characters—tabs, blanks, and newlines—are the default delimiters.

The FontMetrics class is used to obtain the specific dimensions of the the various Fonts. It is useful when you wish to center text. Fonts are inherently platform dependent. For maximum portability, it is best to use default fonts.

= 0pt

4

1. silly
2. silly
3. silly stuff

1. `String str1 = "";`
2. `String str2 = new String("stop");`
3. `String str3 = str1 + str2;`

= 0pt

4

1. 15
2. `"551"`
3. `"5175"`

See Figure [fig-strobjects3]. [ans-strobjects2]

4

1. `"45"`
2. `"121"`

3. `"X"`

1. `String.valueOf(100)`
2. `String.valueOf('V');`
3. `String s = new String(String.valueOf(X * Y));`

4

1. 0
2. 1
3. -1

4

1. 16
2. `"16"`
3. 1
4. 15
5. 1
6. 13
7. 7
8. 3
9. 7
10. 7
11. 3

Evaluate the following expression:

```
String tricky = "abcdefg01234567";
tricky.indexOf(String.valueOf(tricky.indexOf("c")));
tricky.indexOf(String.valueOf(2));
tricky.indexOf("2");
Answer: 9
```

4

1. `"uvwxyz"`
2. `"bcde"`
3. `"xyz"`
4. `"xy"`
5. `"xyz"`

4

1. `"uvwxyz"`
2. `"bcde"`
3. `"xyz"`
4. `"xyz"`
5. `"xyz"`

A class to test the string methods.

```
public class StringProcessorTest {
  public static void main(String[] args) {
    KeyboardReader kb = new KeyboardReader();
    kb.prompt("Input a String or - stop - to quit: ");
    String str = kb.getKeyboardInput();
    while (!str.equals("stop")){
      kb.display("Testing printLetters()\n");
      StringProcessor.printLetters(str);
      kb.display("testing countChars()\n");
      kb.display("Total occurences of e = ");
      kb.display(StringProcessor.countChar(str,'e') + "\n");
      kb.display("Testing reverse()\n");
      kb.display(StringProcessor.reverse(str)+ "\n");
      kb.display("Testing capitalize()\n");
      kb.display(StringProcessor.capitalize(str) + "\n\n");
      kb.prompt("Input a String or - stop - to quit: ");
      str = kb.getKeyboardInput();
    } // while
  } //main()
} // StringProcessorTest class
```

Method to remove all blanks from a string:

```
// Pre: s is a non null string
// Post: s is returned with all its blanks removed
public String removeBlanks(String s) {
  StringBuffer result = new StringBuffer();
  for (int k = 0; k < s.length();  k++)
    if (s.charAt(k) != ' ')      // If this is not a blank
      result.append(s.charAt(k));  //  append it to result
  return result.toString();
}
```

A Alpha Z Zero Zeroes a alpha bath bin z zero

To modify precedes so that it also returns true when its two string arguments are equal, just change the operator in the final return statement to `<=` :

```
    if (s1.charAt(k) <= s2.charAt(k) )
      return true;
```

4

1. true
2. true
3. false
4. false
5. false
6. true
7. false
8. false
9. false

The variables in TestStringEquals are declared static because they are used in static methods. Whenever you call a method directly from main(), it must be static because main() is static. Remember that static elements are associated with the class, not with its instances. So main() can only use static elements because they don't depend on the existence of instances.

1. `String s3 = s1.substring(s1.indexOf('n'))`
   `+ s1.substring(0,s1.indexOf('n'));`
2. `String s4 = s2.substring(6) + " " + s2.substring(0,5);`
3. `String s5 = s2.substring(0,6) + s1.substring(0,3);`

Explain the difference between the following pairs of terms:

Unit indexing and *zero indexing*.

Data structure and *data type*.

StringBuffer and String.

String and StringTokenizer.

Declaring a variable and *instantiating a* String.

A Font and a FontMetrics object.

Fill in the blanks.

=14pt

When the first character in a string has index 0, this is known as

---

.

A sequence of characters enclosed within quotes is known as a

---

.

=11pt

Given the String *str* with the value "to be or not to be that is the question," write Java expressions to extract each of the substrings shown below. For each substring, provide two sets of answers. One that uses the actual index numbers of the substrings—for example, the first "to" goes from 0 to 2—and a second more general solution that will also retrieve the substring from the following string "it is easy to become what you want to become." (*Hint*: In the second case, use length() and indexOf() along with substring() in your expressions. If necessary, you may use local variables to store intermediate results. The answer to (a) is provided as an example.)

the first "to" in the string

```
str.substring(0, 2)                          // Answer 1
str.substring(
     str.indexOf("to"), str.indexOf("to") + 2) // Answer 2
```

2

the last "to" in the string

the first "be" in the string

the last "be" in the string

the first four characters in the string

the last four characters in the string

Identify the syntax errors in each of the following, assuming that s is the literal string "exercise":

2

```
s.charAt("hello")
```

```
s.indexOf(10)
s.substring("er")
s.lastIndexOf(er)
s.length
```

Evaluate each of the following expressions, assuming that s is the literal string “exercise”:

2

```
s.charAt(5)
s.indexOf("er")
s.substring(5)
s.lastIndexOf('e')
s.length()
```

Write your own equalsIgnoreCase() method using only other String methods.

Write your own String equality method without using String. . (*Hint*: Modify the precedes() method.)

Even though Java’s String class has a built-in toLowerCase() method, write your own implementation of this method. It should take a String parameter and return a String with all its letters written in lowercase.

Write a method that converts its String parameter so that letters are written in blocks five characters long. For example, consider the following two versions of the same sentence:

```
Plain :   This is how we would ordinarily write a sentence.
Blocked : Thisi showw ewoul dordi naril ywrit easen tence.
```

Design and implement a Java Swing program that lets the user type a document into a TextArea and then provides the following analysis of the document: the number of words in the document, the number of characters in the document, and the percentage of words that have more than six letters.

Design and write a Java Swing program that searches for single-digit numbers in a text and changes them to their corresponding words. For example, the string “4 score and 7 years ago” would be converted into “four score and seven years ago”.

A palindrome is a string that is spelled the same way backward and forward. For example, *mom, dad, radar, 727* and *able was i ere i saw elba* are all examples of palindromes. Write a Java Swing program that lets the user type in a word or phrase and then determines whether the string is a palindrome.

Write a maze program that uses a string to store a representation of the maze. Write a method that accepts a String parameter and prints a two-dimensional representation of a maze. For example, the maze shown here, where O marks the entrance and exit can be generated from the following string:

```
String: XX_XXXXXXXX__XXX_XXXX_XX____XXX_XX_XX_XXX____X____XXXXXXXX_X
  O
XX XXXXXXX
X  XXX XXX
X XX    XX
X XX XX XX
X    X    O
XXXXXXXX X
```

Write a method that takes a delimited string to store a name and address, from which you can print a mailing label. For example, if the string contains “Sam Penn:14 Bridge St.:Hoboken, NJ 01881,” the method should print the label shown in the margin.

Design and implement a Java Swing program that plays Time Bomb with the user. Here's how the game works. The computer picks a secret word and then prints one asterisk for each letter in the word: * * * * *. The user guesses at the letters in the word. For every correct guess, an asterisk is replaced by a letter:e * * *. For every incorrect guess, the time bomb's fuse grows shorter. When the fuse disappears, after say, six incorrect guesses, the bomb explodes. Store the secret words in a delimited string and invent your own representation for the time bomb.

Challenge: The global replace function is a string-processing algorithm found in every word processor. Write a method that takes three String arguments: a document, a target string, and a replacement string. The method should replace every occurrence of the target string in the document with the replacement string. For example, if the document is "To be or not to be, that is the question," and the target string is "be,", and the replacement string is "see," the result should be, "To see or not to see, that is the question."

Challenge: Design and implement a Java Swing Program that plays the following game with the user. Let the user pick a letter between *A* and *Z*. Then let the computer guess, the secret letter. For every guess the player has to tell the computer whether it's too high or too low. The computer should be able to guess the letter within five guesses. Do you see why?

Challenge: A *list* is a sequential data structure. Design a List class that uses a comma-delimited String—such as, "a,b,c,d,12,dog"—to implement a list. Implement the following methods for this class:

```
void addItem( Object o );      // Use Object.toString()
String getItem(int position);
String toString();
void deleteItem(int position);
void deleteItem(String item);
int getPosition(String item);
String getHead();              // First element
List getTail();                // All but the first element
int length();                  // Number of items
```

Challenge: Use a delimited string to create a PhoneList class with an instance method to insert names and phone numbers, and a method to look up a phone number when a user provides a person's name. Since your class will take care of looking things up, you don't have to worry about keeping the list in alphabetical order. For example, the following string could be used as such a directory:

-4pc

```
mom:860-192-9876::bill g:654-0987-1234::mary lancelot:123-842-1100
```

Design and implement an application that displays a multi-line message in various fonts and sizes input by the user. Let the user choose from among a fixed selection of fonts, sizes, and styles.

This page titled 7.11: Handling Text in a Graphics Context (Optional) is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **7.11: Handling Text in a Graphics Context (Optional)** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

CHAPTER OVERVIEW

8: Arrays and Array Processing

8.1: Introduction
8.2: One-Dimensional Arrays
8.3: Simple Array Examples
8.4: Example- Counting Frequencies of Letters
8.5: Array Algorithms- Sorting
8.6: Array Algorithms- Searching
8.7: Two-Dimensional Arrays
8.8: Multidimensional Arrays (Optional)
8.9: Object-Oriented Design- Polymorphic Sorting (Optional)
8.10: From the Java Library- java.util.Vector
8.11: Case Study- An N-Player Computer Game
8.12: A GUI-Based Game (Optional Graphics)

This page titled 8: Arrays and Array Processing is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

8.1: Introduction

Arrays and Array Processing

After studying this chapter, you will

Know how to use array data structures.

Be able to solve problems that require collections of data.

Know how to sort an array of data.

Be familiar with sequential and binary search algorithms.

Gain a better understanding of inheritance and polymorphism.

Introduction

One-Dimensional Arrays

Simple Array Examples

Example: Counting Frequencies of Letters

Array Algorithms: Sorting

Array Algorithms: Searching

Two-Dimensional Arrays

Multidimensional Arrays (Optional)

Object-Oriented Design: Polymorphic Sorting (Optional)

From the Java Library: java.util.Vector

Case Study: An N-Player Computer Game

A GUI-Based Game (Optional Graphics)

Chapter Summary

Solutions to Self-Study Exercises

Exercises

Introduction

In this chapter we will learn about arrays. An **array** is a named collection of contiguous storage locations—storage locations that are next to each other—that contain data items of the same type.

Arrays offer a more streamlined way to store data than using individual data items for each variable. Arrays also allow you to work with their data more efficiently than with data stored in individual variables.

Let's see why. Suppose you want to create a GUI that has 26 buttons on it, one for each letter of the alphabet. Given our present knowledge of Java, our only alternative would be to declare a separate JButton variable for each letter of the alphabet:

```
JButton button1;
JButton button2;
...
JButton button26;
```

Obviously, requiring 26 separate variables for this problem is tedious and inconvenient. Similarly, to instantiate and assign a label to each button would require 26 statements:

```
button1 = new JButton("A");
button2 = new JButton("B");
...
button26 = new JButton("Z");
```

This approach is also tedious. What we need is some way to use a loop to process each button, using a loop counter, *k*, to refer to the *k*th button on each iteration of the loop. An array lets us do that. For example, the following code will declare an array for storing 26 JButtons and then instantiate and label each button:

```
JButton letter[]  = new JButton[26];
for (int k = 0; k < 26; k++)
    letter[k] = new JButton("A" + k);
```

You don't yet understand the code in this segment, but you can see how economical it is. It uses just three lines of code to do what would have required 50 or 60 lines of code without arrays.

Our discussion of arrays will show how to store and retrieve data from one-, two-, and three-dimensional arrays. We also study sorting and searching algorithms to process arrays. Finally, we illustrate how arrays can be used in a variety of applications, including an animation problem, a sorting class, and a card-playing

One-Dimensional Arrays

An array is considered a *data structure*. A **data structure** is an organized collection of data. In an array, data are arranged in a linear or sequential structure, with one element following another. When referencing elements in an array, we refer to the position of the particular element within the array. For example, if the array is named arr, then the elements are named `arr[0], arr[1], arr[2], ... arr[n-1]` , where *n* gives the number of elements in the array. This naming also reflects the fact that the array's data are contained in storage locations that are next to each other. In Java, as in C, C++, and some other programming languages, the first element of an array has index 0. (This is the same convention we used for Strings.)

Figure [fig-intarray] shows an array named arr that contains 15 int elements. The syntax for referring to elements of an array is

where *arrayname* is the name of the array—any valid identifier will do—and *subscript* is the position of the element within the array. As Figure [fig-intarray] shows, the first element in the array has subscript 0, the second has subscript 1, and so on.

A **subscript** is an integer quantity contained in square brackets that is used to identify an array element. An subscript must be either an integer value or an integer expression. Using Figure [fig-intarray] to illustrate an example, suppose that *j* and *k* are integer variables equaling 5 and 7, respectively. Each of the following then would be valid references to elements of the array *arr*:

```
arr[4]     //Refers to 16
arr[j]     //Is arr[5] which refers to 20
arr[j + k] //Is arr[5+7] which is arr[12] which refers to 45
arr[k % j] //Is arr[7%5] which is arr[2] which refers to -1
```

These examples show that when an expression, such as j + k, is used as a subscript, it is evaluated (to 12 in this case) before the reference is made.

It is a syntax error to use a noninteger type as an array subscript. Each of the following expressions would be invalid:

```
arr[5.0]  // 5.0 is a float and can't be an array subscript
arr["5"]  // "5" is a string not an integer
```

For a given array, a valid array subscript must be in the range `0 ... N-1` , where N is the number of elements in the array or it is considered out-of-bounds. An out-of-bounds subscript creates a run-time error—that is, an error that occurs when the program is running—rather than a syntax error, which can be detected when the program is compiled. For the array arr, each of the following expressions contain out-of-bounds subscripts:

```
arr[-1]  // Arrays cannot have negative subscripts
arr['5'] // Char '5' promoted to its Unicode value, 53
arr[15]  // The last element of arr has subscript 14
arr[j*k] // Since j*k equals 35
```

Each of these references would lead to an IndexOutOfBoundsException. (Exceptions are covered in detail in Chapter 10.)

Declaring and Creating Arrays

For the most part, arrays in Java are treated as objects. Like objects, they are instantiated with the new operator and they have instance variables (for example, length). Like variables for objects, array variables are considered *reference* variables. When arrays are used as parameters, a reference to the array is passed rather than a copy of the entire array. The primary difference between arrays and full-fledged objects is that arrays aren't defined in terms of an Array class. Thus, arrays don't fit into Java's Object hierarchy. They don't inherit any properties from Object and they cannot be subclassed.

You can think of an array as a container that contains a number of variables. As we've seen, the variables contained in an array object are not referenced by name but by their relative position in the array. The variables are called *components*. If an array object has *N* components, then we say that the **array length** is *N*. Each of the components of the array has the same type, which is called the array's *component type*. An *empty* array is one that contains zero variables.

A **one-dimensional** array has components that are called the array's **elements**. Their type is the array's **element type**. An array's elements may be of any type, including primitive and reference types. This means you can have arrays of int, char, boolean, String, Object, Image, TextField, TwoPlayerGame, and so on.

When declaring a one-dimensional array, you have to indicate both the array's element type and its length. Just as in declaring and creating other kinds of objects, creating an array object requires that we create both a name for the array and then the array itself. The following statements create the array shown in Figure [fig-intarray]:

```
int arr[];          // Declare a name for the array
arr = new int[15];  // Create the array itself
```

These two steps can be combined into a single statement as follows:

```
int arr[] = new int[15];
```

In this example, the array's element type is int and its length is 15, which is fixed and cannot be changed. This means that the array contains 15 variables of type int, which will be referred to as arr[0], arr[1], …arr[14].

Array Allocation

Creating the array in Figure [fig-intarray] means allocating 15 storage locations that can store integers. Note that one difference between declaring an array and declaring some other kind of object is that square brackets () are used to declare an array type. The brackets can be attached either to the array's name or to its type, as in the following examples:

```
int arr[];     // The brackets may follow the array's name
int[] arr;     // The brackets may follow the array's type
```

The following example creates an array of five Strings and then uses a for loop to assign the strings "hello1", "hello2", "hello3", "hello4", and "hello5" to the five array locations:

```
String strarr[];               // Declare a name for the array
strarr = new String[5];             // Create the array itself
                               // Assign strings to the array
for (int k = 0; k < strarr.length; k++)     // For each element
  strarr[k] = new String("hello" + (k + 1)); // Assign a string
```

Note that the expression `k < strarr.length` specifies the loop bound. Every array has a length instance variable, which refers to the number of elements contained in the array. As we mentioned, arrays, like Strings, are zero indexed, so the last element of the array is always given by its length-1. However, length is an instance variable for arrays, whereas length() is an instance method for Strings. Therefore, it would be a syntax error in this example to refer to strarr.length().

In the example, we first use the new operator to create strarr, an array of type String of length five. We then use a String constructor to create the five Strings that are stored in the array. It is important to realize that creating an array to store five Objects (as opposed to five primitive data elements) does not also create the Objects themselves that will be stored in the array.

When an array of objects is created, the array's elements are references to those objects (Fig. [fig-arrayobjs]). Their initial values, like all reference variables, are null. So to create and initialize the array strarr, we need to create *six* objects—the array itself, which will contain five Strings, and then the five Strings that are stored in strarr.

One more example will help underscore this point. The following statements create four *new* Objects, an array to store three Students plus the three Students themselves:

```
Student school[] = new Student[3];   // A 3 Student array
school[0] = new Student("Socrates"); // The first Student
school[1] = new Student("Plato");    // The second Student
school[2] = new Student("Aristotle");// The third Student
```

The first statement creates an array named school to store three s, and the next three statements create the individual Students and assign them to the array (Fig. 9.3). Thus, creating the array and initializing its elements require four new statements.

The following sequence of statements would lead to a null pointer exception because the array's elements have not been instantiated:

```
Student students[] = new Student[3]; // A 3 Student array
System.out.println(students[0].getName());
```

In this case, students[0] is a null reference, thus causing the exception.

Now that we've assigned the three Students to the array, we can refer to them by means of subscripted references. A reference to the named "Socrates" is now school[0], and a reference to the named "Plato" is school[1]. In other words, to refer to the three individual students we must refer to their locations within school. Of course, we can also use variables, such as loop counters, to refer to a Student's location within school. The following for loop invokes each Student's getState() method to print out its current state:

```
for (int k = 0; k < school.length; k++)
    System.out.println(school[k].getState());
```

What if the three Students already existed before the array was created? In that case, we could just assign their references to the array elements, as in the following example:

```
Student student1 = new Student("Socrates");
Student student2 = new Student("Plato");
Student student3 = new Student("Aristotle");
Student school = new Student[3]; // A 3 Student array
school[0] = student1;
school[1] = student2;
school[2] = student3;
```

In this case, each of the three Student objects can be referenced by two different references—its variable identifier (such as student1) and its array location (such as school[0]). For arrays of objects, Java stores just the reference to the object in the array

itself, rather than the entire object. This conserves memory, since references require only 4 bytes each whereas each object may require hundreds of bytes (Fig. 9.4).

When an array of *N* elements is created, the compiler *allocates* storage for *N* variables of the element's type. In the case of arr that we discussed earlier, the compiler would allocate storage for 15 ints—60 contiguous bytes of storage, because each int requires 4 bytes (32 bits) of storage. If we declare an array of 20 doubles,

```
double arr[] = new double[20];
```

the compiler will allocate 160 bytes of storage—20 variables of 8 bytes (64 bits) each. In the case of the examples and String examples, because these are objects (not primitive types), the compiler will allocate space for *N* addresses, where *N* is the length of the array and where each address requires 4 bytes.

How much space (in bytes) would be allocated for each of the following?

int a[] = new int[5];

double b[] = new double[10];

char c[] = new char[30];

String s[] = new String[10];

Student p[] = new Student[5];

Initializing Arrays

Array elements are automatically initialized to default values that depend on the element type: Boolean elements are initialized to false, and integer and real types are initialized to 0. Reference types—that is, arrays of objects—are initialized to null.

Arrays can also be assigned initial values when they are created, although this is feasible only for relatively small arrays. An **array initializer** is written as a list of expressions separated by commas and enclosed by braces. For example, we can declare and initialize the array shown in Figure [fig-intarray] with the following statement:

```
int arr[] = {-2,8,-1,-3,16,20,25,16,16,8,18,19,45,21,-2};
```

Similarly, to create and initialize an array of Strings, we can use the following statement:

```
String strings[] = {"hello", "world", "goodbye", "love"};
```

This example creates and stores four Strings in the array. Subsequently, to refer to "hello", we would use the reference strings[0], and to refer to "love", we would use the reference strings[3]. Note in these examples that when an array declaration contains an initializer, it is not necessary to use new and it is not necessary to specify the number of elements in the array. The number of elements is determined from the number of values in the initializer list.

Assigning and Using Array Values

Array elements can be used in the same way as other variables. The only difference, of course, is that references to the elements are subscripted. For example, the following assignment statements assign values to the elements of two arrays, named arr and strings:

```
arr[0] = 5;
arr[5] = 10;
arr[2] = 3;
strings[0] = "who";
strings[1] = "what";
strings[2] = strings[3] = "where";
```

The following loop assigns the first 15 squares—1, 4, 9 …—to the array arr:

```
for (int k = 0; k < arr.length; k++)
    arr[k] = (k+1) * (k+1);
```

The following loop prints the values of the array arr:

```
for (int k = 0; k < arr.length; k++)
    System.out.println(arr[k]);
```

Declare an array named farr that contains ten floats initialized to the values 1.0, 2.0, …, 10.0.

Write an expression that prints the first element of farr.

Write an assignment statement that assigns 100.0 to the last element in farr.

Write a loop to print all of the elements of farr.

Simple Array Examples

The program in Figure [fig-initarrays] creates two arrays of ten elements each and displays their values on the Java console. In this

```
public class PrintArrays {
  static final int ARRSIZE = 10;         // The array's size
  static int intArr[] = new int[ARRSIZE];// Create int array
  static double realArr[] = { 1.1, 2.2, 3.3, 4.4,
     5.5, 6.6, 7.7, 8.8, 9.9, 10.10 }; // And a double array

  public static void main(String args[]) {
    System.out.println("Ints \t Reals");  // Print a heading
                          // For each int and double element
    for (int k = 0; k < intArr.length; k++)
       System.out.println( intArr[k] + " \t " +
                                realArr[k]);  // Print them
    } // main()
} // PrintArrays
```

example, the elements of have not been given initial values whereas the elements of have been initialized. Note the use of the integer constant to store the arrays' size. By using the constant in this way, we do not have to use the literal value 10 anywhere in the program, thereby making it easier to read and to modify the program. If we want to change the size of the array that the program handles, we can just change the value of ARRSIZE. This is an example of the maintainability principle.

Note the use of the static qualifier throughout the PrintArrays class. This enables us to refer to the array and the other variables from within the main() method. [fig:printarraysout] If intArr were not declared static, we would get the compiler error attempt to make static use of a non-static variable. This use of static is justified mainly as a coding convenience rather than a principle of object-oriented design. The only examples we've seen so far in which static elements were a necessary design element were the use of static elements in the Math class—Math.PI and Math.sqrt()—and the use of static final variables in TwoPlayerGame—TwoPlayerGame.PLAYER_ONE.

For large arrays, it is not always feasible to initialize them in an initializer statement. Consider the problem of initializing an array with the squares of the first 100 integers. Not only would it be tedious to set these values in an initializer statement, it would also be error prone, since it is relatively easy to type in the wrong value for one or more of the squares.

The example in Figure [fig-squares100] creates an array of 50 integers and then fills the elements with the values 1, 4, 9, 16, and so on. It then prints the entire array.

```
public class Squares {
  static final int ARRSIZE = 50;     // The array's size
  static int intArr[] = new int[ARRSIZE]; // Instantiate
  public static void main(String args[]) {
    for (int k = 0; k < intArr.length; k++)// Initialize
      intArr[k] = (k+1) * (k+1);
    System.out.print("The first 50 squares are");
    for (int k = 0; k < intArr.length; k++) { // Print
      if (k % 5 == 0)             // For each 5th square
        System.out.println(" ");//   print a new line
      System.out.print( intArr[k] + " ");
    } // for
  } // main()
} // Squares
```

This example illustrates some important points about the use of array variables. The array's elements are individual storage locations. In this example, intArr has 50 storage locations. Storing a value in one of these variables is done by an assignment statement:

```
intArr[k] = (k+1) * (k+1);
```

[fig-squaresout] The use of the variable *k* in this assignment statement allows us to vary the location that is assigned on each iteration of the for loop. Note that in this example, *k* occurs as the array index on the left-hand side of this expression, while *k+1* occurs on the right-hand side as the value to be squared. The reason for this is that arrays are indexed starting at 0 but we want our table of squares to begin with the square of 1. So the square of some number *n+1* will always be stored in the array whose index is one less than the number itself—that is, *n*.

An array's length variable can always be used as a loop bound when iterating through all elements of the array:

```
for (int k = 0; k < intArr.length; k++)
    intArr[k] = (k+1) * (k+1);
```

However, it is important to note that the last element in the array is always at location `length-1` . Attempting to refer to intArr[length] would cause an IndexOutOfBoundsException because no such element exists.

Declare an array of 100 doubles and write a loop to assign the first 100 square roots to its elements. [Use Math.sqrt(double).]

Example: Counting Frequencies of Letters

Suppose you wish to write a program to help break a text message that has been encrypted with one of the historical ciphers that we have discussed in the two previous chapters. It is well known that historical ciphers often can be broken, that is, the plaintext can be found from the ciphertext, by examining the frequencies of letters and comparing them to the average frequencies of typical samples of plaintext. For example, E and T are the two most frequently used letters in the English language. So, in a ciphertext encrypted with a Caesar cipher, E and T are good guesses as the plaintext letter corresponding to the most frequent letter in a ciphertext message.

Let's write a program that will count how many times each of the 26 letters of the English language appears in a given string. There are a number of ways to design such a program depending on how flexible you wish the program to be. Let us keep this example simple by assuming that we will only be interested in counting occurrences of the letters A through Z and not of occurrences of spaces or punctuation marks. Assume further that we will change lowercase letters in our string sample to uppercase before counting letters and that we will want to print out the frequencies of letters to the console window. Finally, assume that, later in the chapter after we discuss sorting arrays, we will want to enhance our program so that it can print out the letter frequencies in order of increasing frequency.

A Class to Store the Frequency of One Letter

It is clear that an array should be used for storing the frequencies, but a decision must also be made as to what to store as the array elements. If we store letter frequencies as int values, with the frequency of *A* stored at index 0, and the frequency of *B* at index 1, and so forth, we will not be able to rearrange the frequencies into increasing order without losing track of which letter corresponds to which frequency. One way of solving this problem is to create an array of objects, where each object stores both a letter and its frequency.

So let us design a LetterFreq class that stores a letter in an instance variable of type char and its frequency in an instance variable of type int. These instance variables can be declared as:

```
private char letter;    //A character being counted
private int freq;       //The frequency of letter
```

We will want a constructor that can initialize these two values and two accessor methods to return these values. We are familiar enough with these kinds of methods that it will not be necessary to discuss them any further. We need one additional method to increment freq whenever we encounter the letter while processing the string:

```
public void incrFreq() {
    freq++;
} //setFreq()
```

A UML diagram for the LetterFreq class is given in Figure 9.9 and the class definition is given

```
public class LetterFreq {
    private char letter;    //A character being counted
    private int freq;       //The frequency of letter

    public LetterFreq(char ch, int fre) {
        letter = ch;
        freq = fre;
    }
    public char getLetter() {
        return  letter;
    }
    public int getFreq() {
        return  freq;
    }
    public void incrFreq() {
        freq++;
    }
} //LetterFreq
```

in Figure [fig-letfreq]. Note that we will have to make a minor modification to this class later in this chapter to enable us to sort an array of objects from this class.

A Class to Count Letter Frequencies

Now let us turn to designing a class named AnalyzeFreq that will use an array of objects of type LetterFreq to count the frequencies of the letters A through Z in a given string. The array, let's call it freqArr, will be the only instance variable of the class. The class needs a constructor to instantiate the array and to create the 26 array elements, each with a different letter and an initial frequency of 0. This class should also have two methods: a method to count the frequencies of the 26 letters in a given string and a method that prints out the frequency of each letter to the console window. The UML diagram for the class is given in Figure 9.11.

The array instance variable can be declared by:

```
  private LetterFreq[] freqArr; //An array of frequencies
```

The constructor creates an array of 26 elements to store references to LetterFreq objects with the statement

```
  freqArr = new LetterFreq[26];
```

The indices of the array range from 0 to 25 and the elements at these locations should store the letters A to Z. Recall that in Java, char data are a form of int data and can be used in arithmetic. If we let k be an integer that ranges between 0 and 25, then the expression (char)('A' + k) will correspond to the letters A to Z. . Thus, the following loop will initialize the array correctly.

```
for (int k = 0; k < 26; k++) {
    freqArr[k] = new LetterFreq((char)('A' + k), 0);
} //for
```

The countLetters() method must identify the array index for LetterFreq object that stores a letter between A and Z. If let is a char variable that stores such a letter, then the expression (let - 'A') will give the index of the array element corresponding to let. Thus the following code will calculate the frequencies the letters in the string parameter, str:

```
 public void countLetters(String str) {
     char let;     //For use in the loop.
     str = str.toUpperCase();
     for (int k = 0; k < str.length(); k++) {
         let = str.charAt(k);
         if ((let >= 'A') && (let <= 'Z')) {
             freqArr[let - 'A'].incrFreq();
         } //if
     } //for
 } //countLetters()
```

The definition of the printArray() method is completely straight forward:

```
public void printArray() {
  for (int k = 0; k < 26; k++) {
    System.out.print("letter: " + freqArr[k].getLetter());
    System.out.println("   freq: " + freqArr[k].getFreq());
  } //for
} //printArray()
```

The entire definition of AnalyzeFreq is

```
public class AnalyzeFreq {
  private LetterFreq[] freqArr; //An array of frequencies

  public AnalyzeFreq() {
    freqArr = new LetterFreq[26];
    for (int k = 0; k < 26; k++) {
      freqArr[k] = new LetterFreq((char)('A' + k), 0);
    } //for
  }
  public void countLetters(String str) {
    char let; //For use in the loop.
    str = str.toUpperCase();
    for (int k = 0; k < str.length(); k++) {
      let = str.charAt(k);
      if ((let >= 'A') && (let <= 'Z')) {
        freqArr[let - 'A'].incrFreq();
      } //if
    } //for
  }
  public void printArray() {
    for (int k = 0; k < 26; k++) {
      System.out.print("letter: " + freqArr[k].getLetter());
      System.out.println(" freq: " + freqArr[k].getFreq());
    } //for
  }
} //AnalyzeFreq
```

given in Figure [fig-anafreq]. We will modify this class later in the chapter to be able to sort the array after counting. The following main() method, either in this class or in its own class will demonstrate how the class methods are used.

```
public static void main(String[] argv) {
  AnalyzeFreq af = new AnalyzeFreq();
  af.countLetters("Now is the time for all good students" +
     " to study computer related topics.");
  af.printArray();
} //main()
```

Rewrite the main() of the AnalyzeFreq class so that it opens a file named freqtest.txt and counts the frequencies of the letters of the text stored in the file. You will need to use the Scanner class to read from the file as was done in Chapter 4. Create a file named freqtest.txt that contains several hundred characters of typical English text to test the new main() method

Array Algorithms: Sorting

an array is the process of arranging its elements in ascending or descending order. Sorting algorithms are among the most widely used algorithms. Any time large amounts of data are maintained, there is some need to arrange them in a particular order. For example, the telephone company needs to arrange its accounts by the last name of the account holder as well as by phone number.

Insertion Sort

The first sorting algorithm we'll look at is known as **insertion sort**, so named because as it traverses through the array from the first to the last element, it inserts each element into its correct position in the partially sorted array.

For an array of *N* elements, let's think of the array as divided into two parts. The *sorted* part will be the left hand side of the array. And the *unsorted* part will be the right hand side of the array. Initially, the sorted part consists of the first element in the array—the element at index 0.

Insertion sort moves through the unsorted portion of the array—that is its loop variable, *k*, ranges from 1 through *N-1*. On each iteration it inserts the *k*th element into its correct position in the sorted part of the array. To insert an element into the sorted part of the array, it may be necessary to move elements greater than the one being inserted out of the way.

In pseudocode, insertion sort can be represented as follows:

-6.5pc

```
Insertion Sort of an array, arr, of N elements into ascending order
1. For k assigned 1 through N-1
2.   Remove the element arr[k] and store it in x.
3.   For i starting at k-1 and for all preceding elements greater than x
4.     Move arr[i] one position to the right in the array.
5.   Insert x at its correct location.
```

As is apparent from the pseudocode, we have a nested for loops. The outer (*k*) loop, iterates through the array from 1 to *N-1*. The inner loop iterates as many times as necessary, starting with the element just to the left of the *k*th element in order to insert the *k*th element into its correct position in the sorted portion. Note that the *k*th element is always removed from the array (and stored in the variable *x*), to make room for elements that have to be moved to the right.

To see how this works, consider an integer array containing the ages of five friends:

```
21 |  20  27  24  19      x = 20
      k
```

For this five-element array, insertion sort initially will assume that the element at index 0 is in the correct position. The vertical line marks the boundary between the sorted and unsorted portions of the array. The outer loop will look at each of the remaining elements, one at a time, inserting it into its proper position in the sorted portion of the array. To insert 20, the number at index 1, the inner loop will move 21 to the right by one position. To do this, the algorithm will remove 20 from its location and store it in x. It will then move 21 one space to the right. Finally, it will insert 20, which is stored in x, at index 0, where it belongs relative to the other elements in the sorted part of the array. At this point, the sorted portion of the array consists of the first two elements, which are in the correct order, relative to each other.

```
20  21 |  27  24  19     x = 27
         k
```

For the next element, 27, none of elements in the sorted portion need to be moved, so the inner for loop will iterate zero times. This gives us:

```
20  21  27 |  24  19    x = 24
            k
```

For the fourth element, 24, only the previous element, 27, needs to be moved to the right, giving:

```
20  21  24  27 | 19    x = 19
               k
```

At this point, the sorted part of the array consists of the first four elements, which are in the correct order relative to each other. Finally, for the last element, 19, all of the elements in the sorted part of the array need to be moved one space to the right. This will require four iterations of the inner loop. We show the state of the array after each iteration of the inner for loop:

```
               k
20 21 24 27 | 19   Remove 19 and store it x = 19
20 21 24 27 | 27   Move 27 to the right
20 21 24 24 | 27   Move 24 to the right
20 21 21 24 | 27   Move 21 to the right
20 20 21 24 | 27   Move 20 to the right
19 20 21 24 27 |   Insert x=19 at index 0
```

Clearly, the fact that so many elements may have to moved on each iteration of the outer loop shows that insertion sort is not a very efficient algorithm.

The Sort class (Fig [fig-bubblemeth]) provides an implementation of the insertionSort() method. There are several points worth noting about this code. First, because it takes an int array as a parameter, the insertionSort() method will sort any array of integers, regardless of the array's length.

```
public class Sort {
  public void insertionSort(int arr[]) {
    int temp;  // Temporary variable for insertion
    for (int k = 1; k < arr.length; k++)  {
      temp = arr[k]; // Remove element from array
      int i;         // For larger preceding elements
      for (i = k-1; i >= 0 && arr[i] > temp; i--)
        arr[i+1] = arr[i]; // Move it right by one
      arr[i+1] = temp;       // Insert the element
    }
  } // insertionSort()
  public void print(int arr[]) {
    for (int k = 0; k < arr.length; k++)// For each integer
      System.out.print( arr[k] + " \t "); //  Print it
    System.out.println();
  } // print()
  public static void main(String args[]) {
    int intArr[] = { 21, 20, 27, 24, 19 };
    Sort sorter = new Sort();
    sorter.print(intArr);
    sorter.insertionSort(intArr); // Passing an array
    sorter.print(intArr);
  } // main()
} //Sort
```

Second, note how empty brackets () are used to declare an array parameter. If the brackets were omitted, then arr would be indistinguishable from an ordinary int parameter. Using the brackets indicates that this method takes an array of integers as its parameter.

Third, note how an array of integers is passed to the insertionSort() method in the main() method:

```
 sorter.insertionSort(intArr); // Pass intArr to the method
```

That is, when passing an array to a method, you use just the name of the array, without brackets. Both of the following statements would cause syntax errors:

```
sorter.insertionSort(intArr[]); // Err: Can't have brackets
sorter.insertionSort(intArr[5]);// Err: passing an integer
```

In the first case, empty brackets are only used when you declare an array variable, not when you are passing the array to a method. In the second case, intArr[5] is an int, not an array, and cannot legally be passed to insertionSort().

Finally, within the insertionSort() method itself, note that we declare the index for the inner for loop outside of the for statement. This is so it can be used outside the scope of the for loop to insert temp at location arr[i+1], its correct location. Note also that the index of its correct location is i+1, rather than just i. This is because the inner loop might iterate past location 0, which would give *i* a value of -1 at that point.

Selection Sort

There are a large variety of array sorting algorithms. Selection sort is different from, but comparable to, insertion sort in its overall performance. To illustrate the **selection sort** algorithm, suppose you want to sort a deck of 25 index cards, numbered from 1 to 25. Lay the 25 cards out on a table, one card next to the other. Starting with the first card, look through the deck and find the smallest card, the number 1 card, and exchange it with the card in the first location. Then, go through the deck again starting at the second card, find the next smallest card, the number 2 card, and exchange it with the card in the second location. Repeat this process 24 times.

Translating this strategy into pseudocode gives the following

```
Selection sort of a 25-card deck from small to large
1. For count assigned 1 to 24  // Outer loop
2.   smallestCard = count
3.   For currentCard assigned count+1 to 25 // Inner loop
4.     If deck[currentCard] < deck[smallestCard]
5.        smallestCard = currentCard
6.   If smallestCard != count // You need to swap
7       Swap deck[count] and deck[smallestCard]
```

For a deck of 25 cards, you need to repeat the outer loop 24 times. In other words, you must select the smallest card and insert it in its proper location 24 times. The inner loop takes care of finding the smallest remaining card.

On each iteration of this outer loop, the algorithm assumes that the card specified by the outer loop variable, count, is the smallest card (line 2). It usually won't be, of course, but we have to start somewhere.

The inner loop then iterates through the remaining cards (from count+1 to 25) and compares each one with the card that is currently the smallest (lines 4 and 5). Whenever it finds a card that is smaller than the smallest card, it designates it as the smallest card (line 5). At the end of the loop, the smallestCard variable will remember where the smallest card is in the deck.

Finally, when the inner loop is finished, the algorithm swaps the smallest card with the card in the location designated by count.

Algorithm: Swapping Memory Elements

An important feature of the selection sort algorithm is its need to swap two array elements, or cards, to continue our example. Swapping two memory elements, whether they are array elements or not, requires the use of a temporary variable. For example, to swap the *j*th and *k*th elements in an int array named arr, you would use the following algorithm:

```
int temp = arr[j]; // Store the jth element in temp
arr[j] = arr[k];   // Move the kth element into j
arr[k] = temp;     // Move the jth element into k
```

The temp variable temporarily stores the *j*th element so its value is not lost when its location is overwritten by the *k*th element. The need for this variable is a subtlety that beginning programmers frequently overlook. But consider what would happen if we used the following erroneous algorithm:

```
arr[j] = arr[k]; // Erroneous swap code
arr[k] = arr[j];
```

If arr[j] refers to 4 and arr[k] refers to 2 in the array , then the erroneous algorithm would produce , the wrong result.

The following method implements the swap algorithm for two elements, *el1* and *el2* of an int array:

```
void swap(int arr[], int el1, int el2) {
    int temp = arr[el1]; // Assign first element to temp
    arr[el1] = arr[el2]; // Overwrite first with second
    arr[el2] = temp;     // Overwrite second with first
} // swap()
```

Sort the array, 24 18 90 1 0 85 34 18, using the insertion sort algorithm. Show the order of the elements after each iteration of the outer loop.

Sort the array, 24 18 90 1 0 85 34 18, using the selection sort algorithm. Show the order of the elements after each iteration of the outer loop.

Write a Java code segment to swap two Student objects, student1 and student2.

Write a Java implementation of the selectionSort() method to sort an array of int.

Passing a Value and Passing a Reference

Recall from Chapter 3 that when an Object is passed to a method, a copy of the reference to the Object is passed. Because an array is an object, a reference to the array is passed to insertionSort(), rather than the whole array itself. This is in contrast to how a value of a primitive type is passed. In that case, a copy of the actual value is passed.

One implication of this distinction is that for arguments of primitive type, the original argument cannot be changed from within the method because the method has only a copy of its value. For example, the following method takes an int parameter *n*, which is incremented within the method:

```
public void add1(int n) {
  System.out.print("n = " + n);
  n = n + 1;
  System.out.println(",  n = " + n);
}
```

But because *n* is a parameter of primitive type, incrementing it within the method has no effect on its associated argument. Thus, in the following segment, the value of *Num*—*n*'s associated argument—will not be affected by what goes on inside the add() method. The output produced by the code segment is shown in the comments:

```
int Num = 5;
System.out.println("Num = " + Num); // Prints Num = 5
add1(Num);                         // Prints n = 5,  n = 6
System.out.println("Num = " + Num); // Prints Num = 5
```

Note that while *n*'s value has changed inside the method, *Num*'s value remains unaffected.

The case is much different when we pass a reference to an object. In that case, the object itself can be manipulated from within the method. The insertionSort() method is a good illustration. In the following code segment, the array anArr is printed, then sorted, and then printed again:

```
 Sort sorter = new Sorter();
 int anArr[] = { 5, 10, 16, -2, 4, 6, 1 };
 sorter.print(anArr);           // Prints 5 10 16 -2 4 6 1
 sorter.insertionSort(anArr);   // Sorts anArr
 sorter.print(anArr);           // Prints -2 1 4 5 6 10 16
```

As you can see, the object itself (the array) has been changed from within the method. This shows that changes within insertionSort to the array referenced by arr are actually being made to anArr itself. If fact, because insertionSort() is passed a copy of the reference variable anArr, both arr and anArr are references to the very same object—that is, to the same array (Fig. [fig-arrayparam]).

The justification for passing a reference to an object rather than the entire object itself is a matter of efficiency. A reference uses just 4 bytes of data, whereas an object may use thousands of bytes. It would just be too inefficient to copy hundreds of bytes each time an object is passed to a method. Instead, the method is passed a reference to the object, thereby giving it access to the object without incurring the expense of copying large amounts of data. Indeed, Java provides no way to pass a copy of an object to a method.

Give the values that will be stored in myArr and k after you invoke mystery(myArr, k), where myArr, k and mystery() are declared as follows:

```
int myArr[] = {1,2,3,4,5};    int k = 3;
void mystery(int a[], int m) {
    ++a[m];
    --m;
}
```

Array Algorithms: Searching

Suppose we have a large array and we need to find one of its elements. We need an algorithm to search the array for a particular value, usually called the *key*. If the elements of the array are not arranged in any particular order, the only way we can be sure to find the key, assuming it is in the array, is to search every element, beginning at the first element, until we find it.

Sequential Search

This approach is known as a **sequential search**, because each element of the array will be examined in sequence until the key is found (or the end of the array is reached). A pseudocode description of this algorithm is as follows:

```
1. For each element of the array
2.    If the element equals the key
3.       Return its index
4. If the key is not found in the array
5.    Return -1 (to indicate failure)
```

This algorithm can easily be implemented in a method that searches an integer array, which is passed as the method's parameter. If the key is found in the array, its location is returned. If it is not found, then -1 is returned to indicate failure.

The Search class (Figs. 9.15 and [fig-search]) provides the Java implementation of the sequentialSearch() method. The method takes two parameters: the array to be searched and the key to be searched for. It uses a for statement to examine each element of the array, checking whether it equals the key or not. If an element that equals the key is found, the method immediately returns that element's index. Note that the last statement in the method will only be reached if no element matching the key is found.

```
public class Search {

  public int sequentialSearch(int arr[], int key) {
    for (int k = 0; k < arr.length; k++)
      if (arr[k] == key)
        return k;
    return -1;           // Failure if this is reached
  } // sequentialSearch()

  public int binarySearch(int arr[], int key) {
    int low = 0;                // Initialize bounds
    int high = arr.length - 1;
    while (low <= high) {   // While not done
      int mid = (low + high) / 2;
      if (arr[mid] == key)
        return mid;             // Success
      else if (arr[mid] < key)
        low = mid + 1;        // Search top half
      else
        high = mid - 1;       // Search bottom half
    }  // while
    return -1;     // Post: if low > high search failed
  } // binarySearch()
}//Search
```

Binary Search

If the elements of an array have been sorted into ascending or descending order, it is not necessary to search sequentially through each element of the array in order to find the key. Instead, the search algorithm can make use of the knowledge that the array is ordered and perform what's known as a **binary search**, which is a divide-and-conquer algorithm that divides the array in half on each iteration and limits its search to just that half that could contain the key.

To illustrate the binary search, recall the familiar guessing game in which you try to guess a secret number between 1 and 100, being told "too high" or "too low" or "just right" on each guess. A good first guess should be 50. If this is too high, the next guess should be 25, because if 50 is too high the number must be between 1 and 49. If 50 was too low, the next guess should be 75, and so on. After each wrong guess, a good guesser should pick the midpoint of the sublist that would contain the secret number.

Proceeding in this way, the correct number can be guessed in at most $log_2 N$ guesses, because the base-2 logarithm of N is the number of times you can divide N in half. For a list of 100 items, the search should take no more than seven guesses ($2^7 = 128 > 100$). For a list of $1,000$ items, a binary search would take at most ten guesses ($2^{10} = 1,024 > 1,000$).

So a binary search is a much more efficient way to search, provided the array's elements are in order. Note that "order" here needn't be numeric order. We could use binary search to look up a word in a dictionary or a name in a phone book.

A pseudocode representation of the binary search is given as follows:

```
TO SEARCH AN ARRAY OF N ELEMENTS IN ASCENDING ORDER

1. Assign 0 low and assign N-1 to high initially
2. As long as low is not greater than high
3.    Assign (low + high) / 2 to mid
4.    If the element at mid equals the key
5.        then return its index
6.    Else if the element at mid is less than the key
7.        then assign mid + 1 to low
8.    Else assign mid - 1 to high
9. If this is reached return -1 to indicate failure
```

Just as with the sequential search algorithm, this algorithm can easily be implemented in a method that searches an integer array that is passed as the method's parameter (Fig. [fig-search]). If the key is found in the array, its location is returned. If it is not found, then -1 is returned to indicate failure. The binarySearch() method takes the same type of parameters as sequentialSearch(). Its local variables, low and high, are used as pointers, or references, to the current low and high ends of the array, respectively. Note the loop-entry condition: `low <= high` . If low ever becomes greater than high, this indicates that key is not contained in the array. In that case, the algorithm returns -1.

As a binary search progresses, the array is repeatedly cut in half and low and high will be used to point to the low and high index values in that portion of the array that is still being searched. The local variable mid is used to point to the approximate midpoint of the unsearched portion of the array. If the key is determined to be past the midpoint, then low is adjusted to mid+1; if the key occurs before the midpoint, then high is set to `mid-1` . The updated values of low and high limit the search to the unsearched portion of the original array.

Unlike sequential search, binary search does not have to examine every location in the array to determine that the key is not in the array. It searches only that part of the array that could contain the key. For example, suppose we are searching for -5 in the following array:

```
int sortArr[] =
   { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
```

The -5 is smaller than the smallest array element. Therefore, the algorithm will repeatedly divide the low end of the array in half until the condition `low > high` becomes true. We can see this by tracing the values that low, mid, and high will take during the search:

```
 Key  Iteration  Low   High    Mid
-----------------------------------
 -5   0          0     19      9
 -5   1          0     8       4
 -5   2          0     3       1
 -5   3          0     0       0
 -5   4          0     -1      Failure
```

As this trace shows, the algorithm examines only four locations to determine that -5 is not in the array. After checking location 0, the new value for high will become -1, which makes the condition `low <= high` false. So the search will terminate.

The TestSearch class (Figs. 9.17 and [fig-testsearch]) provides a program that can be used to test two search methods. It creates an integer array, whose values are in ascending order. It then uses the getInput() method to input an integer from the keyboard and then performs both a sequentialSearch() and a binarySearch() for the number.

For the array containing the elements 2, 4, 6, and so on up to 28 in that order, draw a trace showing which elements are examined if you search for 21 using a binary search.

-2pc

```
import java.io.*;
public class TestSearch {
  public static int getInput() {
    KeyboardReader kb = new KeyboardReader();
    kb.prompt("This program searches for values in an array.");
    kb.prompt(
    "Input any positive integer (or any negative to quit) : ");
    return kb.getKeyboardInteger();
  } // getInput()

  public static void main(String args[]) throws IOException {
    int intArr[] = { 2,4,6,8,10,12,14,16,18,20,22,24,26,28};
    Search searcher = new Search();
    int key = 0, keyAt = 0;
    key = getInput();
    while (key >= 0) {
      keyAt = searcher.sequentialSearch( intArr, key );
      if (keyAt != -1)
        System.out.println("  Sequential: " + key +
                           " is at intArr[" + keyAt + "]");
      else
        System.out.println("  Sequential: " + key
                         + " is not contained in intArr[]");
      keyAt = searcher.binarySearch(intArr, key);
      if (keyAt != -1)
        System.out.println("  Binary: " + key +
                           " is at intArr[" + keyAt + "]");
      else
        System.out.println("  Binary: " + key +
                          " is not contained in intArr[]");
      key = getInput();
    } // while
  } // main()
} // TestSearch
```

Two-Dimensional Arrays

A **two-dimensional array**, an array whose components are themselves arrays, is necessary or useful for certain kinds of problems. For example, you would use this type of array if you are doing a scientific study in which you have to track the amount of precipitation for every day of the year.

One way to organize these data would be to create a one-dimensional array, consisting of 365 elements:

```
double rainfall[] = new double[365];
```

However, with this representation, it would make it very difficult to calculate the average rainfall within a given month, which might be an important part of your study.

A better representation for this problem would be to use a two-dimensional array, one dimension for the months and one for the days. The following statement declares the array variable rainfall and creates a 12 by 31 array object as its reference:

```
double rainfall[][] = new double[12][31];
```

Thus, rainfall is an *array of arrays*. You can think of the first array as the 12 months required for the problem. And you can think of each month as an array of 31 days. The months will be indexed from 0 to 11, and the days will be indexed from 0 to 30.

The problem with this representation is that when we want to refer to the rainfall for January 5, we would have to use rainfall[0][4]. This is awkward and misleading. The problem is that dates—1/5/1999—are unit indexed, while arrays are zero indexed. Because it will be difficult to remember this fact, our representation of the rainfall data may cause us to make errors when we start writing our algorithms.

We can easily remedy this problem by just defining our array to have an extra month and an extra day each month:

```
double rainfall[][] = new double[13][32];
```

This representation creates an array with 13 months, indexed from 0 to 12, with 32 days per month, indexed from 0 to 31. However, we can simply ignore the 0 month and 0 day by using unit indexing in all of the algorithms that process the array. In other words, if we view this array as a two-dimensional table, consisting of 13 rows and 32 columns, we can leave row 0 and column 0 unused (Fig. [fig-rainfall]).

As Figure [fig-rainfall] shows, the very first element of this 416-element array has subscripts (0,0) while the last location has subscripts (12,31). The main advantages of this representation is that the program as a whole will be much easier to read and understand and much less prone to error.

In order to refer to an element in a two-dimensional array, you need to use two subscripts. For the rainfall array, the first subscript will specify the *month* and the second will specify the *day* within the month. Thus, the following statements assign 1.15 to the rainfall element representing January 5, and then print its value:

```
rainfall[1][5] = 1.15;   // Rainfall for January 5
System.out.println( rainfall[1][5] );
```

Just as in the case of one-dimensional arrays, it is an error to attempt to reference an element that is not in the array. Each of the following examples would cause Java to raise an IndexOutOfBoundsException:

```
rainfall[13][32] = 0.15 ;  // No such element
rainfall[11][33] = 1.3;    // No such column
rainfall[14][30] = 0.74;   // No such row
```

If the initial values of an array's elements are supposed to be zero, there is no need to initialize the elements. Java will do it automatically when you create the array with new. However, for many array problems it is necessary to initialize the array elements to some other value. For a two-dimensional array, this would require a nested loop. To illustrate this algorithm, let's use a nested for loop to initialize each element of the rainfall array to 0:

```
// Note that both loops are unit indexed.
for (int month = 1; month < rainfall.length; month++)
  for (int day = 1; day < rainfall[month].length; day++)
     rainfall[month][day] = 0.0;
```

Note that both for loops use unit indexing. This is in keeping with our decision to leave month 0 and day 0 unused.

Remember that when you have a nested for loop, the inner loop iterates faster than the outer loop. Thus, for each month, the inner loop will iterate over 31 days. This is equivalent to processing the array as if you were going across each row and then down to the next row in the representation shown in Figure [fig-rainfall].

Note that for a two-dimensional array, both dimensions have an associated length variable, which is used in this example to specify the upper bound of each for loop. For the rainfall array, the first dimension (months) has a length of 13 and the second dimension (days) has a length of 32.

Another way to view the rainfall array is to remember that it is an array of arrays. The length of the first array, which corresponds to the number (13) of months, is given by rainfall.length. The length of each month's array, which corresponds to the number of days (32) in a month, is given by rainfall[month].length.

The outer loop of the nested for loop iterates through months 1 through 12, and the inner for loop iterates through days 1 through 31. In this way, $372 = 12 \times 31$ elements of the array are set to 0.0. In Table 9.1, the boldface numbers along the top represent the day subscripts, while the boldface numbers along the left represent the month subscripts.

[2pt]	**0**	**1**	**2**	**3**	···	**30**	**31**
[-4pt]							
[2pt] **0**	–	–	–	–	···	–	–
[2pt] **1**	–	0.0	0.0	0.0	···	0.0	0.0
[2pt] **2**	–	0.0	0.0	0.0	···	0.0	0.0
[0pt] ⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
[4pt] **10**	–	0.0	0.0	0.0	···	0.0	0.0
[2pt] **11**	–	0.0	0.0	0.0	···	0.0	0.0
[2pt] **12**	–	0.0	0.0	0.0	···	0.0	0.0
[-4pt]							

Declare a two-dimensional array of int, named int2d, that contains five rows, each of which contains ten integers.

Write a statement that prints the last integer in the third row of the array that you created in the previous exercise. Then write an assignment statement that assigns 100 to the last element in the int2d array.

Write a loop to print all of the elements of int2d, which you declared in the previous exercise. Print one row per line with a space between each element on a line.

Two-Dimensional Array Methods

Now that we have figured out how to represent the data for our scientific experiment, let's develop methods to calculate some results. First, we want a method to initialize the array. This method will simply incorporate the nested loop algorithm we developed previously:

```
public void initRain(double rain[][]) {
    for (int month = 1; month < rain.length; month++)
        for (int day = 1; day < rain[month].length; day++)
            rain[month][day] = 0.0;
} // initRain()
```

Note how we declare the parameter for a multidimensional array. In addition to the element type (double), and the name of the parameter (rain), we must also include a set of brackets for *each* dimension of the array.

Note also that we use the parameter name within the method to refer to the array. As with one-dimensional arrays, the parameter is a reference to the array, which means that any changes made to the array within the method will persist when the method is exited.

The avgDailyRain() Method

One result that we need from our experiment is the average daily rainfall. To calculate this result, we would add up all of the rainfalls stored in the 12×31 array and divide by 365. Of course, the array itself contains more than 365 elements. It contains 416 elements, but we're not using the first month of the array, and within some months—those with fewer than 31 days—we're not using some of the day elements. For example, there's no such day as rainfall[2][30], which would represent February 30. However, because we initialized all of the array's elements to 0, the rainfall recorded for the non-days will be 0, which won't affect our overall average.

The method for calculating average daily rainfall should take our two-dimensional array of as a parameter, and it should return a . Its algorithm will use a nested for loop to iterate through the elements of the array, adding each element to a running total. When the loops exits, the total will be divided by 365 and returned:

```
public double avgDailyRain(double rain[][]) {
    double total = 0;
    for (int month = 1; month < rain.length; month++)
        for (int day = 1; day < rain[month].length; day++)
            total += rain[month][day];
    return total/365;
} // avgDailyRain()
```

The avgRainForMonth() Method

One reason we used a two-dimensional array for this problem is so we could calculate the average daily rainfall for a given month. Let's write a method to solve this problem. The algorithm for this method will not require a nested for loop. We will just iterate through the 31 elements of a given month, so the month subscript will not vary. For example, suppose we are calculating the average for January, which is represented in our array as month 1:

```
double total = 0;
for (int day = 1; day < rainfall[1].length; day++)
    total = total + rainfall[1][day];
```

Thus, the month subscript is held constant (at 1) while the day subscript iterates from 1 to 31. Of course, in our method we would use a parameter to represent the month, thereby allowing us to calculate the average daily rainfall for any given month.

Another problem that our method has to deal with is that months don't all have 31 days, so we can't always divide by 31 to compute the monthly average. There are various ways to solve this problem, but perhaps the easiest is to let the number of days for that month be specified as a third parameter. That way, the month itself and the number of days for the month are supplied by the user of the method:

```
public double avgRainForMonth(double rain[][],
                              int month, int nDays) {
    double total = 0;
    for (int day = 1; day < rain[month].length; day++)
        total = total + rain[month][day];
    return total/nDays;
} // avgRainForMonth()
```

Given this definition, we can call this method to calculate and print the average daily rainfall for March as in the following statement:

```
System.out.println("March: " +
                         avgRainForMonth(rainfall,3,31));
```

Note that when passing the entire two-dimensional array to the method, we just use the name of the array. We do not have to follow the name with

Passing Part of an Array to a Method

Instead of passing the entire rainfall array to the avgRainForMonth() method, we could redesign this method so that it is only passed the particular month that's being averaged. Remember that a two-dimensional array is an array of arrays, so if we pass the month of January, we are passing an array of 32 days. If we use this approach, we need only two parameters: the month, which is array of days, and the number of days in that month:

```
public double avgRainForMonth(double monthRain[],
                                          int nDays) {
    double total = 0;
    for (int day = 1; day < monthRain.length; day++)
        total = total + monthRain[day];
    return total/nDays;
} // avgRainForMonth()
```

Given this definition, we can call it to calculate and print the average daily rainfall for March as in the following statement:

```
System.out.println("March: " +
                     avgRainForMonth(rainfall[3],31));
```

In this case, we're passing an array of double to the method, but in order to reference it, we have to pull it out of the two-dimensional array by giving its *row* subscript as well. Thus, rainfall[3] refers to one month of data in the two-dimensional array, the month of March. But rainfall[3] is itself a one-dimensional array. Figure [fig-twodrow] helps to clarify this point.

It's important to note that deciding whether to use brackets when passing data to a method is not just a matter of whether you are passing an array. It is a matter of what type of data the method parameter specifies. So, whenever you call a method that involves a parameter, you have to look at the method definition to see what kind of data that parameter specifies. Then you must supply an argument that refers to that type of data.

For our two-dimensional rainfall array, we can refer to the entire array as rainfall. We can refer to one of its months as rainfall[j], where *j* is any integer between 1 and 12. And we can refer to any of its elements as rainfall[j][k], where *j* is any integer between 1 and 12, and *k* is any integer between 1 and 31.

The Rainfall class (Figs. 9.21 and [fig-rainfallclass])

```
public class Rainfall {
  /**
   * Initializes the rainfall array
   * @param rain is a 2D-array of rainfalls
   * Pre:  rain is non null
   * Post: rain[x][y] == 0 for all x,y in the array
   * Note that the loops use unit indexing.
   */
  public void initRain(double rain[][]) {
    for (int month = 1; month < rain.length; month++)
      for (int day = 1; day < rain[month].length; day++)
        rain[month][day] = Math.random() * 2.0;    // Random rainfall
  } // initRain()
  /**
   * Computes average daily rainfall for a year of rainfall data
   * @param rain is a 2D-array of rainfalls
   * @return The sum of rain[x][y] / 356
   * Pre:  rain is non null
   * Post: The sum of rain / 365 is calculated
   * Note that the loops are unit indexed
   */
  public double avgDailyRain(double rain[][]) {
    double total = 0;
    for (int month = 1; month < rain.length; month++)
      for (int day = 1; day < rain[month].length; day++)
        total += rain[month][day];
    return total/365;
  } // avgDailyRain()
  /**
   * Computes average daily rainfall for a given month containing nDays
   * @param monthRain is a 1D-array of rainfalls
   * @param nDays is the number of days in monthRain
   * @return The sum of monthRain / nDays
   * Pre:  1 <= nDays <= 31
   * Post: The sum of monthRain / nDays is calculated
   */
  public double avgRainForMonth(double monthRain[], int nDays) {
    double total = 0;
    for (int day = 1; day < monthRain.length; day++)
      total = total + monthRain[day];
    return total/nDays;
  } // avgRainForMonth()
  public static void main(String args[]) {
    double rainfall[][] = new double[13][32];
    Rainfall data = new Rainfall();
    data.initRain(rainfall);
    System.out.println("The average daily rainfall = "
                       +  data.avgDailyRain(rainfall));
    System.out.println("The average daily rainfall for March = "
                       +  data.avgRainForMonth(rainfall[3],31));
  } // main()
}//Rainfall
```

shows how we can test our array algorithms. It creates the rainfall array in the main() method. It then initializes the array and prints out average daily rainfall and average daily rainfall for the month of March. However, note that we have made a slight modification to the initRain() method. Instead of just assigning 0 to each element, we assign a random value between 0 and 2.0:

```
rain[month][day] = Math.random() * 2.0;
```

Using the Math.random() method in this way enables us to generate some realistic test data. In this case, we have scaled the data so that the daily rainfall is between 0 and 2 inches. (Rainfall like this would probably be appropriate for an Amazonian rain forest!) Testing our algorithms with these data provides some indication that our methods are in fact working properly.

Suppose you're going to keep track of the daily newspaper sales at the local kiosk. Declare a 52×7 two-dimensional array of int and initialize each of its elements to 0.

Write a method to calculate the average number of newspapers sold per week, using the array you declared in the previous

Write a method to calculate the average number of newspapers sold on Sundays, using the array you declared in the previous exercise. Assume that Sunday is the last day of the week.

Multidimensional Arrays (Optional)

Java doesn't limit arrays to just two dimensions. For example, suppose we decide to extend our rainfall survey to cover a ten-year period. For each year we now need a two-dimensional array. This results in a three-dimensional array consisting of an array of years, each of which contains an array of months, each of which contains an array of days:

```
final int NYEARS = 10;
final int NMONTHS = 13;
final int NDAYS = 32;
double rainfall[][] = new double[NYEARS][NMONTHS][NDAYS];
```

Following the design convention of not using the 0 month and 0 days, we end up with a $10 \times 13 \times 32$ array. Note the use of final variables to represent the size of each dimension of the array. This helps to make the program more readable.

In Figure [fig-threedarray], each year of the rainfall data is represented as a separate page. On each page, there is a two-dimensional table that consists of 12 rows (1 per month) and 31 columns (1 per day).

You might imagine that our study could be extended to cover rainfall data from a number of different cities. That would result in a four-dimensional array, with the first dimension now being the city. Of course, for this to work, cities would have to be represented by integers, because array subscripts must be integers.

As you might expect, algorithms for processing each element in a three-dimensional table would require a three-level nested loop. For example, the following algorithm would be used to initialize all elements of our three-dimensional rainfall array:

-2pc

```
for (int year = 0; year < rainfall.length; year++)
 for (int month = 0; month < rainfall[year].length; month++)
  for (int day = 0; day < rainfall[year][month].length; day++)
   rainfall[year][month][day] = 0.0;
```

Note again the proper use of the length attribute for each of the three dimensions of the array. In the outer loop, rainfall.length, we're referring to the number of years. In the middle loop, rainfall[year].length, we're referring to number of months within a given year. In the inner loop, rainfall[year][month].length, we're referring to the number of days within a month.

If we added a fourth dimension to our array and wanted to extend this algorithm to initialize it, we would simply embed the three-level loop within another for loop that would iterate over each city.

Array Initializers

It is possible to use an initializer with a **multidimensional array**. For instance, the following examples create several small arrays and initialize their elements:

```
int a[][] = {{1,2,3}, {4,5,6}};
char c[][] = {{'a','b'}, {'c','d'}};
double d[][][] =
    {{1.0,2.0,3.0}, {4.0,5.0}, {6.0,7.0,8.0,9.0}};
```

The first of these declarations creates a 2×3 array of integers. The second example creates a 2×2 array of characters, and the third example creates an array of double consisting of three rows, each of which has a different number of elements. The first row contains three elements, the second contains two elements, and the last row contains four elements. As this last example shows, the rows in a multidimensional array don't all have to have the same length.

Using initializers, as in these examples, is feasible only for relatively small arrays. To see why, just imagine what the initializer expression would be for our three-dimensional rainfall array. It would require $4,160 = 10 \times 13 \times 32$ zeroes, separated by commas!

OBJECT-ORIENTED DESIGN: Polymorphic Sorting (Optional)

One limitation of the sort routines developed so far is that they only work on one particular type of data. If you've written an insertion sort to sort ints, you can't use it to sort doubles. What would be far more desirable is a **polymorphic sort method**—that is, one method that could sort any kind of data. This is easily done by making use of Java wrapper classes, such as Integer and Double, together with the java.lang.Comparable interface, which is specially designed for this purpose.

The java.lang.Comparable interface consists of the method:

```
public abstract interface Comparable {
  public int compareTo(Object o);// Abstract method
}
```

By implementing compareTo(), a class can impose an order on its objects. The Comparable interface is implemented by all of Java's wrapper classes—that is, by Integer, Double, Float, Long, and so on (Fig. 9.24).

As we saw in Chapter 8, Java interfaces allow us to create a form of *multiple inheritance*. For example, as Figure 9.24 shows, an Integer is both an and a Comparable. One implication of this is that an Integer can be used in any method that takes either an Object parameter or a Comparable parameter.

The compareTo() method takes an Object parameter and returns an int. It is meant to be invoked as o1.compareTo(o2), where *o1* and *o2* are objects of the same type. Classes that implement compareTo() must abide by the following rules for its return value:

```
if (o1 < o2)       then o1.compareTo(o2) < 0
if (o1.equals(o2)) then o1.compareTo(o2) == 0
if (o1 > o2)       then o1.compareTo(o2) > 0
```

In other words, if o1 < o2, then o1.compareTo(o2) will return a negative integer. If o1 > o2, then o1.compareTo(o2) will return a positive integer. And if o1 and o2 are equal, then o1.compareTo(o2) will return 0.

For a class that implements Comparable, we can use the method to help sort its elements. For example, the following revised version of insertionSort() method can be used to sort any array of Comparable objects—that is, any array of objects whose class implements Comparable:

-1pc

```
public void sort(Comparable[] arr) {
  Comparable temp;   // Temporary variable for insertion
  for (int k = 1; k < arr.length; k++)  {
    temp = arr[k];    // Remove it from array
    int i;
    for (i = k-1; i >= 0 && arr[i].compareTo(temp) > 0; i--)
      arr[i+1] = arr[i];   // Move it right by one
    arr[i+1] = temp;        // Insert the element
  }
} // sort()
```

In this version, the parameter is an array of Comparable. Thus, we can pass it any array whose elements implement Comparable, including an array of Integer or Float, and so on. Then, to compare elements of a Comparable array, we use the compareTo() method:

```
for (i = k-1; i >= 0 && arr[i].compareTo(temp) > 0; i--)
```

Note that our algorithm no longer refers to ints, as in the original insertion sort. Indeed, it doesn't mention the specific type—Integer, Float, or whatever—of the objects that it is sorting. It refers only to Comparables. Therefore, we can use this method to sort any type of object, as long as the object's class implements the Comparable interface. Thus, by using Comparable, we have a more general insertionSort() method, one that can sort any one-dimensional array of Comparables.

The TestSort class (Figs. 9.25 and [fig-testsort]) provides an example of how to use the polymorphic sort() method.

It contains three methods: The sort() method that we just described; a polymorphic print() method, which can be used to print the values of any array of Comparable; and a main() method. The main() method creates arrays of Integer and Float and then uses the polymorphic sort() method to sort them. Note how the print() method uses the polymorphic toString() method to print the elements of a Comparable array.

-2pc

```
public class TestSort {
  public static int MAXSIZE = 25;

  public void sort(Comparable[] arr) {
    Comparable temp;      // Temporary variable for insertion
    for (int k = 1; k < arr.length; k++) { // For each element
      temp = arr[k];                     // Remove it
      int i;
      for (i = k-1; i >= 0 && arr[i].compareTo(temp) > 0; i--)
        arr[i+1] = arr[i]; // Move larger to the right
      arr[i+1] = temp;       // Insert the element
    }
  } // sort()
  public void print(Comparable arr[]) {
    for (int k = 0; k < arr.length; k++) {
      if (k % 5 == 0)  System.out.println();  // New row
        System.out.print(arr[k].toString() + "\t");
    }
    System.out.println();
  }
  // Sorts two different types of array with the same method.
  public static void main(String args[]) {
    Integer iArr[] = new Integer[TestSort.MAXSIZE];
    Float fArr[] = new Float[TestSort.MAXSIZE];
    for (int k = 0; k < TestSort.MAXSIZE; k++) { // Create data
      iArr[k] = new Integer((int) (Math.random() * 10000));
      fArr[k] = new Float(Math.random() * 10000);
    }
    TestSort test = new TestSort();
    test.sort(iArr);     // Sort and print Integers
    test.print(iArr);
    test.sort(fArr);     // Sort and print Floats
    test.print(fArr);
  } // main()
}
```

This example of polymorphic sorting illustrates once again the great power of inheritance and polymorphism in object-oriented programming. The Integer and Float classes use class inheritance to inherit features from the Object class, and they use interface implementation to inherit the compareTo() method from the Comparable class. By implementing versions of the toString() and compareTo() methods that are appropriate for these wrapper classes, Java makes it easier to use Integer and Float objects in a variety of contexts. Taken together, inheritance and polymorphism enable us to design very general and extensible algorithms. In this example, we defined a sort() method that can sort an array containing any kind of object as long as the object implements the Comparable interface.

The java.util.Arrays.sort() Method

While sorting algorithms provide a good way to introduce the concepts of array processing, real-world programmers never write their own sort algorithms. Instead they use library methods, which have been written and optimized by programming experts. Moreover, library sort routines use sort algorithms that are much more efficient than the ones we've discussed.

The java.util.Arrays class contains a polymorphic sort method that is very simple to use. For example, here's how we would use it to sort the two arrays declared in the TestSort program:

```
java.util.Arrays.sort(iArr);
java.util.Arrays.sort(fArr);
```

That's all there is to it! Obviously, learning how to use Java's class and method library, saves real-word programmers lots of effort.

Add a definition of a compareTo() method to the LetterFreq class so that it implements the Comparable interface. The method should define one object to be less than another object if its freq instance variable is less.

Add a definition of a sort() method that can be added to the definition of the AnalyzeFreq class. Make it so the array in the class can be sorted into ascending order using the ordering of LetterFreq defined in the previous exercise. Use the java.util.Arrays.sort() method.

Rewrite the main() of the AnalyzeFreq class to make use of the sort() method of the previous exercise.

From the Java Library: java.util.Vector

The java.util.Vector class implements an array of objects that can grow in size as needed. One limitation of regular arrays is that their lengths remain fixed. Once the array is full—once every element is used—you can't allocate additional elements.

The Vector class contains methods for storing and retrieving objects, and for accessing objects by their index position within the Vector (Fig. 9.27).

One use for a Vector would be when a program needs to store input from the user or a file without knowing in advance how many items there are. Using a Vector is less efficient than an array in terms of processing speed, but it gives you the flexibility of growing the data structure to meet the storage requirements.

As an illustration of this idea, the program in Figure [fig-vectordemo] creates a random number of integers and then stores them in a Vector. The Vector, which is declared and instantiated in main(), is initially empty. Integers from 0 to the random bound are then inserted into the Vector. In this case, insertions are done with the addElement() method, which causes the Vector object to insert the element at the next available location, increasing its size, if necessary.

```
import java.util.Vector;

public class VectorDemo {

  public static void printVector(Vector v) {
    for (int k=0; k < v.size(); k++)
      System.out.println(v.elementAt(k).toString());
  } // printVector()

  public static void main(String args[]) {
    Vector vector = new Vector();  // An empty vector

    int bound = (int)(Math.random() * 20);
    for (int k = 0; k < bound; k++ )   // Insert a random
      vector.addElement(new Integer(k));// number of Integers
    printVector(vector);            // Print the elements
  } // main()
}// VectorDemo
```

Once all the integers have been inserted, the printVector() method is called. Note that it uses the size() method to determine how many elements the Vector contains. This is similar to using the length() method to determine the number of characters in a String.

Finally, note that a Vector stores objects. It cannot be used to store primitive data values. You cannot store an int in a Vector. Therefore, we need to use the Integer wrapper class to convert ints into Integers before they can be inserted into the Vector. Because you can't just print an Integer, or any other Object, the toString() method is used to print the string representation of the object.

By defining Vector to store Objects, Java's designers have made it as general as possible and, therefore, as widely useful as possible.

Case Study: An N-Player Computer Game

In this section we will make use of arrays to extend our game-playing library by developing a design that can support games that involve more than two players. We will use an array to store a variable number of players. Following the object-oriented design principles described in Chapter 8, we will make use of inheritance and polymorphism to develop a design that is flexible and extensible, one that can be used to implement a wide variety of computer games. As in our TwoPlayer game example from Chapter 8, our design will allow both humans and computers to play the games. To help simplify the example, we will modify the WordGuess game that we developed in the Chapter 8. As you will see, it requires relatively few modifications to convert it from a subclass of TwoPlayerGame to a subclass of ComputerGame, the superclass for our N-Player game hierarchy.

The ComputerGame Hierarchy

Figure [fig-game] provides a summary overview of the ComputerGame hierarchy. This figure shows the relationships among the many classes and interfaces involved. The two classes whose symbols are bold, WordGuess and WordGuesser, are the classes that define the specific game we will be playing. The rest of the classes and interfaces are designed to be used with any N-player game.

At the root of this hierarchy is the abstract ComputerGame class. Note that it uses from 1 to *N* Players. These objects will be stored in a one-dimensional array in ComputerGame. Recall from Chapter 8 that an *IPlayer* was any class that implements the makeAMove() method. In this design, we have put the abstract makeAMove() method into the Player class, a class that defines a generic player of computer games. For the WordGuess game, the WordGuesser class extends Player. In order to play Word Guess, we will create a WordGuess instance, plus one or more instances of WordGuessers. This is similar to the OneRowNim example from the previous chapter,

Note where the TwoPlayerGame and OneRowNim classes occur in the hierarchy. TwoPlayerGame will now be an extension of ComputerGame. This is in keeping with the fact that a two-player game is a special kind of N-player computer game. As we will see when we look at the details of these classes, TwoPlayerGame will override some of the methods inherited from ComputerGame.

Because it contains the abstract makeAMove() method, the Player class is an abstract class. Its purpose is to define and store certain data and methods that can be used by any computer games. For example, one important piece of information defined in Player is whether the player is a computer or a person. Player's data and methods will be inherited by WordGuesser and by other classes that extend Player. Given its position in the hierarchy, we will be able to define polymorphic methods for WordGuessers that treat them as Players. As we will see, this will give our design great flexibility and extensibility.

The ComputerGame Class

Figure 9.30 shows the design details of the ComputerGame class. One of the key tasks of the ComputerGame class is to manage the one or more computer game players. Because this is a task that is common to all computer games, it makes sense to manage it here in the superclass. Toward this end, ComputerGame declares four instance variables and several methods. Three int variables define the total number of players (nPlayers), the number of players that have been added to the game (addedPlayers), and the player whose turn it is (whoseTurn). An array named player stores the Players. In keeping with the zero indexing convention of arrays, we number the players from 0 to *nPlayers-1*. These variables are all declared protected, so that they can be referenced directly by ComputerGame subclasses, but as protected variables, they remain hidden from all other classes.

The ComputerGame(int) constructor allows the number of players to be set when the game is constructed. The default constructor sets the number of players to one. The constructors create an array of length nPlayers:

```
public ComputerGame(int n) {
  nPlayers = n;
  player = new Player[n]; // Create the array
}
```

The setPlayer() and getPlayer() methods are the mutator and accessor methods for the whoseTurn variable. This variable allows a user to determine and set whose turn it is, a useful feature for initializing a game. The changePlayer() method uses the default expression,

```
whoseTurn = (whoseTurn + 1) % nPlayers
```

for changing whose turn it is. Assuming that players are numbered from 0 to *nPlayers-1*, this code gives the turn to the next player, wrapping around to player 0, if necessary. Of course, a subclass of ComputerGame can override this method if the game requires some other order of play.

The addPlayer(Player) method is used to add a new Player to the game, including any subclass of Player. The method assumes that addedPlayers is initialized to 0. It increments this variable by 1 each time a new player is added to the array. For the game WordGuess, we would be adding Players of type WordGuesser to the game.

The complete source code for ComputerGame is shown in Figure [fig-gamesource].

```
public abstract class ComputerGame {
  protected int nPlayers;
  protected int addedPlayers = 0;
  protected int whoseTurn;
  protected Player player[];  // An array of players

  public ComputerGame() {
    nPlayers = 1;       // Default: 1 player game
    player = new Player[1];
  }
  public ComputerGame(int n) {
    nPlayers = n;
    player = new Player[n]; // N-Player game
  }
  public void setPlayer(int starter) {
    whoseTurn = starter;
  }
  public int getPlayer() {
    return whoseTurn;
  }
  public void addPlayer(Player p) {
    player[addedPlayers] = p;
    ++addedPlayers;
  }
  public void changePlayer() {
    whoseTurn = (whoseTurn + 1) % nPlayers;
  }
  public String getRules() {
    return "The rules of this game are: ";
  }
  public String listPlayers() {
    StringBuffer result =
          new StringBuffer("\nThe players are:\n");
    for (int k = 0; k < nPlayers; k++)
      result.append("Player" + k + " " +
                    player[k].toString() + "\n");
      result.append("\n");
      return result.toString();
  }
  public abstract boolean gameOver(); // Abstract
  public abstract String getWinner(); //  methods
} //ComputerGame
```

There are several points worth noting about this implementation. First, note that just as in the case of the abstract TwoPlayerGame class from Chapter 8, the methods gameOver() and getWinner() are defined as abstract and the getRules() method is given a generic implementation. The intent here is that the subclass will override getRules() and will provide game-specific implementations for the abstract methods.

Second, note how the addPlayer() method is coded. It uses the addedPlayers variable as the index into the player array, which always has length nPlayers. An attempt to call this method when the array is already full will lead to the following exception being thrown by Java:

```
Exception in thread ``main''
    java.lang.ArrayIndexOutOfBoundsException: 2
    at ComputerGame.addPlayer(ComputerGame.java:22)
    at TwentyOne.main(TwentyOne.java:121)
```

In other words, it is an error to try to add more players than will fit in the player array. In Chapter 11, we will learn how to design our code to guard against such problems.

Finally, note the implementation of the listPlayers() method (Fig. [fig-gamesource]). Here is a good example of polymorphism at work. The elements of the player array have a declared type of Player. Their dynamic type is WordGuesser. So when the expression player[k].toString() is invoked, dynamic binding is used to bind this method call to the implementation of toString() defined in the WordGuesser class. Thus, by allowing toString() to be bound at run time, we are able to define a method here that doesn't know the exact types of the objects it will be listing.

The power of polymorphism is the flexibility and extensibility it lends to our class hierarchy. Without this feature, we would not be able to define listPlayers() here in the superclass, and would instead have to define it in each subclass.

The WordGuess and WordGuesser Classes

We will assume here that you are familiar with the WordGuess example from Chapter 8. If not, you will need to review that section before proceeding. Word Guess is a game in which players take turns trying to guess a secret word by guessing its letters. Players keep guessing as long as they correctly guess a letter in the word. If they guess wrong, it becomes the next player's turn. The winner of the game is the person who guesses the last letter secret letter, thereby completely identifying the word.

Figure 9.32 provides an overview of the WordGuess class. If you compare it with the design we used in Chapter 8, the only change in the instance methods and instance variables is the addition of a new constructor, WordGuess(int), and an init() method. This constructor takes an integer parameter representing the number of players. The default constructor assumes that there is one player. Of course, this version of WordGuess extends the ComputerGame class, rather than the TwoPlayerGame class. Both constructors call the init() method to initialize the game:

```
public WordGuess() { super(1); init(); }
public WordGuess(int m) { super(m); init(); }
public void init() {
    secretWord = getSecretWord();
    currentWord = new StringBuffer(secretWord);
    previousGuesses = new StringBuffer();
    for (int k = 0; k < secretWord.length(); k++)
       currentWord.setCharAt(k,'?');
    unguessedLetters = secretWord.length();
}
```

The only other change required to convert WordGuess to an N-player game, is to rewrite its play() method. Because the new play() method makes use of functionality inherited from the ComputerGame() class, it is actually much simpler than the play() method in the Chapter 8 version:

```
public void play(UserInterface ui) {
  ui.report(getRules());
  ui.report(listPlayers());
  ui.report(reportGameState());

  while(!gameOver()) {
    WordGuesser p = (WordGuesser)player[whoseTurn];
    if (p.isComputer())
      ui.report(submitUserMove(p.makeAMove(getGamePrompt())));
    else {
      ui.prompt(getGamePrompt());
      ui.report(submitUserMove(ui.getUserInput()));
    }
    ui.report(reportGameState());
  } // while
}
```

The method begins by displaying the game's rules and listing its players. The listPlayers() method is inherited from the ComputerGame class. After displaying the game's current state, the method enters the play loop. On each iteration of the loop, a player is selected from the array:

```
WordGuesser p = (WordGuesser)player[whoseTurn];
```

The use of the WordGuesser variable, p, just makes the code somewhat more readable. Note that we have to use a cast operator, (WordGuesser), to convert the array element, a Player, into a WordGuesser. Because p is a WordGuesser, we can refer directly to its isComputer() method.

If the player is a computer, we prompt it to make a move, submit the move to the submitUserMove() method, and then report the result. This is all done in a single statement:

```
ui.report(submitUserMove(p.makeAMove(getGamePrompt())));
```

If the player is a human, we prompt the player and use the KeyboardReader's getUserInput() method to read the user's move. We then submit the move to the submitUserMove() method and report the result. At the end of the loop, we report the game's updated state. The following code segment illustrates a small portion of the interaction generated by this play() method:

```
Current word ???????? Previous guesses GLE
Player 0 guesses next.Sorry, Y is NOT a new letter
                      in the secret word
Current word ???????? Previous guesses GLEY
Player 1 guesses next.Sorry, H is NOT a new letter
                      in the secret word
Current word ???????? Previous guesses GLEYH
Player 2 guesses next.
Guess a letter that you think is in the secret word: a
Yes, the letter A is in the secret word
```

In this example, players 0 and 1 are computers and player 2 is a human.

In our new design, the WordGuesser class is a subclass of Player (Figure 9.33). The WordGuesser class itself requires no changes other than its declaration:

```
public class WordGuesser extends Player
```

As we saw when we were discussing the WordGuess class, the play() method invokes WordGuesser's isComputer() method. But this method is inherited from the Player class. The only other method used by play() is the makeAMove() method. This method is coded exactly the same as it was in the previous version of WordGuesser.

Figure [fig-player] shows the implementation of the Player class. Most of its code is very simple. Note that the default value for the kind variable is HUMAN and the default id is -1, indicating the lack of an assigned identification.

```
public abstract class Player  {
  public static final int COMPUTER=0;
  public static final int HUMAN=1;
  protected int id = -1;  // Id between 0 and nPlayers-1
  protected int kind = HUMAN; // Default is HUMAN

  public Player() { }
  public Player(int id, int kind) {
    this.id = id;
    this.kind = kind;
  }
  public void setID(int k) { id = k; }
  public int getID() { return id; }
  public void setKind(int k) { kind = k; }
  public int getKind() { return kind; }
  public boolean isComputer() { return kind == COMPUTER; }
  public abstract String makeAMove(String prompt);
} // Player
```

What gives Player its utility is the fact that it encapsulates those attributes and actions that are common to all computer game players. Defining these elements, in the superclass, allows them to be used throughout the Player hierarchy. It also makes it possible to establish an association between a Player and a ComputerGame.

Given the ComputerGame and Player hierarchies and the many interfaces they contain, the task of designing and implementing a new N-player game is made much simpler. This too is due to the power of object-oriented programming. By learning to use a library of classes, such as these, even inexperienced programmers can create relatively sophisticated and complex computer games.

Finally, the following main() method instantiates and runs an instance of the WordGuess game in a command-line user interface (CLUI):

```
public static void main(String args[])
{ KeyboardReader kb = new KeyboardReader();
  ComputerGame game = new WordGuess(3);
  game.addPlayer(new WordGuesser((WordGuess)game, 0,
                                          Player.HUMAN));
  game.addPlayer(new WordGuesser((WordGuess)game, 1,
                                        Player.COMPUTER);
  game.addPlayer(new WordGuesser((WordGuess)game, 2,
                                        Player.COMPUTER);
  ((CLUIPlayableGame)game).play(kb);
} //main()
```

In this example, we create a three player game in which two of the players are computers. Note how we create a new WordGuesser, passing it a reference to the game itself, as well as its individual identification number, and its type (HUMAN or COMPUTER). To run the game, we simply invoke its play() method. You know enough now about object-oriented design principles to recognize that the use of play() in this context is an example of polymorphism.

A GUI-Based Game (Optional Graphics)

Most modern computer games do not use a command-line interface. This section addresses this shortcoming by expanding our ComputerGame hierarchy so that it works with Graphical User Interfaces (GUIs) as well as Command-Line User Interfaces (CLUIs).

The Sliding Tile Puzzle is a puzzle game. It is played by one player, a human. The puzzle consists of six tiles arranged on a board containing seven spaces. Three of the tiles are labeled *L* and three are labeled *R*. Initially the tiles are arranged as *RRR_LLL*. In other words, the *R* tiles are arranged to the left of the *L* tiles, with the blank space in the middle. The object of the puzzle is to rearrange the tiles into *LLL_RRR*. The rules are that tiles labeled *R* can only move right. Tiles labeled *L* can only move left. Tiles may move directly into the blank space or they can jump over one tile into the blank space.

Our purpose in this section is to develop a GUI that plays this game. An appropriate GUI is shown Figure [fig-capture]. Here the tiles and the blank space are represented by an array of buttons. To make a move the user clicks on the 'tile' he or she wishes to move. The GUI will assume that the user wants to move that tile into the blank space. If the proposed move is legal, the GUI will carry out the move. Otherwise, it will just ignore it. For example, if the user were to click on the third *R* button from the left, a legal move, the GUI would rearrange the labels on the buttons so that their new configuration would be *RR_RLLL*. On the other hand, if the user were to click on the rightmost *L* button, the GUI would ignore that move, because it is illegal.

The GUIPlayableGame Interface

How should we extend our game-playing hierarchy to accommodate GUI-based games? As we learned in Chapter 4, one difference between GUI-based applications and CLUI-based applications, is the locus of control. In a CLUI-based application, control resides in the computational object which, for games, is the game object. That's why the play() method in our CLUI-based games contains the game's control loop. By contrast, control resides in the GUI's event loop in GUI-based applications. That's why, we learned how to manage Java's event hierarchy in Chapter 4. Thus, in the GUI shown in Figure [fig-capture], the GUI will *listen* and take *action* when the user clicks one of its buttons.

However, given that control will reside in the GUI, there is still a need for communication between the GUI and the game object. In the CLUI-based games, we have used the CLUIPlayableGame interface to manage the communication between the game and the user interface. We will follow the same design strategy in this case. Thus, we will design a GUIPlayableGame interface that can be implemented by any game that wishes to use a GUI (Fig. 9.36).

What method(s) should this interface contain? One way to answer this question is to think about the type of interaction that must take place when the user clicks one of the tiles. If the user clicks the third *R* button, the GUI should pass this information to the game. The game should then decide whether or not that is a legal move and communicate this back to the GUI. Assuming it is a legal move, the game should also update its representation of the game's state to reflect that the tile array has changed. And it should somehow make communicate the game's state to the GUI.

Because it is impossible to know in advance just what form of data a game's moves might take, we will use Java Strings to communicate between the user interface and the game object. Thus, a method with the following signature will enable us to submit a String representing the user's move to the game and receive in return a String representing the game object's response to the move:

```
submitUserMove(String move): String;
```

In addition to this method, a GUI interface could use the reportGameState() and getGamePrompt() methods that are part of the IGame interface. Th design shown in Figure 9.36 leads to the following definition for the GUIPlayableGame interface:

```
public interface GUIPlayableGame extends IGame {
    public String submitUserMove(String theMove);
}
```

Because it extends IGame, this interface inherits the getGamePrompt() and reportGameState() from the IGame interface. The GUI should be able to communicate with any game that implements this interface.

The SlidingTilePuzzle

Let's now discuss the design and details of the SlidingTilePuzzle itself. Its design is summarized in Figure 9.37. Most of the methods should be familiar to you, because the design closely follows the design we employed in the WordGuess example. It has implementations of inherited methods from the ComputerGame class and the GUIPlayableGame interface.

We will represent the sliding tile puzzle in a one-dimensional array of char. We will store the puzzle's solution in a Java String and we will use an int variable to keep track of where the blank space is in the array. This leads to the following class-level declarations:

```
private char puzzle[] = {'R','R','R',' ','L','L','L'};
private String solution = "LLL RRR";
private int blankAt = 3;
```

Note how we initialize the puzzle array with the initial configuration of seven characters. Taken together, these statements initialize the puzzle's state.

Because a puzzle is a one-person game and our sliding tile puzzle will be played by a human, this leads to a very simple constructor definition:

```
public SlidingTilePuzzle() {
    super(1);
}
```

We call the super() constructor (ComputerGame()) to create a one-person game.

The puzzle's state needs to be communicated to the GUI as a String. This is the purpose of the reportGameState() method:

```
public String reportGameState() {
    StringBuffer sb = new StringBuffer();
    sb.append(puzzle);
    return sb.toString();
}
```

We use a StringBuffer() to convert the puzzle, a char array, into a String.

The most important method for communicating with the GUI is the submitUserMove() method:

```
public String submitUserMove(String usermove) {
  int tile = Integer.parseInt(usermove);
  char ch = puzzle[tile];
  if (ch == 'L' &&
              (blankAt == tile-1 || blankAt == tile-2))
    swapTiles(tile,blankAt);
  else if (ch == 'R' &&
              (blankAt == tile+1 || blankAt == tile+2))
    swapTiles(tile,blankAt);
  else
    return "That's an illegal move.\n";
  return "That move is legal.\n";
}
```

This is the method that processes the user's move, which is communicated through the GUI. As we saw, the puzzle's 'tiles' are represented by an array of buttons in the GUI. The buttons are indexed 0 through 6 in the array. When the user clicks a button, the

GUI should pass its index, represented as a String to the submitUserMove() method. Given the index number of the tile that was selected by the user, this method determines if the move is legal.

The Integer.parseInt() method is used to extract the tile's index from the method's parameter. This index is used to get a 'tile' from the puzzle array. The logic in this method reflects the rules of the game. If the tile is an *L*, then it can only move into a blank space that is either 1 or 2 spaces to its left. Similarly, an *R* tile can only move into a blank space that is 1 or 2 spaces to its right. All other moves are illegal. For legal moves, we simply swap the tile and the blank space in the array, a task handled by the swap() method. In either case, the method returns a string reporting whether the move was legal or illegal.

Figure [fig-puzzlecode] shows the full implementation for the SlidingTilePuzzle, the remaining details of which are straight forward.

```
public class SlidingTilePuzzle extends ComputerGame
                            implements GUIPlayableGame {
  private char puzzle[] = {'R','R','R',' ','L','L','L'};
  private String solution = "LLL RRR";
  private int blankAt = 3;
  public SlidingTilePuzzle() { super(1); }

  public boolean gameOver() { // True if puzzle solved
    StringBuffer sb = new StringBuffer();
    sb.append(puzzle);
    return sb.toString().equals(solution);
  }
  public String getWinner() {
    if (gameOver())
      return "\nYou did it! Very Nice!\n";
    else return "\nGood try. Try again!\n";
  }
  public String reportGameState() {
    StringBuffer sb = new StringBuffer();
    sb.append(puzzle);
    return sb.toString();
  }
  public String getGamePrompt() {
    return "To move a tile, click on it.";
  } //prompt()

  public String submitUserMove(String usermove) {
    int tile = Integer.parseInt(usermove);
    char ch = puzzle[tile];
    if (ch=='L' && (blankAt==tile-1 || blankAt==tile-2))
      swapTiles(tile,blankAt);
    else if (ch=='R' &&
                  (blankAt==tile+1 || blankAt==tile+2))
      swapTiles(tile,blankAt);
    else
      return "That's an illegal move.\n";
    return "That move is legal.\n";
  }
  private void swapTiles(int ti, int bl) {
    char ch = puzzle[ti];
    puzzle[ti] = puzzle[bl];
    puzzle[bl] = ch;
    blankAt = ti;   // Reset the blank
  }
} //SlidingTilePuzzle
```

The SlidingGUI Class

Let's now implement a GUI that can be used to play the sliding tile puzzle. We will model the GUI itself after those we designed in Chapter 4.

Figure 9.39 provides a summary of the design. As an implementor of the interface, SlidingGUI implements the actionPerformed() method, which is where the code that controls the puzzle is located. The main data structure is an array of seven JButtons,

representing the seven tiles in the puzzles. The buttons' labels will reflect the state of the puzzle. They will be rearranged after every legal move by the user. The reset button is used to reinitialize the game. This allows users to play again or to start over if they get stuck.

The puzzleState is a String variable that stores the puzzle's current state, which is updated repeatedly from the SlidingTilePuzzle by calling its reportGameState() method. The private labelButtons() method will read the puzzleState and use its letters to set the labels of the GUI's buttons.

The implementation of SlidingGUI is shown in Figure [fig-GUIsource]. Its constructor and buildGUI() methods are responsible for setting up the GUI. We use of a for loop in buildGUI() to create the JButtons, associate an ActionListener with them, and add them to the GUI. Except for the fact that we have an array of buttons, this is very similar to the GUI created in Chapter 4. Recall that associating an ActionListener with the buttons allows the program to respond to button clicks in its actionPerformed() method.

Note how an instance of the SlidingTilePuzzle is created in the constructor, and how its state is retrieved and stored in the puzzleState variable:

```
puzzleState = sliding.reportGameState();
```

The labelButtons() method transfers the letters in puzzleState onto the buttons.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class SlidingGUI extends JFrame implements ActionListener {
  private JButton tile[] = new JButton[7];
  private JButton reset = new JButton("Reset");
  private SlidingTilePuzzle sliding;
  private String puzzleState;
  private Label label;
  private String prompt = "Goal: [LLL RRR]. " +
    " Click on the tile you want to move." +
    " Illegal moves are ignored.";
  public SlidingGUI(String title) {
    sliding = new SlidingTilePuzzle();
    buildGUI();
    setTitle(title);
    pack();
    setVisible(true);
  } // SlidingGUI()
  private void buildGUI() {
    Container contentPane = getContentPane();
    contentPane.setLayout(new BorderLayout());
    JPanel buttons = new JPanel();
    puzzleState = sliding.reportGameState();
    for (int k = 0; k < tile.length; k++) {
      tile[k] = new JButton(""+puzzleState.charAt(k));
      tile[k].addActionListener(this);
      buttons.add(tile[k]);
    }
    reset.addActionListener(this);
    label = new Label(prompt);
    buttons.add(reset);
    contentPane.add("Center", buttons);
    contentPane.add("South", label);
  } // buildGUI()
```

```
  } // buildGUI()
  private void labelButtons(String s) {
    for (int k = 0; k < tile.length; k++)
      tile[k].setText(""+ s.charAt(k));
  } // labelButtons()
  public void actionPerformed(ActionEvent e) {
    String result = "";
    if (e.getSource() == reset) { // Reset clicked?
      sliding = new SlidingTilePuzzle();
      label.setText(prompt);
    }
    for (int k = 0; k < tile.length; k++) // Tile clicked?
      if (e.getSource() == tile[k])
        result = ((GUIPlayableGame)sliding).submitUserMove(""+ k);
    if (result.indexOf("illegal") != -1)
      java.awt.Toolkit.getDefaultToolkit().beep();
    puzzleState = sliding.reportGameState();
    labelButtons(puzzleState);
    if (sliding.gameOver())
      label.setText("You did it! Very nice!");
  } // actionPerformed()
  public static void main(String args[]) {
    new SlidingGUI("Sliding Tile Puzzle");
  } // main()
} // SlidingGUI
```

The most important method in the GUI is the actionPerformed() method. This method controls the GUI's actions and is called automatically whenever one of the GUI's buttons is clicked. First, we check whether the reset button has been clicked. If so, we reset the puzzle by creating a new instance of SlidingTilePuzzle and re-initializing the prompt label.

Next we use a for loop to check whether one of the tile buttons has been clicked. If so, we use the loop index, *k*, as the tile's identification and submit this to the puzzle as the user's move:

```
if (e.getSource() == tile[k])
  result = ((GUIPlayableGame)sliding).submitUserMove(""+ k);
```

The cast operation is necessary here because we declared sliding as a SlidingTilePuzzle rather than as a GUIPlayableGame. Note also that we have to convert *k* to a String when passing it to submitUserMove().

As a result of this method call, the puzzle returns a result, which is checked to see if the user's move was illegal. If result contains the word "illegal", the computer beeps to signal an error:

```
 if (result.indexOf("illegal") != -1)
    java.awt.Toolkit.getDefaultToolkit().beep();
```

The java.awt.Toolkit is a class that contains lots of useful methods, including the beep() method. Note that no matter what action is performed, a reset or a tile click, we update puzzleState by calling reportGameState() and use it to relabel the tile buttons. The last task in the actionPerformed() method is to invoke the puzzle's gameOver() method to check if the user has successfully completed the puzzle. If so, we display a congratulatory message in the GUI's window.

Finally, the main() for a GUI is very simple, consisting of a single line of code:

```
    new SlidingGUI("Sliding Tile Puzzle");
```

Once a SlidingGUI is created, with the title of “Sliding Tile Puzzle,” it will open a window and manage the control of the puzzle.

array

array initializer

array length

binary search

data structure

element

element type

insertion sort

multidimensional array

one-dimensional array

polymorphic sort method

selection sort

sequential search

sorting

subscript

two-dimensional array

An **array** is a named collection of contiguous storage locations, each of which stores a data item of the same data type. Each element of an array is referred to by a *subscript*—that is, by its position in the array. If the array contains *N* elements, then its length is *N* and its indexes are 0, 1, …N-1.

Array elements are referred to using the following subscript notation *arrayname[subscript]*, where *arrayname* is any valid identifier, and *subscript* is an integer value in the range 0 to arrayname.length - 1. The array’s length instance variable can be used as a bound for loops that process the array.

An *array declaration* provides the name and type of the array. An array instantiation uses the keyword new and causes the compiler to allocate memory for the array’s elements:

```
int arr[]; // Declare a one-dimensional array variable
arr = new int[15];// Allocate 15 int locations for it
```

Multidimensional arrays have arrays as their components:

```
int twoDarr[][]; // Declare a two-dimensional array variable
twoDarr = new int[10][15]; // Allocate 150 int locations
```

An array’s values must be initialized by assigning values to each array location. An *initializer* expression may be included as part of the array declaration.

Insertion sort and selection sort are examples of array sorting algorithms. Both algorithms require several passes over the array.

When an array is passed as a argument to a method, a reference to the array is passed rather than the entire array itself.

Swapping two elements of an array, or any two locations in memory, requires the use of a temporary variable.

Sequential search and binary search are examples of array searching algorithms. Binary search requires that the array be sorted.

For multidimensional arrays, each dimension of the array has its own length variable.

Inheritance and polymorphism are useful design features for developing a hierarchy of computer games.

Space (in bytes) allocated for each of the following?

1. `int a[] = new int[5];  // 5 * 4 = 20 bytes`
2. `double b[] = new double[10];  // 10 * 8 = 80 bytes`
3. `char c[] = new char[30];  // 30 * 2 = 60 bytes`
4. `String s[] = new String[10];  // 10 * 4 (reference) = 40 bytes`
5. `Student s[] = new Student[5];  // 5 * 4 (reference) = 20 bytes`

An array containing 10 floats, 1.0 to 10.0.

```
float farr[] = {1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0};
```

Prints the first element of farr.

```
System.out.println(farr[0]);
```

Assigns 100.0 to the last element in farr.

```
farr[farr.length-1] = 100.0;
```

A loop to print all of the elements of farr.

```
for (int j = 0; j < farr.length; j++)
    System.out.println(farr[j]);
```

An array containing the first 100 square roots.

```
double doubarr[] = new double[100];
for (int k = 0; k < doubarr.length; k++)
    doubarr[k] = Math.sqrt(k+1);
```

Analyzing the letter frequencies in a file.

```
import java.io.*;
import java.util.Scanner;
public static void main(String[] argv) {
  Scanner fileScan;  // To read lines of text from the file
  String str;        // To store the line of text
  AnalyzeFreq af = new AnalyzeFreq();
  try {         // Create a Scanner
    File theFile = new File("freqtest.txt");
    fileScan = Scanner.create(theFile);
    fileScan = fileScan.useDelimiter("\r\n"); //For Windows
    while (fileScan.hasNext()) { // Read and count
      str = fileScan.next();
      af.countLetters(str);
    } //while
    af.printArray(); // Print frequencies
  } catch (Exception e) {
     e.printStackTrace();
  } //catch()
} //main()
```

Sort 24 18 90 1 0 85 34 18 with insertion sort.

```
  24 18 90 1  0  85 34 18 // Initial
  18 24 90 1  0  85 34 18 // Pass 1
  18 24 90 1  0  85 34 18 // Pass 2
  1  18 24 90 0  85 34 18 // Pass 3
  0  1  18 24 90 85 34 18 // Pass 4
  0  1  18 24 85 90 34 18 // Pass 5
  0  1  18 24 34 85 90 18 // Pass 6
  0  1  18 18 24 34 85 90 // Pass 7
```

Sort 24 18 90 1 0 85 34 18 with selection sort.

```
24 18 90 1   0   85 34 18 // Initial
0  18 90 1   24  85 34 18 // Pass 1
0  1  90 18  24  85 34 18 // Pass 2
0  1  18 90  24  85 34 18 // Pass 3
0  1  18 18  24  85 34 90 // Pass 4
0  1  18 18  24  85 34 90 // Pass 5
0  1  18 18  24  34 85 90 // Pass 6
0  1  18 18  24  34 85 90 // Pass 7
```

Code to swap two Students.

```
Student tempStud = student1;
student1 = student2;
student2 = tempStud;
```

Implementation of the selectionSort().

```
 public void selectionSort(int arr[]) {
    int smallest;      // Location of smallest element
    for (int k = 0; k < arr.length-1; k++) {
        smallest = k;
        for (int j = k+1; j < arr.length; j++)
            if (arr[j] < arr[smallest])
                smallest = j;
        if (smallest != k) {   // Swap smallest and kth
            int temp = arr[smallest];
            arr[smallest] = arr[k];
            arr[k] = temp;
        } // if
    } // outer for
 } // selectionSort()
```

After mystery(myArr,k), myArr will store 1,2,3,5,5 and *k* will store 3.

Binary search trace for 21 in 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28:

```
key  iteration  low   high    mid
-------------------------------------
21   0          0     13      6
21   1          7     13      10
21   2          7     9       8
21   3          9     9       9
21   4          10    9       failure
```

A two-dimensional array with 5 rows of 10 integers.

```
int int2d[][] = new int[5][10];
```

Prints the last integer in the third row int2d and assigns 100 to its last element.

```
System.out.println(int2d[2][9]);
int2d[4][9] = 100;
```

Prints all of the elements of int2d.

```
for (int k = 0; k < int2d.length; k++) {
     for (int j = 0; j < int2d[k].length; j++)
         System.out.print( int2d[k][j] + " ");
     System.out.println();    // new line
}
```

A 52×7 two-dimensional array of int.

```
int sales[][] = new int[52][7];
for (int k = 0; k < sales.length; k++)
    for (int j= 0; j < sales[k].length; j++)
        sales[k][j] = 0;
```

A method to calculate average number of newspapers per week.

```
double avgWeeklySales(int arr[][]) {
    double total = 0;
    for (int k = 0; k < arr.length; k++)
        for (int j= 0; j < arr[k].length; j++)
            total += arr[k][j];
    return total/52;
}
```

A method to calculate average Sunday newspapers.

```
double avgSundaySales(int arr[][]) {
    double total = 0;
    for (int k = 0; k < arr.length; k++)
        total += arr[k][6];
    return total/52;
}
```

A compareTo() for LetterFreq.

```
public int compareTo(Object lf) {
     LetterFreq  letFreq = (LetterFreq)lf;
     if (freq < letFreq.getFreq())
         return -1;
     else if (freq > letFreq.getFreq())
         return +1;
     else return 0; //The frequencies must be equal.
} //compareTo()
```

A sort() for AnalyzeFreq.

```
public void sort() {
     java.util.Arrays.sort(freqArr);
} //sort()
```

A new AnalyzeFreq.main() that uses sort().

```
public static void main(String[] argv) {
  AnalyzeFreq af = new AnalyzeFreq();
  af.countLetters("Now is the time for all good students" +
     " to study computer-related topics.");
  af.sort();
  af.printArray();
} //main()
```

Explain the difference between the following pairs of terms:

An *element* and an element *type*.

A *subscript* and an *array element*.

A *one-dimensional* array and *two-dimensional* array.

An *array* and a *vector*.

A *insertion sort* and a *selection sort*.

A *binary search* and a *sequential search*.

Fill in the blanks.

=14pt

The process of arranging an array’s elements into a particular order is known as

---

.

One of the preconditions of the binary search method is that the array has to be

---

.

An

---

is an object that can store a collection of elements of the same type.

An

---

is like an array except that it can grow.

For an array, its

is represented by an instance variable.

An expression that can be used during array instantiation to assign values to the array is known as an

.

A

is an array of arrays.

A sort method that can be used to sort different types of data is known as a

method.

To instantiate an array you have to use the

operator.

An array of objects stores

to the objects.

=11pt

Make each of the following array declarations:

A 4×4 array of doubles.

A 20×5 array of Strings.

A 3×4 array of char initialized to '*';

A $2 \times 3 \times 2$ array of boolean initialized to true.

A 3×3 array of Students.

A 2×3 array of Strings initialized to "one," "two," and so on.

Identify and correct the syntax error in each of the following expressions:

```
int arr = new int[15];
int arr[] = new int(15);
float arr[] = new [3];
float arr[] = new float {1.0,2.0,3.0};
int arr[] = {1.1,2.2,3.3};
int arr[][] = new double[5][4];
int arr[][] = { {1.1,2.2}, {3.3, 1} };
```

Evaluate each of the following expressions, some of which may be erroneous:

```
int arr[] = { 2,4,6,8,10 };
```

2

```
arr[4]
arr[ arr.length ]
arr[ arr[0] ]
arr[ arr.length / 2 ]
```

```
arr[ arr[1] ]
arr[ 5 % 2 ]
arr[ arr[ arr[0] ] ]
arr[ 5 / 2.0 ]
arr[ 1 + (int) Math.random() ]
arr[ arr[3] / 2 ]
```

What would be printed by the following code segment?

```
int arr[] = { 24, 0, 19, 21, 6, -5, 10, 16};
for (int k = 0; k < arr.length; k += 2)
  System.out.println( arr[k] );
```

What would be printed by the following code segment?

```
int arr[][] = { {24, 0, 19}, {21, 6, -5}, {10, 16, 3},
                                          {1, -1, 0} };
for (int j = 0; j < arr.length; j++)
  for (int k = 0; k < arr[j].length; k++)
    System.out.println( arr[j][k] );
```

What would be printed by the following code segment?

```
int arr[][] = { {24, 0, 19}, {21, 6, -5}, {10, 16, 3},
                                          {1, -1, 0} };
for (int j = 0; j < arr[0].length; j++)
    for (int k = 0; k < arr.length; k++)
        System.out.println(arr[k][j]);
```

What's wrong with the following code segment, which is supposed to swap the values of the int variables, *n1* and *n2*?

```
int temp = n1;
n2 = n1;
n1 = temp;
```

Explain why the following method does not successfully swap the values of its two parameters? Hint: Think about the difference between value and reference parameters.

```
public void swapEm(int n1, int n2) {
    int temp = n1;
    n1 = n2;
    n2 = temp;
}
```

Declare and initialize an array to store the following two-dimensional table of values:

```
1   2   3   4
5   6   7   8
9  10  11  12
```

For the two-dimensional array you created in the previous exercise, write a nested for loop to print the values in the following order: 1 5 9 2 6 10 3 7 11 4 8 12. That is, print the values going down the columns instead of going across the rows.

Define an array that would be suitable for storing the following values:

The GPAs of 2,000 students.

The lengths and widths of 100 rectangles.

A week's worth of hourly temperature measurements, stored so that it is easy to calculate the average daily temperature.

A board for a tic-tac-toe game.

The names and capitals of the 50 states.

Write a code segment that will compute the sum of all the elements of an array of int.

Write a code segment that will compute the sum of the elements in a two-dimensional array of int.

Write a method that will compute the average of all the elements of a two-dimensional array of float.

Write a method that takes two parameters, an int array and an integer, and returns the location of the last element of the array that is greater than or equal to the second parameter.

Write a program that tests whether a 3×3 array, input by the user, is a *magic square*. A magic square is an $N \times N$ matrix of numbers in which every number from 1 to N^2 must appear just once, and every row, column, and diagonal must add up to the same total—for example,

```
6 7 2
1 5 9
8 3 4
```

Revise the program in the previous exercise so that it allows the user to input the dimensions of the array, up to 4×4.

Modify the AnalyzeFreq program so that it can display the relative frequencies of the 10 most frequent and 10 least frequent letters.

The *merge sort* algorithm takes two collections of data that have been sorted and merges them together. Write a program that takes two 25-element int arrays, sorts them, and then merges them, in order, into one 50-element array.

Challenge: Design and implement a BigInteger class that can add and subtract integers with up to 25 digits. Your class should also include methods for input and output of the numbers. If you're really ambitious, include methods for multiplication and division.

Challenge: Design a data structure for this problem: As manager of Computer Warehouse, you want to track the dollar amount of purchases made by those clients that have regular accounts. The accounts are numbered from 0, 1, …, N. The problem is that you don't know in advance how many purchases each account will have. Some may have one or two purchases. Others may have 50 purchases.

An *anagram* is a word made by rearranging the letters of another word. For example, *act* is an anagram of *cat*, and *aegllry* is an anagram of *allergy*. Write a Java program that accepts two words as input and determines if they are anagrams.

Challenge: An *anagram dictionary* is a dictionary that organizes words together with their anagrams. Write a program that lets the user enter up to 100 words (in a TextField, say). After each word is entered, the program should display (in a TextArea perhaps) the complete anagram dictionary for the words entered. Use the following sample format for the dictionary. Here the words entered by the user were: *felt*, *left*, *cat*, *act*, *opt*, *pot*, *top*.

```
act:  act cat
eflt:  felt left
opt:   opt pot top
```

Acme Trucking Company has hired you to write software to help dispatch its trucks. One important element of this software is knowing the distance between any two cities that it services. Design and implement a Distance class that stores the distances between cities in a two-dimensional array. This class will need some way to map a city name, *Boise*, into an integer that can be

used as an array subscript. The class should also contain methods that would make it useful for looking up the distance between two cities. Another useful method would tell the user the closest city to a given city.

Rewrite the main() method for the WordGuess example so that it allows the user to input the number of players and whether each players is a computer or a human. Use a KeyboardReader.

Write a smarter version of the WordGuesser class that “knows” which letters of the English language are most frequent. HINT: Rather than using random guesses, store the player’s guesses in a string in order of decreasing frequency: "ETAONRISHLGCMFBDGPUKJVWQXYZ".

Write a CLUI version of the SlidingTilePuzzle. You will need to make modifications to the SlidingTilePuzzle class.

This page titled 8.1: Introduction is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **9.1: Introduction** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

8.2: One-Dimensional Arrays

This page titled 8.2: One-Dimensional Arrays is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

8.3: Simple Array Examples

This page titled 8.3: Simple Array Examples is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

8.4: Example- Counting Frequencies of Letters

This page titled 8.4: Example- Counting Frequencies of Letters is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

8.5: Array Algorithms- Sorting

This page titled 8.5: Array Algorithms- Sorting is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

8.6: Array Algorithms- Searching

This page titled 8.6: Array Algorithms- Searching is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

8.7: Two-Dimensional Arrays

8.7: Two-Dimensional Arrays is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

8.8: Multidimensional Arrays (Optional)

8.8: Multidimensional Arrays (Optional) is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

8.9: Object-Oriented Design- Polymorphic Sorting (Optional)

8.9: Object-Oriented Design- Polymorphic Sorting (Optional) is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

8.10: From the Java Library- java.util.Vector

8.10: From the Java Library- java.util.Vector is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

8.11: Case Study- An N-Player Computer Game

8.11: Case Study- An N-Player Computer Game is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

8.12: A GUI-Based Game (Optional Graphics)

8.12: A GUI-Based Game (Optional Graphics) is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

CHAPTER OVERVIEW

9: Exceptions- When Things Go Wrong

Learning Objectives

After studying this chapter, you will

- Understand Java's exception-handling mechanisms.
- Be able to use the Java try/catch statement.
- Know how to design effective exception handlers.
- Appreciate the importance of exception handling in program design.
- Be able to design your own Exception subclasses.

Introduction

Mistakes happen. Making mistakes is the norm rather than the exception. This is not to say that we make mistakes more often than we get it right. It is to say that (almost) nothing we do or build is ever perfectly correct, least of all computer software. No matter how well-designed a program is, there is always the chance that some kind of error will arise during its execution.

An **exception** is an erroneous or anomalous condition that arises while a program is running. Examples of such conditions that we have discussed in this text include attempting to divide by 0 (arithmetic exception), reading a decimal value when an integer is expected (number format exception), attempting to write to a file that doesn't exist (I/O exception), or referring to a nonexistent character in a string (index out of bounds exception). The list of potential errors and anomalies is endless.

A well-designed program should include code to guard against errors and other exceptional conditions when they arise. This code should be incorporated into the program from the very first stages of its development. That way it can help identify problems during development. In Java, the preferred way of handling such conditions is to use **exception handling**, a divide-and-conquer approach that separates a program's normal code from its error-handling code.

This chapter describes Java's exception handling features. We begin by contrasting the traditional way of handling errors within a program with Java's default exception-handling mechanism. We show how exceptions are raised (thrown) and handled (caught) within a program and identify the rules that apply to different kinds of exceptions. We then focus on some of the key design issues that govern when, where, and how to use exceptions in your programs. Finally, we show how to design and implement one's own Exception subclass.

9.1: Introduction
9.2: Handling Exceptional Conditions
9.3: Java's Exception Hierarchy
9.4: Handling Exceptions Within a Program
9.5: Error Handling and Robust Program Design
9.6: Creating and Throwing Your Own Exceptions
9.7: From the Java Library- javax.swing.JOptionPane

This page titled 9: Exceptions- When Things Go Wrong is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

9.1: Introduction

9.1: Introduction is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

9.2: Handling Exceptional Conditions

```
 /**
  * Precondition:  N > 0
  * Postcondition: avgFirstN() = (1+2+...+N)/N
  */
public double avgFirstN(int N) {
    int sum = 0;
    for (int k = 1; k <= N; k++)
        sum += k;
    return sum/N;         // What if N is 0?
} // avgFirstN()
```

To introduce you to handling exceptional conditions, Figure [fig-intavg1] shows a method that computes the average of the first *N* integers, an admittedly contrived example. We use it mainly to illustrate the basic concepts involved in exception handling. As its precondition suggests, the avgFirstN() method expects that *N* will be greater than 0. If *N* happens to be 0, an error will occur in the expression sum/N, because you cannot divide an integer by 0.

Traditional Error Handling

Obviously, the method in Figure [fig-intavg1] should not simply ignore the possibility that *N* might be 0. Figure [fig-traditional] shows a revised version of the method, which includes code that takes action if the method's precondition fails. Because there is no way to compute an average of 0 elements, the revised method decides to abort the program. Aborting the program appears to be a better alternative than returning 0 or some other default value (like -1) as the method's result and thereby allowing an erroneous value to spread throughout the program. That would just compound the error.

```
 /**
  * Precondition:  N > 0
  * Postcondition: avgFirstN() equals (1+2+...+N) divided by N
  */
public double avgFirstN(int N) {
    int sum = 0;
    if (N <= 0) {
      System.out.println(
           "ERROR avgFirstN: N <= 0. Program terminating.");
      System.exit(0);
    }
    for (int k = 1; k <= N; k++)
        sum += k;
    return sum/N;         // What if N is 0?
} // avgFirstN()
```

The revised avgFirstN() method takes the traditional approach to error handling: Error-handling code is built right into the algorithm. If *N* happens to be 0 when avgFirstN() is called, the following output will be generated:

```
ERROR avgFirstN: N <= 0. Program terminating.
```

Java's Default Exception Handling

To help detect and handle common runtime errors, Java's creators incorporated an exception-handling model into the language itself. In the case of our divide-by-zero error, the Java Virtual Machine (JVM) would detect the error and abort the program. To see

this, consider the program in Figure [fig-calcavg]. Note that the avgFirstN() method is passed an argument of 0 in the CalcAvgTest.main(). When the JVM detects the error, it will abort the program and print the following message:

```
Exception in thread "main"
   java.lang.ArithmeticException:  / by zero
        at CalcAverage.avgFirstN(Compiled Code)
        at CalcAvgTest.main(CalcAvgTest.java:5)
```

The error message describes the error and provides a trace of the method calls, from last to first, that led to the error. This trace shows that the error occurred in the CalcAverage.avgFirstN() method, which was called by the CalcAvgTest.main() method.

```
public class CalcAverage {

    public double avgFirstN(int N) {
        int sum = 0;
        for (int k = 1; k <= N; k++)
            sum += k;
        return sum/N;         // What if N is 0?
    } // avgFirstN()
}//CalcAverage

public class CalcAvgTest {
    public static void main(String args[]) {
        CalcAverage ca = new CalcAverage();
        System.out.println( "AVG + " + ca.avgFirstN(0) );
    }//main
}//CalcAvgTest
```

As this example suggests, Java's default exception handling is able to detect and handle certain kinds of errors and exceptional conditions. In the next section, we will identify what kinds of conditions are handled by the JVM.

This page titled 9.2: Handling Exceptional Conditions is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **10.2: Handling Exceptional Conditions** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

9.3: Java's Exception Hierarchy

[pg-sec-excepthier]

The Java class library contains a number of predefined exceptions, some of which are shown in Figure [fig-excepthier]. The most general type of exception, java.lang.Exception, is located in the java.lang package, but most of its subclasses are contained in other packages. Some of the various IOException classes are contained in the java.io package, while others are contained in the java.net package. In general, exception classes are placed in the package that contains the methods that throw those exceptions.

Each of the classes in Figure [fig-excepthier] identifies a particular type of exception, and each is a subclass of the Exception class. Obviously a subclass defines a more specific exception than its superclass. Thus, both ArrayIndexOutOfBoundsException and StringIndexOutOfBoundsException are more specific than IndexOutOfBoundsException.

ll
Class&Description

ArithmeticException&Division by zero or some other kind of arithmetic problem&An array index is less than zero or greater &than or equal to the array's length&Reference to a file that cannot be found&Calling a method with an improper argument&An array or string index is out of bounds&Reference to an object that has not been instantiated&Use of an illegal number format, such as when calling a method&A String index is less than zero or greater than &or equal to the String's length

Table 10.1 gives a brief summary of some of the most important exceptions. You've undoubtedly encountered some of these exceptions, because they are thrown by methods we have used repeatedly in programming examples. Table 10.2 summarizes the exceptions raised by some of the methods we've used most frequently.

llll
Class& Method & Exception Raised & Description

Double&valueOf(String)&NumberFormatException&The String is not a double&parseInt(String)&NumberFormatException&The String is not a int&String(String)&NullPointerException&The String is null &indexOf(String)&NullPointerException&The String is null &lastIndexOf(String)&NullPointerException&The String is null &charAt(int)&StringIndexOutOfBoundsException&The int is not a valid index &substring(int)&StringIndexOutOfBoundsException&The int is not a valid index &substring(int,int)&StringIndexOutOfBoundsException&An int is not a valid index

What type of exception would be thrown for the following statements?

Integer.parseInt("26.2");

String s; s.indexOf('a');

String s = "hello"; s.charAt(5);

Checked and Unchecked Exceptions

Java's exception hierarchy is divided into two types of exceptions. A **checked exception** is one that can be analyzed by the Java compiler. Checked exceptions are thrown by methods such as the BufferedReader.readLine() method, in which there is a substantial likelihood that something might go wrong. When the compiler encounters one of these method calls, it checks whether the program either handles or declares the exception. Compile-time checking for these exceptions is to reduce the number of exceptions that are not properly handled within a program. This improves the security of Java programs.

The throws Clause

The IOException, which we encountered in Chapter 4 , is a checked exception. The Java compiler knows that readLine() is a method that can throw an IOException. A method that contains an expression that might throw a checked exception must either handle the exception or declare it. Otherwise, the compiler would generate a syntax error. The simplest way to avoid such a syntax error is to *declare the exception*, in our case that means qualifying the method header with the expression throws IOException.

In general, any method that contains an expression that might throw a checked expression must declare the exception. However, because one method can call another method, declaring exceptions can get a little tricky. If a method calls another method that

contains an expression that might throw an unchecked exception, then both methods must have a throws clause. For example, consider the following program:

```
import java.io.*;
public class Example {
    BufferedReader input = new BufferedReader
            (new InputStreamReader(System.in));
    public void doRead() throws IOException {
        // May throw IOException
        String inputString = input.readLine();
    }
    public static void main(String argv[])
                        throws IOException {
        Example ex = new Example();
        ex.doRead();
    }
}
```

In this case, the doRead() method contains a readLine() expression, which might throw an IOException. Therefore, the doRead() method must declare that it throws IOException. However, because doRead() is called by main(), the main() method must also declare the IOException.

The alternative approach would be to *catch* the IOException within the body of the method. We will discuss this approach in the next section.

Unchecked Exceptions

An **unchecked exception** is any exception belonging to a subclass of RuntimeException (Fig. [fig-excepthier]). Unchecked exceptions are not checked by the compiler. The possibility that some statement or expression will lead to an ArithmeticException or NullPointerException is extremely difficult to detect at compile time. The designers of Java decided that forcing programmers to declare such exceptions would not significantly improve the correctness of Java programs.

Therefore, unchecked exceptions do not have to be handled within a program. And they do not have to be declared in a throws clause. As shown in the chapter's early divide-by-zero exception example, unchecked exceptions are handled by Java's default exception handlers, unless your program takes specific steps to handle them directly. In many cases leaving the handling of such exceptions up to Java may be the best course of action, as we will see Section 5.

The Exception Class

The java.lang.Exception class itself is very simple, consisting of just two constructor methods (Fig. 10.5). The Throwable class, from which Exception is derived, is the root class of Java's exception and error hierarchy. It contains definitions for the getMessage() and printStackTrace() methods, which are two methods that we will use frequently in our error-handling routines.

Which of the following are examples of unchecked

IOException

IndexOutOfBoundsException

NullPointerException

ClassNotFoundException

NumberFormatException

This page titled 9.3: Java's Exception Hierarchy is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **10.3: Java's Exception Hierarchy** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

9.4: Handling Exceptions Within a Program

This section will describe how to handle exceptions within the program rather than leaving them to be handled by the JVM.

Trying, Throwing, and Catching an Exception

In Java, errors and other abnormal conditions are handled by throwing and catching exceptions. When an error or an exceptional condition is detected, you can *throw an exception* as a way of signaling the abnormal condition. This is like pulling the fire alarm. When an exception is thrown, an exception handler will catch the exception and deal with it (Fig. 10.6). We will discuss try blocks, which typically

are associated with catching exceptions, later in the section.

If we go back to our avgFirstN() example, the typical way of handling this error in Java would be to throw an exception in the method and catch it in the calling method. Of course, the calling method could be in the same object or it could belong to some other object. In the latter case, the detection of the error is separated from its handling. This division of labor opens up a wide range of possibilities. For example, a program could dedicate a single object to serve as the handler for all its exceptions. The object would be sort of like the program's fire department.

To illustrate Java's try/throw/catch mechanism, let's revisit the CalcAvgTest program. The version shown in Figure [fig-calcavg2] mimics the way Java's default exception handler works. If the avgFirstN() method is called with an argument that is zero or negative, an IllegalArgumentException is thrown. The exception is caught by the catch clause in the CalcAvgTest.main() method.

```
public class CalcAverage {
  /**
   * Precondition:  N > 0
   * Postcondition: avgFirstN() equals the average of (1+2+...+N)
   */
  public double avgFirstN(int N) {
    int sum = 0;
    if (N <= 0)
      throw new IllegalArgumentException("ERROR: Illegal argument");
    for (int k = 1; k <= N; k++)
      sum += k;
    return sum/N;         // What if N is 0?
  } // avgFirstN()
} // CalcAverage

public class CalcAvgTest {
  public static void main(String args[]) {
    try {
      CalcAverage ca = new CalcAverage();
      System.out.println( "AVG + " + ca.avgFirstN(0));
    }
    catch (IllegalArgumentException e) { // Exception Handler
      System.out.println(e.getMessage());
      e.printStackTrace();
      System.exit(0);
    }
  }//main
}// CalcAvgTest
```

Let's go through this example step by step. The first thing to notice is that if the CalcAverage.avgFirstN() method has a zero or negative argument, it will throw an exception:

```
if (N <= 0)
  throw new IllegalArgumentException("ERROR: Illegal argument");
```

Note the syntax of the throw statement. It creates a new IllegalArgumentException object and passes it a message that describes the error. This message becomes part of the exception object. It can be retrieved using the getMessage() method, which is inherited from the Throwable class (Fig. [fig-excepthier]).

When a throw statement is executed, the JVM interrupts the normal execution of the program and searches for an exception handler. We will describe the details of this search shortly. In this case, the exception handler is the catch clause contained in the CalcAvgTest.main() method:

```
catch (IllegalArgumentException e) {// Exception Handler
   System.out.println(e.getMessage());
   e.printStackTrace();
   System.exit(0);
}
```

When an IllegalArgumentException is thrown, the statements within this catch clause are executed. The first statement uses the getMessage() method to print a copy of the error message. The second statement uses the printStackTrace() method, which is defined in Throwable and inherited by all Exceptions, to print a trace of the method calls leading up to the exception. The last statement causes the program to terminate.

When we run this program, the following output will be generated as a result of the illegal argument error:

```
ERROR: Can't average 0 elements
java.lang.IllegalArgumentException: ERROR: Illegal argument
   at java.lang.Throwable.fillInStackTrace(Native Method)
   at java.lang.Throwable.<init>(Throwable.java:94)
   at java.lang.Exception.<init>(Exception.java:42)
   at java.lang.RuntimeException.<init>
                     (RuntimeException.java:47)
   at java.lang.IllegalArgumentException.<init>
                     (IllegalArgumentException.java:43)
   at CalcAverage.avgFirstN(Compiled Code)
   at CalcAvgTest.main(CalcAvgTest.java:5)
```

Thus, as in the previous example of Java's default exception handler, our exception handler also prints out a description of the error and a trace of the method calls that led up to the error. However, in this example, we are directly handling the exception rather than leaving it up to Java's default exception handler. Of course, this example is intended mainly for illustrative purposes. It would make little sense to write our own exception handler if it does nothing more than mimic Java's default handler.

Finally, note that the catch clause is associated with a try block. The handling of exceptions in Java takes place in two parts: First, we *try* to execute some statements, which may or may not lead to an exception. These are the statements contained within the try clause:

```
try {
    CalcAverage ca = new CalcAverage();
    System.out.println( "AVG + " + ca.avgFirstN(0));
}
```

Second, we provide one or more catch clauses to handle particular types of exceptions. In this case, we are only handling IllegalArgumentExceptions.

As we said earlier, throwing an exception is like pulling a fire alarm. The throw occurs somewhere within the scope of the try block. The “fire department” in this case is the code contained in the catch clause that immediately follows the try block. This is the exception handler for this particular exception. There’s something like a game of catch going on here: Some method within the try block throws an Exception object, which is caught and handled by the catch block located in some other object (Fig. [fig-exccatch]).

Separating Error Checking from Error Handling

As we see in the CalcAvgTest example, an important difference between Java’s exception handling and more traditional approaches is that error handling can be separated from the normal flow of execution within a program. The CalcAverage.avgFirstN() method still checks for the error and it still throws IllegalArgumentException if *N* does not satisfy the method’s precondition. But it does not contain code for handling the exception. The exception-handling code is located in the CalcAvgTest class.

Thus, the CalcAvgTest program creates a clear separation between the normal algorithm and the exception-handling code. One advantage of this design is that the normal algorithm is uncluttered by error-handling code and, therefore, easier to read.

Another advantage is that the program’s response to errors has been organized into one central location. By locating the exception handler in CalcAvgTest.main(), one exception handler can be used to handle other errors of that type. For example, this catch clause could handle *all* IllegalArgumentExceptions that get thrown in the program. Its use of printStackTrace() will identify exactly where the exception occurred. In fact, because a Java application starts in the main() method, encapsulating all of a program’s executable statements within a single try block in the main() method will effectively handle all the exceptions that occur within a program.

Syntax and Semantics of Try/Throw/Catch

A **try block** begins with the keyword try followed by a block of code enclosed within curly braces. A *catch clause* or **catch block** consists of the keyword catch, followed by a parameter declaration that identifies the type of Exception being caught, followed by a collection of statements enclosed within curly braces. These are the statements that handle the exception by taking appropriate action.

Once an exception is thrown, control is transferred out of the try block to an appropriate catch block. Control does not return to the try block.

The complete syntax of the try/catch statement is summarized in Figure [fig-trystmt]. The try block is meant to

```
try {
     // Block of statements
     // At least one of which may throw an exception
     if ( /* Some condition obtains */ )
         throw new ExceptionName();
} catch (ExceptionName ParameterName) {
    // Block of statements to be executed
    // If the ExceptionName exception is thrown in try
}  catch (ExceptionName2 ParameterName) {
    // Block of statements to be executed
    // If the ExceptionName2 exception is thrown in try
...  // Possibly other catch clauses
} finally {
    // Optional block of statements that is executed
    // Whether an exception is thrown or not
}
```

include a statement or statements that might throw an exception. The catch blocks—there can be one or more—are meant to handle exceptions that are thrown in the try block. A catch block will handle any exception that matches its parameter class, including subclasses of that class. The **finally block** clause is an optional clause that is always executed, whether an exception is thrown or not.

The statements in the try block are part of the program's normal flow of execution. By encapsulating a group of statements within a try block, you thereby indicate that one or more exceptions may be thrown by those statements, and that you intend to catch them. In effect, you are *trying* a block of code with the possibility that something might go wrong.

If an exception is thrown within a try block, Java exits the block and transfers control to the first catch block that matches the particular kind of exception that was thrown. Exceptions are thrown by using the throw statement, which takes the following general form:

```
throw new ExceptionClassName(OptionalMessageString);
```

The keyword throw is followed by the instantiation of an object of the ExceptionClassName class. This is done the same way we instantiate any object in Java: by using the new operator and invoking one of the exception's constructor methods. Some of the constructors take an OptionalMessageString, which is the message that gets returned by the exception's getMessage() method.

A catch block has the following general form:

```
catch (ExceptionClassName ParameterName) {
    // Exception handling statements
}
```

A catch block is very much like a method definition. It contains a parameter, which specifies the class of exception that is handled by that block. The *ParameterName* can be any valid identifier, but it is customary to use e as the catch block parameter. The parameter's scope is limited to the catch block, and it is used to refer to the caught exception.

The *ExceptionClassName* must be one of the classes in Java's exception hierarchy (see Fig. [fig-excepthier]). A thrown exception will match any parameter of its own class or any of its superclasses. Thus, if an ArithmeticException is thrown, it will match both an ArithmeticException parameter and an Exception parameter, because ArithmeticException is a subclass of Exception.

Note that there can be multiple catch clauses associated with a given try block, and the order with which they are arranged is important. A thrown exception will be caught by the first catch clause it matches. Therefore, catch clauses should be arranged in order from most specific to most general (See the exception hierarchy in Figure [fig-excepthier]). If a more general catch clause precedes a more specific one, it will prevent the more specific one from executing. In effect, the more specific clause will be hidden by the more general one. You might as well just not have the more specific clause at all.

To illustrate how to arrange catch clauses, suppose an ArithmeticException is thrown in the following try/catch statement:

```
try {
     // Suppose an ArithmeticException is thrown here
} catch (ArithmeticException e) {
     System.out.println("ERROR: " + e.getMessage() );
     e.printStackTrace();
     System.exit(1);
} catch (Exception e) {
     System.out.println("ERROR: " + e.getMessage() );
}
```

In this case, the exception would be handled by the more specific ArithmeticException block. On the other hand, if some other kind of exception is raised, it will be caught by the second catch clause. The Exception class will match any exception that is thrown. Therefore, it should always occur last in a sequence of catch clauses.

Restrictions on the try/catch/finally Statement

There are several important restrictions that apply to Java's exception-handling mechanism. We'll describe these in more detail later in this chapter.

A try block must be immediately followed by one or more catch clauses and a catch clause may only follow a try block.

A throw statement is used to throw both checked exceptions and unchecked exceptions, where unchecked exceptions are those belonging to RuntimeException or its subclasses. Unchecked exceptions need not be caught by the program.

A throw statement must be contained within the dynamic scope of a try block, and the type of Exception thrown must match at least one of the try block's catch clauses. Or the throw statement must be contained within a method or constructor that has a throws clause for the type of thrown Exception.

Dynamic Versus Static Scoping

How does Java know that it should execute the catch clause in CalcAvgTest.main() when an exception is thrown in avgFirstN()? Also, doesn't the latest version of avgFirstN() (Fig. [fig-calcavg2]) violate the restriction that a throw statement must occur within a try block?

An exception can only be thrown within a *dynamically enclosing* try block. This means that the throw statement must fall within the **dynamic scope** of an enclosing try block. Let's see what this means.

Dynamic scoping refers to the way a program is executed. For example, in CalcAverage (Fig. [fig-calcavg2]), the avgFirstN() method is called from within the try block located in CalcAvgTest.main(). Thus, it falls within the dynamic scope of that try block.

Contrast dynamic with what you have learned about **static scope**, which we've used previously to define the scope of parameters and local variables (Fig. [fig-dynamic]). Static scoping refers to the way a program is written. A statement or variable occurs within the scope of a block if its text is actually written within that block. For example, consider the definition of MyClass (Fig. [fig-myclass]). The variable X occurs within the (static) scope of method1(), and the variable Y occurs within the (static) scope of method2().

A method's parameters and local variables occur within its static scope. Also, in the MyClass definition, the System.out.println() statements occur within the static scope of method1() and method2(), respectively. In general, static scoping refers to where a variable is declared or where a statement is located. Static scoping can be completely determined by just reading the program.

Dynamic scoping can only be determined by running the program. For example, in the order in which its statements are executed depends on the result of Math.random(). Suppose that when random() is executed it returns the value 0.99. In that case, main() will call method2(), which will call System.out.println(), which will print "Hello2." In that case, the statement System.out.println("Hello" + Y) has the following dynamic scope:

```
main()
    method2()
        System.out.println("Hello" + Y);
```

```
public class MyClass {
    public void method1() {
        int X = 1;
        System.out.println("Hello" + X);
    }
    public void method2() {
        int Y = 2;
        System.out.println("Hello" + Y);
    }
    public static void main( String argv[] ) {
        MyClass myclass = new MyClass();
        if (Math.random() > 0.5)
            myclass.method2();
        else
            myclass.method1();
    }
} // MyClass
```

It occurs within the (dynamic) scope of method2(), which is within the (dynamic) scope of main(). On the other hand, if the result of random() had been 0.10, that particular println() statement wouldn't have been executed at all. Thus, to determine the dynamic scope of a particular statement, you must trace the program's execution. In fact, this is what the printStackTrace() method does. It prints a trace of a statement's dynamic scope.

Exception Propagation: Searching for a Catch Block

When an exception is thrown, Java uses both static and dynamic scoping to find a catch clause to handle it. Java knows how the program is defined—after all, it compiled it. Thus, the static scope of a program's methods is determined by the compiler. Java also places a record of every method call the program makes on a method call stack. A **method call stack** is a data structure that behaves like a stack of dishes in the cafeteria. For each method call, a *method call block* is placed on top of the stack (like a dish), and when a particular method call returns, its block is removed from the top of the stack (Fig. [fig-methodstack]).

An important feature of the method call stack is that the current executing method is always represented by the top block on the method call stack. If an exception happens during that method call, you can trace backward through the method calls, if necessary, to find an exception handler for that exception. In Figure [fig-methodstack], you can visualize this back trace as a matter of reversing the direction of the curved arrows.

In order to find a matching catch block for an exception, Java uses its knowledge of the program's static and dynamic scope to perform a **method stack trace**. The basic idea is that Java traces backward through the program until it finds an appropriate catch clause. The trace begins within the block that threw the exception. Of course, one block can be nested (statically) within another block. If the exception is not caught by the block in which it is thrown, Java searches the enclosing block. This is static scoping. If it is not caught within the enclosing block, Java searches the next higher enclosing block, and so on. This is still static scoping.

If the exception is not caught at all within the method in which it was thrown, Java uses the method call stack (Fig. [fig-methodstack]) to search backward through the method calls that were made leading up to the exception. This is dynamic scoping. In the case of our CalcAvgTest() example (Fig. [fig-calcavg2]), Java would search backward to the CalcAvgTest.main() method, which is where avgFirstN() was called, and it would find the catch clause there for handling IllegalArgumentExceptions. It would, therefore, execute that catch clause.

Suppose a program throws an ArrayIndexOutOfBoundsException. Using the exception hierarchy in Figure [fig-excepthier], determine which of the following catch clauses could handle that exception.

catch (RunTimeException e)

catch (StringIndexOutOfBoundsException e)

catch (IndexOutOfBoundsException e)

catch (Exception e)

catch (ArrayStoreException e)

In the program that follows suppose that the first time random() is called it returns 0.98, and the second time it is called it returns 0.44. What output would be printed by the program?

```
class MyClass2 {
    public void method1(double X) {
        if (X > 0.95)
            throw new ArithmeticException(X
                    + " is out of range");
        System.out.println("Hello " + X);
    }
    public void method2(double Y) {
        if (Y > 0.5)
             throw new ArithmeticException(Y
                    + " is out of range");
        System.out.println("Hello " + Y);
    }
    public static void main(String argv[]) {
        MyClass2 myclass = new MyClass2();
        try {
            myclass.method1(Math.random());
            myclass.method2(Math.random());
        } catch (ArithmeticException e) {
            System.out.println(e.getMessage());
        }
    } // main()
} // MyClass2
```

For the values returned by random() in the previous exercise, show what would be output if printStackTrace() were called in addition to printing an error message.

In the MyClass2 program, suppose that the first time random() is called it returns 0.44, and the second time it is called it returns 0.98. What output would be printed by the program?

For the values returned by random() in the previous exercise, show what would be output if printStackTrace() were called instead of printing an error message.

Find the divide-by-zero error in the following program, and then show what stack trace would be printed by the program:

```
public class BadDivide {
    public void method1 (int n) {
        method2(100, n);
    }
    public void method2 (int n, int d) {
         System.out.println(n / d);
    }
    public static void main(String args[]) {
        BadDivide bd = new BadDivide();
        for (int k = 0; k < 5; k++)
            bd.method1(k);
    }
}
```

Modify method2() so that it handles the divide-by-zero exception itself, instead of letting Java handle it. Have it print an error message and a stack trace.

What would be printed by the following code segment if someValue equals 1000?

```
int M = someValue;
try {
    System.out.println("Entering try block");
    if (M > 100)
        throw new Exception(M + " is too large");
    System.out.println("Exiting try block");
} catch (Exception e) {
    System.out.println("ERROR: " + e.getMessage());
}
```

What would be printed by the code segment in the preceding question if someValue equals 50?

Write a try/catch block that throws an Exception if the value of variable X is less than zero. The exception should be an instance of Exception and, when it is caught, the message returned by getMessage() should be “ERROR: Negative value in X coordinate.”

This page titled 9.4: Handling Exceptions Within a Program is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **10.4: Handling Exceptions Within a Program** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

9.5: Error Handling and Robust Program Design

[pg-sec-robust] An important element of program design is to develop appropriate ways of handling erroneous and exceptional conditions. As we have seen, the JVM will catch any unchecked exceptions that are not caught by the program itself. For your own (practice) programs, the best design may simply be to use Java's default exception handling. The program will terminate when an exception is thrown, and then you can debug the error and recompile the program.

On the other hand, this strategy would be inappropriate for commercial software, which cannot be fixed by its users. A well-designed commercial program should contain exception handlers for those truly exceptional conditions that may arise.

In general there are three ways to handle an exceptional condition that isn't already handled by Java (Table 10.3). If the

lll
Kind of Exception&Kind of Program&Action to Be Taken

Caught by Java&&Let Java handle it Fixable condition&&Fix the error and resume execution Unfixable condition&Stoppable&Report the error and terminate
&&the program Unfixable condition&Not stoppable&Report the error and resume
&&processing

exceptional condition cannot be fixed, the program should be terminated, with an appropriate error message. Second, if the exceptional condition can be fixed without invalidating the program, then it should be remedied and the program's normal execution should be resumed. Third, if the exception cannot be fixed, but the program cannot be terminated, the exceptional condition should be reported or logged in some way, and the program should be resumed.

Print a Message and Terminate

Our illegal argument example is a clear case in which the exception is best handled by terminating the program. In this case, this particular error is best left to Java's default exception handling, which will terminate the program when the exception is thrown. There is simply no way to satisfy the postcondition of the avgFirstN() method when *N* is less than or equal to 0. This type of error often calls attention to a design flaw in the program's logic that should be caught during program development. The throwing of the exception helps identify the design flaw.

Similar problems can (and often do) arise in connection with errors that are not caught by Java. For example, suppose that your program receives an erroneous input value, whose use would invalidate the calculation it is making. This won't be caught by Java. But it should be caught by your program, and an appropriate alternative here is to report the error and terminate the program. Fixing this type of error may involve adding routines to validate the input data before they are used in the calculation.

In short, rather than allowing an erroneous result to propagate throughout the program, it is best to terminate the program.

Log the Error and Resume

Of course, the advice to stop the program assumes that the program can *be* terminated reasonably. Some programs—such as programs that monitor the space shuttle or programs that control a nuclear magnetic resonance (NMR) machine—cannot (and should not) be terminated because of such an error.

Such programs are called *failsafe* because they are designed to run without termination. For these programs, the exception should be reported in whatever manner is most appropriate, but the program should continue running. If the exceptional condition invalidates the program's computations, then the exception handler should make it clear that the results are tainted.

Other programs—such as programs that analyze a large transaction database—should be designed to continue processing after catching such errors. For example, suppose the program a large airline runs a program once a day to analyze the ticketing transactions that took place. This kind of program might use exceptions to identify erroneous transactions or transactions that involve invalid data of some sort. Because there are bound to be many errors of this kind in the database, it is not reasonable to stop the program. This kind of program shouldn't stop until it has finished processing all of the transactions. An appropriate action for this kind of program is to log the exceptions into some kind of file and continue processing the transactions.

Suppose a divide-by-zero error happened in one of these programs. In that case, you would override Java's default exception handling to ensure that the program is *not* terminated. More generally, it's important that these types of programs be designed to

catch and report such exceptions. This type of exception handling should be built right into the program's design.

Fix the Error and Resume

As an example of a problem that can be addressed as the program runs, consider the task of inputting an integer into a text field. As you have probably experienced, if a program is expecting an integer and you attempt to input something beside an integer, a NumberFormatException is generated and the program will terminate. For example, if you enter "$55" when prompted to input an integer dollar amount, this will generate an exception when the Integer.parseInt() method is invoked. The input string cannot be parsed into a valid int. However, this is the kind of error that can be addressed as the program is running.

Let's design a special IntField that functions like a normal text field but accepts only integers. If the user enters a value that generates a NumberFormatException, an error message should be printed and the user should be invited to try again. As

Figure 10.13 shows, we want this special field to be a subclass of JTextField and to inherit the basic JTextField functionality. It should have the same kind of constructors that a normal JTextField has. This leads to the definition shown in Figure [fig-intfieldprog].

```
import javax.swing.*;

public class IntField extends JTextField {
    public IntField () {
        super();
    }
    public IntField (int size) {
        super(size);
    }
    public int getInt() throws NumberFormatException {
        return Integer.parseInt(getText());
    } // getInt()
} // IntField
```

Note that the constructor methods use super to call the JTextField constructor. For now, these two constructors should suffice. However, later we will introduce a third constructor that allows us to associate a bound with the IntField later in this chapter.

Our IntField class needs a method that can return its contents. This method should work like JTextField.getText(), but it should return a valid integer. The getInt() method takes no parameters and will return an int, assuming that a valid integer is typed into the IntField. If the user enters "$55," a NumberFormatException will be thrown by the Integer.parseInt() method. Note that getInt() declares that it throws this exception. This is not necessary because a NumberFormatException is not a checked exception, but it makes the code clearer.

Where and how should this exception be handled? The exception cannot easily be handled within the getInt() method. This method has to return an integer value. If the user types in a non-integer, there's no way to return a valid value. Therefore, it's better just to throw the exception to the calling method, where it can be handled more easily.

In a GUI application or applet, the calling method is likely to be an actionPerformed() method, such as the following:

```
public void actionPerformed(ActionEvent e) {
  try {
    userInt = intField.getInt();
    message = "You input " + userInt + " Thank you.";
  } catch (NumberFormatException ex) {
    JOptionPane.showMessageDialog(this,
     "The input must be an integer.  Please re-enter.");
  } finally {
      repaint();
  }
} // actionPerformed()
```

The call to getInt() is embedded in a try/catch block. This leads to the design summarized in Figure [fig-numformat]. The IntField throws an exception that is caught by the GUI, which then displays an error message.

If the user inputs a valid integer, the program will just report a message that displays the value. A more real-world example would make a more significant use of the value. On the other hand, if the user types an erroneous value, the program will pop up the dialog box shown in Figure [fig-errmsg]. (See the "From the Library" section of this chapter for more on dialog boxes.) When the user clicks the OK button, the program will resume normal execution, so that when an exception is raised, the enter value is not used, and no harm is done by an erroneous value. The user can try again to input a valid integer. Note that the finally clause repaints the GUI. In this case, repainting would display the appropriate message on the applet or the application.

This is an example of what we might call *defensive* design. Defensive design is when we anticipate a possible input error and take steps to ensure that a bad value is not propagated throughout the program.

Admittedly, the sense in which the error here is "fixed" is simply that the user's original input is ignored and reentered. This is a legitimate and simple course of action for this particular situation. It is far preferable to ignoring the exception. If the program does not handle this exception itself, Java will catch it and will print a stack trace and terminate the program. That would not be a very user-friendly interface!

Clearly, this is the type of exceptional condition that should be anticipated during program design. If this happens to be a program designed exclusively for your own use, then this type of exception handling might be unnecessary. But if the program is meant to be used by others, it is important that the program be able to handle erroneous user input without crashing.

To Fix or Not to Fix

Let's now consider a problem where it is less clear whether an exception can be successfully fixed "on the fly." Suppose you have a program that contains an array of Strings, which is initially created with just two elements.

```
String list[] = new String[2];
```

If an attempt is made to add more than two elements to the array, an will be raised. This exception can be handled by extending the size of the array and inserting the element. Then the program's normal execution can be resumed.

To begin creating such a program, let's first design a method that will insert a string into the array. Suppose that this is intended to be a private method that will only be used within the program. Also, let's suppose that the program maintains a variable, count, that tracks how many values have been stored in the array. Therefore, it will not be necessary to pass the array as a parameter. So, we are creating a void method with one parameter, the String to be inserted:

```
private void insertString(String str) {
     // Might throw ArrayIndexOutOfBoundsException
    list[count] = str;
    ++count;
}
```

The comment notes where an exception might be thrown.

Can we handle this exception? When this exception is raised, we could create a new array with one more element than the current array. We could copy the old array into the new array and then insert the String in the new location. Finally, we could set the variable list, the array reference, so that it points to the new array. Thus, we could use the following try/catch block to handle this exception:

```
private void insertString(String str) {
    try {
        list[count] = str;
    } catch (ArrayIndexOutOfBoundsException e) {
        // Create a new array
        String newList[] = new String[list.length+1];
        for (int k = 0; k < list.length; k++) // Copy array
            newList[k] = list[k];
        newList[count] = str;   // Insert into new array
        list = newList;         // Make old point to new
    } finally {         // Since the exception is now fixed
        count++;        // Increase the count
    }
} // insertString()
```

The effect of the catch clause is to create a new array, still referred to as list, but that contains one more element than the original array.

Note the use of the finally clause here. For this problem it's important that we increment count in the finally clause. This is the only way to guarantee that count is incremented exactly once whenever an element is assigned to the array.

The design of the FixArrayBound class is shown in Figure 10.17. It provides a simple GUI interface that enables you to test the insertString() method. This program has a standard Swing interface, using a JFrame as the top-level window. The program's components are contained within a JPanel that's added to the JFrame in the main() method.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class FixArrayBound extends JPanel
                           implements ActionListener  {
  public static final int WIDTH = 350, HEIGHT = 100;
  private JTextField inField = new JTextField(10);
  private JLabel prompt = new JLabel(
                     "Input a word and type <ENTER>: ");
                  // Initially list has 2 elements
  private String list[] = new String[2];
  private int count = 0;

  public  FixArrayBound() {
    inField.addActionListener(this);
    add(prompt);
    add(inField);
    setSize(WIDTH, HEIGHT);
  } // FixArrayBound()

  public void paintComponent(Graphics g) {
    g.setColor(getBackground());  // Clear the background
    g.fillRect(0, 0, WIDTH, HEIGHT);
    g.setColor(getForeground());
    String tempS = "";
    for (int k = 0; k < list.length; k++)
      tempS = tempS +  list[k] + " ";
    g.drawString(tempS, 10, 50);
  } // paintComponent

  private void insertString(String str) {
    try {
      list[count] = str;
    } catch (ArrayIndexOutOfBoundsException e) {
      String newList[] = new String[list.length+1]; // New array
      for (int k = 0; k < list.length; k++) // Copy old to new
        newList[k] = list[k];
        newList[count] = str; // Insert item into new
        list = newList;       // Make old point to new
    } finally {            // The exception is now fixed
     count++;              //  so increase the count
    }
  } // insertString()

  public void actionPerformed(ActionEvent evt) {
    insertString(inField.getText());
    inField.setText("");
    repaint();
  } // actionPerformed()
```

```
    public static void main( String args[] ) {
        JFrame f = new JFrame("Array Fixer");
        FixArrayBound panel = new FixArrayBound();
        f.getContentPane().add(panel);
        f.setSize(panel.WIDTH, panel.HEIGHT);
        f.setVisible(true);
        f.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);  // Quit the application
            }
        });
    } // main()
} // FixArrayBound
```

Each time the user types a string into the text field, the actionPerformed() method calls the insertString() method to add the string to the array. On each user action, the JPanel is repainted. The paintComponent() method simply clears the panel and then displays the array's elements (Fig. 10.18).

The complete implementation of FixArrayBound is given in Figure 1019. This example illustrates how an exception *can* be handled successfully and the program's normal flow of control resumed. However, the question is whether such an exception *should* be handled this way.

Unfortunately, this is not a well-designed program. The array's initial size is much too small for the program's intended use. Therefore, the fact that these exceptions arise at all is the result of poor design. In general, exceptions should *not* be used as a remedy for poor design.

For a program that uses an array, the size of the array should be chosen so that it can store all the objects required by the program. If the program is some kind of failsafe program, which cannot afford to crash, then something like the previous approach might be justified, provided this type of exception occurs very rarely. Even in that case it would be better to generate a message that alerts the program's user that this condition has occurred. The alert will indicate a need to modify the program's memory requirements and restart the program.

If it is not known in advance how many objects will be stored in an array, a better design would be to make use of the java.util.Vector class (see "From the Java Library" in Chapter 9). Vectors are designed to grow as new objects are inserted. In some ways the exception-handling code in our example mimics the behavior of a vector. However, the Vector class makes use of efficient algorithms for extending its size. By contrast, exception-handling code is very inefficient. Because exceptions force the system into an abnormal mode of execution, it takes considerably longer to handle an exception than it would to use a Vector for this type of application.

For each of the following exceptions, determine whether it can be handled in such a way that the program can be resumed or whether the program should be terminated:

A computer game program detects a problem with one of its GUI elements and throws a NullPointerException.

A factory assembly-line control program determines that an important control value has become negative and generates an ArithmeticException.

A company's Web-based order form detects that its user has entered an invalid String and throws a SecurityException.

This page titled 9.5: Error Handling and Robust Program Design is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **10.5: Error Handling and Robust Program Design** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

9.6: Creating and Throwing Your Own Exceptions

[pg-sec-throws]

Like other Java classes, the Exception class can be extended to handle cases that are not already covered by Java's built-in exceptions. Exceptions that you define will be handled the same way by the Java interpreter, but you will have to throw them yourself.

For example, Figure 10.20 shows the design of an exception that can be used for validating that an integer is less than or equal to a certain maximum value. It would be coded as follows:

```
 /**
  *  IntOutOfRangeException reports an exception when
  *    an integer exceeds its bound.
  */
public class IntOutOfRangeException extends Exception {

    public IntOutOfRangeException (int Bound) {
      super("The input value exceeds the bound " + Bound);
    }
}
```

The class extends Exception and consists entirely of a constructor method that calls the superclass constructor. The argument passed to the superclass constructor is the message that will be returned by getMessage() when an instance of this exception is created.

Now let's consider an example where this new exception will be thrown. Suppose we wish to constrain the IntField class that we developed previously (Fig. 10.14) so that it will only accept numbers that are less than a certain bound. First, let's modify IntField so that its bound can be set when an instance is created. We want its bound to be an instance variable with some initial value, and we want to provide a constructor that can be used to override the default (Fig. 10.21).

This leads to the following revision of IntField:

```
public class IntField extends JTextField {
    private int bound = Integer.MAX_VALUE;

    public IntField(int size, int max) {
        super(size);
        bound = max;
    }
   // The rest of the class is unchanged for now
} // IntField
```

Our new constructor has the signature IntField(int,int), which doesn't duplicate any of JTextField's constructors. This is good design, because in extending a class, we want to be careful about the effect that our definitions have on the original methods in the superclass. Superclass methods should be overridden by design, not by accident. If a method is redefined inadvertently, it might not function as expected by users of the subclass.

Note how we have handled the problem of setting the default value of the bound. Integer.MAX_VALUE is a class constant that sets the maximum value for the int type. It's an appropriate value to use, because any valid int that the user types should be less than or equal to MAX_VALUE. Given these changes to IntField, let's now incorporate our new exception into its getInt() method (Fig. [fig-intfield2]).

This new version of getInt() throws an exception if the integer entered by the user is greater than the IntField's bound. Here again, it is difficult to handle this exception appropriately in this method. The method would either have to return an erroneous value—

because it must return something—or it must terminate. Neither is an acceptable alternative. It is far better to throw the exception to the calling method.

```
import javax.swing.*;

public class IntField extends JTextField {
    private int bound = Integer.MAX_VALUE;

    public IntField (int size) {
        super(size);
    }

    public IntField(int size, int max) {
        super(size);
        bound = max;
    }

    public int getInt() throws NumberFormatException,
                               IntOutOfRangeException {
        int num = Integer.parseInt(getText());
        if (num > bound)
            throw new IntOutOfRangeException(bound);
        return num;
    } // getInt()

} // IntField
```

The IntFieldTester class (Fig. [fig-intbound]) has the design and functionality shown in Figure [fig-numformat]. It provides a simple GUI interface to test the IntField class. It prompts the user to type an integer that is less than 100, and then it echoes the user's input. Note how the exception is handled in the actionPerformed() method. If an exception is thrown in IntField.getInt(), the actionPerformed() method pops up an error dialog, and the erroneous input is not used. Instead, the user is given another chance to enter a valid integer.

-4.5pc

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class IntFieldTester extends JPanel
                                    implements ActionListener  {
  public static final int WIDTH = 300, HEIGHT = 300;
  private JLabel prompt = new JLabel("Input an integer <= 100: ");
  private IntField intField = new IntField(12, 100);
  private int userInt;
  private String message = "Hello";

  public IntFieldTester() {
    add(prompt);
    intField.addActionListener(this);
    add(intField);
    setSize(WIDTH, HEIGHT);
  } // IntFieldTester()
```

```
  public void paintComponent( Graphics g ) {
    g.setColor(getBackground());            // Clear the panel
    g.fillRect(0, 0, WIDTH, HEIGHT);
    g.setColor(getForeground());
    g.drawString(message, 10, 70);
  } // paintComponent()

  public void actionPerformed(ActionEvent evt) {
    try {
      userInt = intField.getInt();
      message = "You input " + userInt + " Thank you.";
    } catch (NumberFormatException e) {
      JOptionPane.showMessageDialog(this,
       "The input must be an integer.  Please reenter.");
    } catch (IntOutOfRangeException e) {
      JOptionPane.showMessageDialog(this, e.getMessage());
    } finally {
      repaint();
    }
  } // actionPerformed()

  public static void main(String args[]) {
    JFrame f = new JFrame("IntField Tester");
    IntFieldTester panel = new IntFieldTester();
    f.getContentPane().add(panel);
    f.setSize(panel.WIDTH, panel.HEIGHT);
    f.setVisible(true);
    f.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);   // Quit the application
      }
    });
  } // main()
} // IntFieldTester
```

Define a new Exception named FieldIsEmptyException, which is meant to be thrown if the user forgets to enter a value into a IntField.

Modify the IntField.getInt() method so that it throws and catches the FieldIsEmptyException.

This page titled 9.6: Creating and Throwing Your Own Exceptions is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **10.6: Creating and Throwing Your Own Exceptions** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

9.7: From the Java Library- javax.swing.JOptionPane

[pg-sec-dialogs]

is a window that can be opened by a program to communicate in some way with the user. Dialog boxes come in many varieties and have many uses in a GUI environment. You've undoubtedly encountered them when using your own computer.

For example, a *file dialog* is opened whenever you want to open or save a file. It provides an interface that lets you name the file and helps you search through the computer's directory structure to find a file.

A *warning dialog* or **error dialog** is opened whenever a program needs to notify or warn you that some kind of error occurred. It usually presents an error message and an OK button that you click to dismiss the dialog.

Dialogs are easy to create and use in Java. The Swing component set provides several different kinds of basic dialogs that can be incorporated into your program with one or two lines of code. For example, the IntFieldTester class makes use of a simple message dialog to report an input error to the user. This dialog was created by the following code segment in the program (see Figure [fig-intbound]):

```
catch (NumberFormatException e) {
  JOptionPane.showMessageDialog(this,
    "The input must be an integer.  Please reenter.");
}
```

This method call displays the window shown in Figure [fig-errmsg]. It contains the error message and an OK button that is used to close the window. The showMessageDialog() method is a static method of the javax.swing.JOptionPane class. This class provides a collection of similar methods for creating and displaying basic dialog boxes.

A dialog differs from other kinds of top-level windows—such as JApplet and JFrame—in that it is associated with another window (Fig. 1024). The first parameter in this version of the showMessageDialog() method is a reference to the dialog's parent window. The second parameter is a String representing the message.

The basic message dialog used in this example is known as a **modal dialog**. This means that once it's been displayed, you can't do anything else until you click the OK button and dismiss the dialog. It's also possible to create *nonmodal* dialogs. These can stay around on the screen while you move on to other tasks.

Note that the dialog box also contains an *icon* that symbolizes the purpose of the message (Fig. [fig-errmsg2]). The icon

is representative of the dialog's message type. Among the basic types available in JOptionPane are the following:

```
JOptionPane.PLAIN_MESSAGE
JOptionPane.INFORMATIONAL_MESSAGE     // Default
JOptionPane.WARNING_MESSAGE
JOptionPane.QUESTION_MESSAGE
JOptionPane.ERROR_MESSAGE
```

To set the dialog to anything other than the default (informational) type, you can use the following version of showMessageDialog():

```
showMessageDialog(Component comp, Object message,
                  String title, int msgType);
```

The first parameter is a reference to the parent window. The second is the message string. The third is a string used as the dialog window's title, and the fourth is one of the five dialog types. For example, we can change our dialog to an error dialog with the following statement:

```
catch (IntOutOfRangeException e) {
    JOptionPane.showMessageDialog(this,
            e.getMessage(),
            "Error dialog",
            JOptionPane.ERROR_MESSAGE);
}
```

This would produce the dialog shown in Figure [fig-errmsg2].

The other kinds of basic dialogs provided by the JOptionPane class are listed in Table 10.4. All of the dialogs listed there can be created with a line or two of code. In addition to these, it's also possible to create sophisticated dialogs that can be as customized as any other GUI interface you can build in Java.

ll
Dialog&Description

Message Dialog&Presents a simple error or informational message Confirm Dialog&Prompts the user to confirm a particular action Option Dialog&Lets the user choose from some options Input Dialog&Prompts and inputs a string

In this chapter, you have learned how to handle exceptional conditions that occur in programs. You now know that Java has a default exception handler that can take of many situations, and you also understand that proper program design using Java excpetion-handling elements helps deal with many other situations. This chapter continues the emphasis on good program design for creating useful, stable programs.

catch block

catch an exception

checked exception

dialog box

dynamic scope

error dialog

exception

exception handling

finally block

method call stack

method stack trace

modal dialog

static scope

throw an exception

try block

unchecked exception

The try/catch/finally statement has the following syntax:

```
try {
    // Block of statements
    // At least one of which may throw an exception

    if ( * Some condition obtains */ )
        throw new ExceptionName();
} catch (ExceptionName ParameterName) {
    // Block of statements to be executed
    // If the ExceptionName exception is thrown in try
}
..
} catch (ExceptionName2 ParameterName) {
    // Block of statements to be executed
    // If the ExceptionName2 exception is thrown in try
} finally {
    // Optional block of statements that is executed
    // Whether an exception is thrown or not
}
```

The try block is meant to include a statement or statements that might throw an exception. The catch blocks—there can be one or more—are meant to handle exceptions that are thrown in the try block. A catch block will handle any exception that matches its parameter class, including subclasses of that class. The finally block is optional. It will be executed whether an exception is thrown or not. If an exception is thrown in the try block, the try block is exited permanently.

The throw statement inside the try block is there to illustrate how throw can be used. You will usually not see a throw statement in a try block, because most throws are done from within Java library methods, which are called from a try block.

In Java, when an error or exceptional condition occurs, you throw an Exception, which is caught by special code known as an *exception handler*. A throw statement—throw new Exception()—is used to throw an exception.

A *try block* is a block of statements containing one or more statements that may throw an exception. Embedding a statement in a try block indicates your awareness that it might throw an exception and your intention to handle the exception.

Java distinguishes between *checked* and *unchecked* exceptions. Checked exceptions must either be caught by the method in which they occur or you must declare that the method containing that statement throws the exception.

Unchecked exceptions are those that belong to subclasses of RuntimeException. If they are left uncaught, they will be handled by Java's default exception handlers.

A *catch block* is a block of statements that handles the exceptions that match its parameter. A catch block can only follow a try block, and there may be more than one catch block for each try block.

The try/catch syntax allows you to separate the normal parts of an algorithm from special code meant to handle errors and exceptional conditions.

A *method stack trace* is a trace of the method calls that have led to the execution of a particular statement in the program. The Exception.printStackTrace() method can be called by exception handlers to print a trace of exactly how the program reached the statement that threw the exception.

Static scoping refers to how the text of the program is arranged. If a variable is declared within a method or a block, its static scope is confined to that method or block.

Dynamic scoping refers to how the program is executed. A statement is within the dynamic scope of a method or block if it is called from that method or block, or if it is called by some other method that was called from that method or block.

When searching for a catch block to handle an exception thrown by a statement, Java searches upward through the statement's static scope and backward through its dynamic scope until it finds a matching catch block. If none is found, the Java Virtual Machine will handle the exception itself by printing an error message and a method stack trace.

Many Java library methods throw exceptions when an error occurs. These throw statements do not appear in the program. For example, Java's integer division operator will throw an ArithmeticException if an attempt is made to divide by zero.

Generally, there are four ways to handle an exception: (1) Let Java handle it; (2) fix the problem that led to the exception and resume the program; (3) report the problem and resume the program; and (4) print an error message and terminate the program. Most erroneous conditions reported by exceptions are difficult or impossible to fix.

A finally statement is an optional part of a try/catch block. Statements contained in a finally block will be executed whether an exception is raised or not.

A well-designed program should use exception handling to deal with truly exceptional conditions, not as a means of normal program control.

User-defined exceptions can be defined by extending the Exception class or one of its subclasses.

1. Integer.parseInt("26.2"); ==> NumberFormatException
2. String s; s.indexOf('a'); ==> NullPointerException
3. String s = "hello"; s.charAt(5); ==> StringIndexOutOfBoundsException

The unchecked exceptions are IndexOutOfBoundsException, NumberFormatException, and NullPointerException, because these are subclasses of RuntimeException. The others are checked exceptions.

An ArrayIndexOutOfBoundsException could be handled by the handlers in a, c, or d, because their classes are all superclasses of ArrayIndexOutOfBoundsException.

If Math.random() in MyClass2 returns 0.98 and then 0.44, the program will generate the following output:

```
0.98 is out of range
```

Note that because the out-of-range error occurs in method1(), method2() is not called at all.

If Math.random() in MyClass2 returns 0.98 and then 0.44, the following stack trace would be printed:

```
java.lang.ArithmeticException: 0.98 is out of range
    at MyClass2.method1(MyClass2.java:3)
    at MyClass2.main(MyClass2.java:15)
```

If Math.random() in MyClass2 returns 0.44 and then 0.98, the program will generate the following output:

```
Hello 0.44
0.98 is out of range
```

If Math.random() in MyClass2 returns 0.44 and then 0.98, the following stack trace would be printed:

```
java.lang.ArithmeticException: 0.98 is out of range
    at MyClass2.method2(MyClass2.java:8)
    at MyClass2.main(MyClass2.java:16)
```

The divide-by-zero error in BadDivide occurs in the expression n/d in Method2(). It would generate the following stack trace:

```
java.lang.ArithmeticException: divide by zero
    at BadDivide.method2(BadDivide.java:7)
    at BadDivide.method1(BadDivide.java:3)
    at BadDivide.main(BadDivide.java:13)
```

The following version of BadDivide.method2() will handle the divide-by-zero error itself:

```
public void method2 (int n, int d) {
    try {
        System.out.println(n / d);
    } catch (ArithmeticException e) {
        System.out.println(e.getMessage());
        e.printStackTrace();
        System.exit(0);
    }
}
```

If someValue equals 1000, the code segment will print

```
Entering try block
ERROR: 1000 is too large
```

If someValue equals 50, the code segment will print

```
Entering try block
Exiting try block
```

```
try {
 if (X < 0)
    throw new Exception(
           "ERROR: Negative value in X coordinate");
} catch (Exception e) {
 System.out.println( e.getMessage() );
}
```

1. It depends. This is a computer game, so one way to handle this problem would be to generate a message into a log file and resume the game. If the GUI element is crucial to the game, it's hard to see how it could be successfully handled.
2. It depends. You would have to decide whether it would be more harmful or dangerous to continue production than not.
3. The program could report the security violation to the user and to the system manager and then keep accepting user input.

```
public class FieldIsEmptyException extends Exception {
    public FieldIsEmptyException () {
        super("The input field is empty ");
    }
}
```

```
public int getInt() {
    int num = 0;
    try {
        String data = getText();
        if (data.equals(""))
            throw new FieldIsEmptyException();
        num = Integer.parseInt( getText() );
        if (num > bound)
            throw new IntOutOfRangeException(bound);
    } catch (FieldIsEmptyException e) {
        System.out.println("Error: " + e.getMessage() );
    } catch (NumberFormatException e) {
      System.out.println("Error: You must input an integer.
                                      Please try again.");
    } catch (IntOutOfRangeException e) {
        System.out.println(e.getMessage());
        return 0;
    }
    return num;
}
```

Explain the difference between the following pairs of terms:

Throwing an exception and *catching an exception*.

Try block and *catch block*.

Catch block and *finally block*.

Try block and *finally block*.

Dynamic scope and *static scope*.

Dialog box and *top-level window*.

Checked and *unchecked* exception.

Method stack and *method call*.

Fill in the blanks.

=14pt

---

an exception is Java's way of signaling that some kind of abnormal situation has occurred.

The only place that an exception can be thrown in a Java program is within a

---

.

The block of statements placed within a catch block is generally known as an

---

.

To determine a statement's

---

scope, you have to trace the program's execution.

To determine a statement's

---

scope, you can just read its definition.

When a method is called, a representation of the method call is placed on the

---

.

The root of Java’s exception hierarchy is the

---

class.

A

---

exception must be either caught or declared within the method in which it might be thrown.

An

---

exception can be left up to Java to handle.

=11pt

Compare and contrast the four different ways of handling exceptions within a program.

Suppose you have a program that asks the user to input a string of no more than five letters. Describe the steps you’d need to take in order to design a StringTooLongException to handle cases where the user types in too many characters.

Exceptions require more computational overhead than normal processing. Explain.

Suppose the following ExerciseExample program is currently executing the if statement in method2():

```
public class ExerciseExample {
  public void method1(int M) {
    try {
      System.out.println("Entering try block");
      method2( M );
      System.out.println("Exiting try block");
    } catch (Exception e) {
      System.out.println("ERROR: " + e.getMessage());
    }
  } // method1()

  public void method2(int M) {
    if (M > 100)
      throw new ArithmeticException(M + " is too large");
  }

  public static void main(String argv[]) {
    ExerciseExample ex = new ExerciseExample();
    ex.method1(500);
  }
} // ExerciseExample
```

Draw a picture of the method call stack that represents this situation.

Repeat the previous exercise for the situation where the program is currently executing the second println() statement in method1().

Draw a hierarchy chart that represents the static scoping relationships among the elements of the ExerciseExample program.

What would be printed by the ExerciseExample program when it is run?

What would be printed by the ExerciseExample program, if the statement in its main method were changed to ex.method1(5)?

Consider again the ExerciseExample program. If the exception thrown were Exception rather than ArithmeticException, explain why we would get the following error message: java.lang.Exception must be caught, or it must be declared….

Write a try/catch block that throws an Exception if the value of variable X is less than zero. The exception should be an instance of Exception and, when it is caught, the message returned by getMessage() should be “ERROR: Negative value in X coordinate.”

Look at the IntFieldTester program (Fig. [fig-intbound]) and the IntField class definition (Fig. [fig-intfield2]). Suppose the user inputs a value that's greater than 100. Show what the method call stack would look like when the IntField.getInt() method is executing the num > bound expression.

As a continuation of the previous exercise, show what the program's output would be if the user input a value greater than 100.

As a continuation of the previous exercise, modify the IntOutOfRangeException handler so that it prints the message call stack. Then show what it would print.

Define a subclass of RuntimeException named InvalidPasswordException, which contains two constructors. The first constructor takes no parameters and an exception thrown with this constructor should return "ERROR: invalid password" when its getMessage() is invoked. The second constructor takes a single String parameter. Exceptions thrown with this constructor should return the constructor's argument when getMessage() is invoked.

Extend the IntField class so that it will constrain the integer JTextField to an int between both a lower and upper bound. In other words, it should throw an exception if the user types in a value lower than the lower bound or greater than the upper bound.

Design Issue: One of the preconditions for the InsertionSort() method (Fig. 9.13) is that its array parameter not be null. Of course, this precondition would fail if the array were passed a null array reference. In that case, Java would throw a NullPointerException and terminate the program. Is this an appropriate way to handle that exception?

With respect to the previous exercise, suppose you decide that it is more appropriate to handle the NullPointerException by presenting an error dialog. Modify the method to accommodate this behavior.

Design Issue: Another possible way to design the sequentialSearch() method (Fig. 9.16) would be to have it throw an exception when its key is not found in the array. Is this a good design? Explain.

This page titled 9.7: From the Java Library- javax.swing.JOptionPane is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **10.7: From the Java Library- javax.swing.JOptionPane** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

CHAPTER OVERVIEW

10: Recursive Problem Solving

Learning Objectives

After studying this chapter, you will

- Understand the concept of recursion.
- Know how to use recursive programming techniques.
- Have a better understanding of recursion as a problem-solving technique.

Introduction

The pattern in Figure 12.1 is known as the Sierpinski gasket. Its overall shape is that of an equilateral triangle. But notice how inside the outer triangle there are three smaller triangles that are similar to the overall pattern. And inside each of those are three even smaller triangles, and so on. The Sierpinski gasket is known as a *fractal* because when you divide it up, you end up with a smaller version of the overall pattern. The overall gasket pattern is repeated over and over, at smaller and smaller scales, throughout the figure.

How would you draw this pattern? If you try to use some kind of nested loop structure, you'll find that it is very challenging. It can be done using loops, but it isn't easy. On the other hand, if you use an approach known as *recursion*, this problem is much easier to solve. It's a little bit like the representation issue we discussed in Chapter 5. Your ability to solve a problem often depends on how you represent the problem. Recursion gives you another way to approach problems that involve repetition, such as the problem of drawing the Sierpinski gasket.

The main goal of this chapter is to introduce recursion as both a problem-solving technique and as alternative to loops (which we discussed in Chapter 6) for implementing repetition. We begin with the notion of a *recursive definition*, a concept used widely in mathematics and computer science. We then introduce the idea of a *recursive method*, which is the way recursion is implemented in a program.

Recursion is a topic that is taken up in considerable detail in upper-level computer science courses, so our goal here is mainly to introduce the concept and give you some idea of its power as a problem-solving approach. To illustrate recursion, we use a number of simple examples throughout the chapter. One risk in using simple examples, though, is that you might be tempted to think that recursion is only good for "toy problems." Nothing could be further from the truth. Recursion is often used for some of the most difficult algorithms. Some of the exercises at the end of the chapter are examples of more challenging problems.

10.1: Introduction
10.2: Recursive Definition
10.3: Recursive String Methods
10.4: Recursive Array Processing
10.5: Example- Drawing (Recursive) Fractals
10.6: Object-Oriented Design- Tail Recursion
10.7: Object-Oriented Design- Recursion or Iteration?
10.8: Special Topic- Exploring the Mandelbrot Set
10.9: From the Java Library- javax.swing.JComboBox

This page titled 10: Recursive Problem Solving is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

10.1: Introduction

10.1: Introduction is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

10.2: Recursive Definition

Recursion as Repetition

A **recursive method** is a method that calls itself. An **iterative method** is a method that uses a loop to repeat an action. In one sense, *recursion* is an alternative to the iterative (looping) control structures we studied in Chapter 6. In this sense, recursion is just another way to repeat an action.

For example, consider the following iterative method for saying "Hello" *N* times:

```
public void hello(int N)  {
    for (int k = 0; k < N; k++)
        System.out.println("Hello");
} // hello()
```

A recursive version of this method would be defined as follows:

```
public void hello(int N)  {
    if (N > 0) \verb!{!
        System.out.println("Hello");
        hello(N - 1);               // Recursive call
} // hello()
```

This method is recursive because it calls itself when *N* is greater than 0. However, note that when it calls itself, it passes $N-1$ as the value for its parameter. If this method is initially called with *N* equal to 5, the following is a trace of what happens. The indentations indicate each time the method calls itself:

```
hello(5)
    Print "Hello"
    hello(4)
        Print "Hello"
        hello(3)
            Print "Hello"
            hello(2)
                Print "Hello"
                hello(1)
                    Print "Hello"
                    hello(0)
```

Thus, "Hello" will be printed five times, just as it would be in the iterative version of this method.

So, in one sense, recursion is just an alternative to iteration. In fact, there are some programming languages, such as the original versions of LISP and PROLOG, that do not have loop structures. In these languages, *all* repetition is done by recursion. In contrast, if a language contains loop structures, it can do without recursion. Anything that can be done iteratively can be done recursively, and vice versa.

Moreover, it is much less efficient to call a method five times than to repeat a for loop five times. Method calls take up more memory than loops and involve more **computational overhead**—for such tasks as passing parameters, allocating storage for the method's local variables, and returning the method's results. Therefore, because of its reliance on repeated method calls, recursion is often less efficient than iteration as a way to code a particular algorithm.

What would be printed if we call the following method with the expression mystery(0)?

```
public void mystery(int N) {
    System.out.print(N + " ");
    if (N <= 5)
        mystery(N + 1);
} // mystery()
```

What about mystery(100)?

What would be printed if we call the following method with the expression mystery(5)?

```
public void mystery(int N) {
    System.out.print(N + " ");
    if (N <= 5)
        mystery(N - 1);
} // mystery()
```

Recursion as a Problem-Solving Approach

Given that recursion is not really necessary—if a programming language has loops—and is not more efficient than loops, why is it so important? The answer is that, in a broader sense, recursion is an effective approach to problem solving. It is a way of viewing a problem. It is mostly in this sense that we want to study recursion.

Recursion is based on two key problem-solving concepts: *divide and conquer* and **self-similarity**. In recursive problem solving we use the divide-and-conquer strategy repeatedly to break a big problem into a sequence of smaller and smaller problems until we arrive at a problem that is practically trivial to solve.

What allows us to create this series of subproblems is that each subproblem is similar to the original problem—that is, each subproblem is just a smaller version of the original problem. Look again at the task of saying "Hello" N times. Solving this task involves solving the similar task of saying "Hello" $N-1$ times, which can be divided into the similar task of saying "Hello" $N-2$ times. And so on.

The ability to see a problem as being composed of smaller, self-similar problems is at the heart of the recursive approach. And although you might not have thought about this before, a surprising number of programming problems have this self-similarity characteristic. Let's illustrate these ideas with some simple examples.

Recursive Definition

One place you might have already seen recursion is in mathematics. A **recursive definition** consists of two parts: a recursive part in which the *n*th value is defined in terms of the $(n-1)$st value, and a nonrecursive, *boundary* or **base case**, which defines a limiting condition.

Factorial: N!

For example, consider the problem of calculating the factorial of *n*—that is, *n*! for $n \geq 0$. As you may recall, *n*! is calculated as follows:

```
n! = n * (n-1) * (n-2) * ... * 1, for n > 0
```

In addition, 0! is defined as 1. Let's now look at some examples for different values of *n*:

```
4! = 4 * 3 * 2 * 1 = 24
3! = 3 * 2 * 1 = 6
2! = 2 * 1 = 2
1! = 1
0! = 1
```

As these examples suggest, *n*! can always be calculated in terms of $(n-1)$! This relationship might be clearer if we rewrite the previous calculations as follows:

```
4! = 4 * 3 * 2 * 1 = 4 * 3! = 24
3! = 3 * 2 * 1     = 3 * 2! = 6
2! = 2 * 1         = 2 * 1! = 2
1!                 = 1 * 0! = 1
0!                 = 1
```

The only case in which we can't calculate *n*! in terms of $(n-1)$! is when *n* is 0. Otherwise, in each case we see that

```
n! = n * (n-1)!
```

This leads to the following recursive definition:

```
n! = 1          if n = 0    // Boundary (or base) case
n! = n * (n-1)! if n > 0    // Recursive case
```

Note that if we had omitted the base case, the recursion would have continued to (-1)! and (-2)! and so on.

The recursive case uses divide and conquer to break the problem into a smaller problem, but the smaller problem is just a smaller version of the original problem. This combination of self-similarity and divide and conquer is what characterizes recursion. The base case is used to stop or limit the recursion.

Drawing a Nested Pattern

As another example, consider the problem of drawing the nested boxes pattern in Figure 12.2. The self-similarity occurs in the fact that for this pattern, its parts resemble the whole. The basic shape involved is a square, which is repeated over and over at an ever-smaller scale. A recursive definition for this pattern would be

```
Base case:      if side < 5 do nothing
Recursive case: if side >= 5
                  draw a square
                  decrease the side and draw a smaller
                            pattern inside the square
```

This definition uses the length of the square's side to help define the pattern. If the length of the side is greater than or equal to 5, draw a square with dimensions $side \times side$. Then decrease the length of the side and draw a smaller version of the pattern inside that square. In this case, the *side* variable will decrease at each level of the drawing. When the length of the side becomes less than 5, the recursion stops. Thus, the length of the side serves as the limit or bound for this algorithm.

You should note that the length of the side functions here like a parameter in a method definition: It provides essential information for the definition, just as a method parameter provides essential data to the method. Indeed, this is exactly the role that parameters play in recursive methods. They provide essential information that determines the method's behavior.

Figure 12.3 illustrates how we would apply the definition. Suppose the side starts out at 20 and decreases by 5 at each level of recursion. Note that as you move from top to bottom across the four patterns, each pattern contains the one below it. A nestedBoxes(20) can be drawn by first drawing a 20×20 square and then drawing a nestedBoxes(15) pattern inside it. Similarly, a nestedBoxes(15) can be drawn by first drawing a 15×15 square and then drawing a nestedBoxes(10) pattern inside it. And so on.

These examples illustrate the power of recursion as a problem-solving technique for situations that involve repetition. Like the iterative (looping) control structures we studied in Chapter 6, recursion is used to implement repetition within a bound. For recursive algorithms, the bound is defined by the base case, whereas for loops, the bound is defined by the loop's entry condition. In either case, repetition stops when the bound is reached.

You can calculate 2^n by multiplying 2 by itself *n* times. For example, 2^3 is $2 \times 2 \times 2$. Note also that $2^0 = 1$. Given these facts, write a recursive definition for 2^n, for $n \geq 0$.

Generalize your solution to the previous exercise by giving a recursive definition for x^n, where x and n are both integers ≥ 0.

Is the recursive definition given earlier for the nested boxes equivalent to the following recursive definition? Explain.

```
Draw a square.          // in every case
If side > 5
    draw a smaller nested boxes inside the square
```

In this case, the base case $(side <= 5)$ is implicit.

Write a recursive definition for the recursive pattern shown in Figure 12.4.

This page titled 10.2: Recursive Definition is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **12.2: Recursive Definition** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

10.3: Recursive String Methods

Remember that a *recursive method* is a method that calls itself. Like recursive definitions, recursive methods are designed around the divide-and-conquer and self-similarity principles. Defining a recursive method involves a similar analysis to the one we used in designing recursive definitions. We identify a self-similar subproblem of the original problem plus one or more limiting cases.

The idea of a method calling itself seems a bit strange at first. It's perhaps best understood in terms of a clone or a copy. When a method calls itself, it really calls a copy of itself, one that has a slightly different internal state. Usually the difference in state is the result of a difference in the invoked method's parameters.

Printing a String

To illustrate the concept of a recursive method, let's define a recursive method for printing a string. This is not intended to be a practical method—we already have the println() method for printing strings. But pretend for a moment that you only have a version of println() that works for characters, and your task is to write a version that can be used to print an entire string of characters.

A little terminology will help us describe the algorithm. Let's call the first letter of a string the *head* of the string, and let's refer to all the remaining letters in the string as the *tail* of the string. Then the problem of printing a string can be solved using a **head-and-tail algorithm**, which consists of two parts: printing the head of the string and recursively printing its tail. The limiting case here is when a string has no characters in it. It's trivial to print the empty string—there is nothing to print! This leads to the method definition shown in Figure [fig-printstring].

```
 /**
  * printString() prints each character of the string s
  * Pre: s is initialized (non-null)
  * Post: none
  */
public void printString(String s) {
    if (s.length() == 0)
        return;                  // Base case: do nothing
    else {        // Recursive case:
        System.out.print(s.charAt(0)); // Print head
        printString(s.substring(1));   // Print tail
    }
} // printString()
```

The base case here provides a limit and bounds the recursion when the length of *s* is 0—that is, when the string is empty. The recursive case solves the problem of printing *s* by solving the smaller, self-similar problem of printing a substring of *s*. Note that the recursive case makes progress toward the limit. On each recursion, the tail will get smaller and smaller until it becomes the empty string.

Let's now revisit the notion of a method calling itself. Obviously this is what happens in the recursive case, but what does it mean —what actions does this lead to in the program? Each recursive call to a method is really a call to a *copy* of that method, and each copy has a slightly different internal state. We can define printString()'s internal state completely in terms of its recursion parameter, *s*, which is the string that's being printed. A **recursion parameter** is a parameter whose value is used to control the progress of the recursion. In this case, if *s* differs in each copy, then so will s.substring(1) and s.charAt(0).

Figure [fig-traceprint] illustrates the sequence of recursive method calls and the output that results when

printString("hello") is invoked. Each box represents a separate instance of the printString() method, with its own internal state. In this illustration, its state is represented by its parameter, s. Because each instance has a different parameter, behaviors differ slightly. Each box shows the character that will be printed by that instance (s.charAt(0)), and the string that will be passed on to the next instance (s.substring(1)).

The arrows represent the method calls and returns. Note that the first return executed is the one in the base case. Each instance of the method must wait for the instance it called to return before it can return. That's why the instances "pile up" in a cascade-like structure. The arrowless lines trace the order in which the output is produced.

Each instance of printString() is similar to the next in that each will print one character and then pass on a substring, but each performs its duties on a different string. Note how the string, the recursion parameter in this case, gets smaller in each instance of printString(). This represents progress toward the method's base case s.length() == 0. When the empty string is passed as an argument, the recursion will stop. If the method does not make progress toward its bound in this way, the result will be an infinite recursion.

Note also the order in which things are done in this method. First s.charAt(0) is printed, and then s.substring(1) is passed to printString() in the recursion. This is a typical structure for a head-and-tail algorithm. What makes this work is that the tail is a smaller, self-similar version of the original structure.

What would be printed by the following version of the printString2() method, if it is called with printString2("hello")?

```
public void printString2(String s)  {
 if (s.length() == 1)
   System.out.print(s.charAt(0));         // Base case:
 else {
                           // Print last char
   System.out.print(s.charAt(s.length() - 1));
                          // Print rest of string
   printString2(s.substring(0, s.length() - 1));
 }
} // printString2()
```

Printing the String Backward

What do you suppose would happen if we reversed the order of the statements in the printString() method? That is, what if the recursive call came before s.charAt(0) is printed, as in the following method:

```
 /**
  * printReverse() prints each character s in reverse order
  * Pre: s is initialized (non-null)
  * Post: none
  */
public void printReverse(String s) {
  if (s.length() > 0) {      // Recursive case:
     printReverse(s.substring(1));  // Print tail
     System.out.print(s.charAt(0)); // Print first char
  }
} // printReverse()
```

As its name suggests, this method will print the string in reverse order. The trace in Figure [fig-tracerevprint] shows how

this works. Before printReverse("hello") can print *h*, it calls printReverse("ello") and must wait for that call to complete its execution and return. But printReverse("ello") calls printReverse("llo") and so must wait for that call to complete its execution and return.

This process continues until printReverse("") is called. While the base case is executing, the other five instances of printReverse() must each wait for the instance that they called to complete its execution. It is only after the base case returns, that printReverse("o") can print its first character and return. So the letter *o* will be printed first. After printReverse("o") has returned, then printReverse("lo") can print its first character. So the letter *l* will be printed next, and so on, until the original call to printReverse("hello") is completed and returns. Thus, the string will be printed in reverse order.

Note that the method call and return structure in this example follows a **last-in–first-out (LIFO) protocol**. That is, the last method called is always the first method to return. This is the protocol used by all method calls, recursive or otherwise.

For example, compare the order in which things happen in Figure [fig-tracerevprint] with the method stack trace in Figure 10.12. The only real difference between the two figures is that here the method stack is represented as growing downward, whereas in Figure 10.12 it grows upward. As each method call is made, a representation of the method call is placed on the *method call stack*. When a method returns, its block is removed from the top of the stack. The only difference between recursive and nonrecursive method calls is that recursive methods call instances of the same method definition. Of course, as we've seen, the instances are all slightly different from each other.

Write a recursive method called countDown() that takes a single int parameter, $N \geq 0$, and prints a countdown, such as "5, 4, 3, 2, 1, blastoff." In this case, the method would be called with countDown(5).

Revise the method in the previous exercise so that when it's called with countDown(10), it will print "10 8 6 4 2 blastoff"; if it's called with countDown(9), it prints "9 7 5 3 1 blastoff."

Counting Characters in a String

Suppose you're writing an encryption program and you need to count the frequencies of the letters of the alphabet. Let's write a recursive method for this task.

This method will have two parameters: a String to store the string that will be processed and a char to store the target character—the one we want to count. The method should return an int, representing the number of occurrences of the target character in the string:

```
// Goal: count the occurrences of ch in s
public int countChar(String s, char ch) {
    ...
}
```

Here again our analysis must identify a recursive step that breaks the problem into smaller, self-similar versions of itself, plus a base case or limiting case that defines the end of the recursive process. Because the empty string will contain no target characters, we can use it as our base case. So, if it is passed the empty string, countChar() should just return 0 as its result.

For the recursive case, we can divide the string into its head and tail. If the head is the target character, then the number of occurrences in the string is (1 + the number of occurrences in its tail). If the head of the string is not the target character, then the number of occurrences is (0 + the number of occurrences in its tail). Of course, we'll use recursion to calculate the number of occurrences in the tail.

This analysis leads to the recursive method shown in Figure [fig-countch]. Note that for both recursive cases the

```
/**
 * Pre:  s is a non-null String, ch is any character
 * Post: countChar() == the number of occurrences of ch in str
 */
public int countChar(String s, char ch) {
 if (s.length() == 0)    // Base case: empty string
   return 0;
 else if (s.charAt(0) == ch)  // Recursive case 1
   return 1 + countChar(s.substring(1), ch); // Head = ch
 else                         // Recursive case 2
   return 0 + countChar(s.substring(1), ch); // Head != ch
} // countChar()
```

same recursive call is used. In both cases we pass the tail of the original string, plus the target character. Note also how the return statement is evaluated:

```
return 1 + countChar(s.substring(1),ch); // Head = ch
```

Before the method can return a value, it must receive the result of calling countChar(s.substring(1),ch) and add it to 1. Only then can a result be returned to the calling method. This leads to the following evaluation sequence for countChar("dad",'d'):

```
countChar("dad",'d');
1 + countChar("ad",'d');
1 + 0 + countChar("d",'d');
1 + 0 + 1 + countChar("",'d');
1 + 0 + 1 + 0 = 2             // Final result
```

In this way, the final result of calling countChar("dad",'d') is built up recursively by adding together the partial results from each separate instance of countChar(). The evaluation process is shown graphically in Figure 12.9.

Here's a numerical problem. Write a recursive method to compute the sum of 1 to *N*, given *N* as a parameter.

Translating a String

A widely used string-processing task is to convert one string into another string by replacing one character with a substitute throughout the string. For example, suppose we want to convert a Unix path name, which uses the forward slash "/" to separate one part of the path from another, into a Windows path name, which uses the backslash character "\" as a separator. For example, we want a method that can translate the following two strings into one another:

```
/unix_system/myfolder/java
\Windows_system\myfolder\java
```

Thus, we want a method that takes three parameters: a String, on which the conversion will be performed, and two char variables, the first being the original character in the string and the second being its substitute. The precondition for this method is simply that each of these three parameters has been properly initialized with a value. The postcondition is that all occurrences of the first character have been replaced by the second character.

As in our previous string-processing methods, the limiting case in this problem is the empty string, and the recursive case will divide the string into its head and its tail. If the head is the character we want to replace, we concatenate its substitute with the result we obtain by recursively converting its tail.

This analysis leads to the definition shown in Figure 12.10. This method has more or less the same head and tail structure as the preceding example. The difference is that here the operation we perform on the head of the string is concatenation rather than addition.

-3pc

```
 /**
  * Pre:  str, ch1, ch2 have been initialized
  * Post: the result contains a ch2 everywhere that ch1
  *       had occurred in str
  */
public static String convert(String str, char ch1, char ch2) {
 if (str.length() == 0)         // Base case: empty string
   return str;
 else if (str.charAt(0) == ch1) // Recursive 1: ch1 at head
                                // Replace ch1 with ch2
   return ch2 + convert(str.substring(1), ch1, ch2);
 else                         // Recursive 2: ch1 not at head
   return str.charAt(0) + convert(str.substring(1), ch1, ch2);
} // convert()
```

The base case is still the case in which str is the empty string. The first recursive case occurs when the character being replaced is the head of str. In that case, its substitute (ch2) is concatenated with the result of converting the rest of the string and returned as the result. The second recursive case occurs when the head of the string is *not* the character being replaced. In this case, the head of the string is simply concatenated with the result of converting the rest of the string. Figure [fig-traceconvert] shows an example of its execution.

Write a recursive method that changes each blank in a string into two consecutive blanks, leaving the rest of the string

Printing all Possible Outcomes when Tossing N Coins

Suppose that a student who is studying probability wishes to have a Java program that, for any positive integer N, will print out a list of all possible outcomes when N coins are tossed. For purposes of analyzing the problem, it is assumed that the coins are numbered 1 through N and that they are tossed one at a time. An outcome will be represented by a string of Hs and Ts corresponding to heads and tails. Thus, if $N = 2$, the string HT represents a head on the first coin and a tail on the second coin. What we need is a method which, when given the number of coins, will print out all strings of this type. In case of two coins the output should be:

```
HH
HT
TH
TT
```

Let's devise a strategy, given any positive integer N, for printing the strings that correspond to all possible outcomes when tossing N coins. Clearly, for $N = 1$, the method needs to print an H on one line and a T on the next line. For an arbitrary number of coins N, one way to generate all outcomes is to think of two kinds of strings—those that start with an H and those that start with a T. The strings that start with H can be generated by inserting an H in front of each of the outcomes that occur when $N - 1$ coins are tossed. The strings beginning with T can be generated in a similar manner. Thus, using the outcomes for two coins above, we know that the outcomes for three coins for which the first coin is H are:

```
HHH
HHT
HTH
HTT
```

Using an argument similar to the one above, we can generalize this to a description of the recursive case for an algorithm. We want an algorithm that generates all those outcomes for N coins where we are given a string STR representing one particular outcome for all but the last K coins where $0 < K <= N$. To print out all such outcomes, just print all outcomes with a fixed outcome corresponding to STR + "H" for all but the last $K - 1$ coins and then print all outcomes with the fixed outcome STR + "T" for all but the last $K - 1$ coins. The base case is the special case $K = 1$ when you just need STR + "H" on one line and STR + "T" on the next. If you start the algorithm with STR = "" and $K = N$, this algorithm will print out all the outcomes for tossing N coins.

To translate this into Java code we can create a class called CoinToss which has a single static method called printOutcomes(String str,int N). The above recursive description easily translates into code for the method in Figure [fig-cointoss].

```
 /**
 * printOutcomes(str, N) prints out all possible outcomes
 * beginning with str when N more coins are tossed.
 * Pre: str is a string of Hs and Ts.
 * Pre: N is a positive integer.
 * Post: none
 */
 public static void printOutcomes(String str,int N){
     if (N == 1){ // The base case
         System.out.println(str + "H");
         System.out.println(str + "T");
     } else {  // The recursive case
         printOutcomes(str + "H", N - 1);
         printOutcomes(str + "T", N - 1);
     } //else
 }// printOutcomes()
```

To print out all outcomes when tossing, say, seven coins, just make the method call CoinToss.printOutcomes("",7). This particular call would generate the desired output:

```
HHHHHHH
HHHHHHT
.......
TTTTTTH
TTTTTTT
```

To better understand how the recursive method definition generates its output, it might be helpful to trace the order of recursive calls and output to System.out that occurs when executing printOutcomes("",3) as shown in Figure 12.13.

Notice that the recursive case in the method implementation makes two calls to itself and as a result it is not so clear how this method would be written using a loop instead of recursion. This example is more typical of the type of problem for which a recursive method is shorter and clearer than a method that solves the same problem without using recursion.

Modify the above printOutcomes() method so that it will print out all possible outcomes when a chess player plays a series of N games. The outcome of each game is to be represented by a W, L, or D corresponding to the player winning, losing, or drawing the game.

This page titled 10.3: Recursive String Methods is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **12.3: Recursive String Methods** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

10.4: Recursive Array Processing

Like strings, arrays also have a recursive structure. Just as each substring of a string is similar to the string as a whole, each portion of an array is similar to the array as a whole. Similarly, just as a string can be divided into a head and a tail, an array can be divided into its *head*, the first element, and its *tail*, the rest of its elements (Fig. 12.14). Because the tail of an array is itself an array, it satisfies the self-similarity principle. Therefore, arrays have all the appropriate characteristics that make them excellent candidates for recursive processing.

Recursive Sequential Search

Let's start by developing a recursive version of the sequential search algorithm that we discussed in Chapter 9. Recall that the sequential search method takes two parameters: the array being searched and the *key*, or target value, being searched for. If the key is found in the array, the method returns its index. If the key is not found, the method returns -1, thereby indicating that the key was not contained in the array. The iterative version of this method has the following general form:

```
 /**
  * Performs a sequential search of an integer array
  * @param arr is the array of integers
  * @param key is the element being searched for
  * @return the key's index is returned if the key is
  *  found otherwise -1 is returned
  * Pre:  arr is not null
  * Post: either -1 or the key's index is returned
  */
public int sequentialSearch(int arr[], int key) {
    return -1;           // failure if this is reached
}
```

If we divide the array into its head and tail, then one way to describe a recursive search algorithm is as follows:

```
If the array is empty, return -1
If the array's head matches the key, return its index
If the array's head doesn't match the key,
 return the result of searching the tail of the array
```

This algorithm clearly resembles the approach we used in recursive string processing: Perform some operation on the head of the array and recurse on the tail of the array.

The challenge in developing this algorithm is not so much knowing what to do but knowing how to represent concepts like the head and the tail of the array. For strings, we had methods such as s.charAt(0) to represent the head of the string and s.substring(1) to represent the string's tail. For an array named arr, the expression arr[0] represents the head of the array. Unfortunately, we have no method comparable to the substring() method for strings that lets us represent the tail of the array.

To help us out of this dilemma, we can use an integer parameter to represent the head of the array. Let's have the int parameter, *head*, represent the current head of the array (Fig. [fig-headtail2]). Then $head + 1$ represents the start of the tail, and arr.length-1 represents the end of the tail. Our method will always be passed a reference to the whole array, but it will restrict the search to the portion of the array starting at *head*. If we let *head* vary from 0 to arr.length on each recursive call, the method will recurse through the array in head/tail fashion, searching for the key. The method will stop when head = arr.length.

```
  /**
   * search(arr,head,key)---Recursively search arr for key
   *  starting at head
   * Pre:  arr != null and 0 <= head <= arr.length
   * Post: if arr[k] == key for k, 0 <= k < arr.length,
   *       return k else return -1
   */
 private int search(int arr[], int head, int key)  {
  if (head == arr.length) // Base: empty list - failure
    return -1;
  else if (arr[head] == key) // Base: key found---success
    return head;
  else               // Recursive: search the tail
    return search(arr, head + 1, key);
 }
```

This leads to the definition for recursive search shown in Figure [fig-rsearch]. Note that the recursive search method takes three parameters: the array to be searched, arr, the key being sought, and an integer head that gives the starting location for the search. The algorithm is bounded when head = arr.length. In effect, this is like saying the recursion should stop when we have reached a tail that contains 0 elements. This underscores the point we made earlier about the importance of parameters in designing recursive methods. Here the *head* parameter serves as the *recursion parameter*. It controls the progress of the recursion.

Note also that for the search algorithm we need two base cases. One represents the successful case, where the key is found in the array. The other represents the unsuccessful case, which comes about after we have looked at every possible head in the array and not found the key. This case will arise through exhaustion—that is, when we have exhausted all possible locations for the key.

Information Hiding

Note that in order to use the search() method, you would have to know that you must supply a value of 0 as the argument for the *head* parameter. This is not only awkward but also impractical. After all, if we want to search an array, we just want to pass two arguments, the array and the key we're searching for. It's unreasonable to expect users of a method to know that they also have to pass 0 as the head in order to get the recursion started. This design is also prone to error, because it's quite easy for a mistake to be made when the method is called.

For this reason, it is customary to provide a nonrecursive interface to the recursive method. The interface hides the fact that a recursive algorithm is being used, but this is exactly the kind of implementation detail that should be hidden from the user. This is an example of the principle of information hiding that we introduced in Chapter 0. A more appropriate design would make the recursive method a private method that's called by the public method, as shown Figure [fig-searcher] and implemented in the Searcher class (Fig. [fig-searcherclass]).

-3pc

```
public class Searcher {
  /**
   * search(arr,key) -- searches arr for key.
   * Pre:  arr != null and 0 <= head <= arr.length
   * Post: if arr[k] == key for k,  0 <= k < arr.length,
   *       return k,  else return -1
   */
  public int search(int arr[], int key) {
    return search(arr, 0, key); // Call recursive search
  }
  /**
   * search(arr, head, key) -- Recursively search arr for key
   *  starting at head
   * Pre:  arr != null and 0 <= head <= arr.length
   * Post: if arr[k] == key for k,  0 <= k < arr.length, return k
   *       else return -1
   */
  private int search(int arr[], int head, int key)  {
    if (head == arr.length)   // Base case: empty list - failure
      return -1;
    else if (arr[head] == key)// Base case: key found -- success
      return head;
    else                      // Recursive case: search the tail
        return search(arr, head + 1, key);
  } // search()
  public static void main(String args[]) {
    int numbers[] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18};
    Searcher searcher = new Searcher();
    for (int k = 0; k <= 20; k++) {
        int result = searcher.search(numbers, k);
        if (result != -1)
          System.out.println(k + " found at " + result);
        else
          System.out.println(k + " is not in the array ");
    } // for
  } // main()
} // Searcher
```

Write a main() method for the Searcher class to conduct the following test of search(). Create an int array of ten elements, initialize its elements to the even numbers from 0 to 18, and then use a for loop to search the array for each of the numbers from 0 to 20.

Recursive Selection Sort

Next we want you to think back to Chapter 9, where we introduced the selection sort algorithm. To review the concept, suppose you have a deck of 52 cards. Lay them out on a table, face up, one card next to the other. Starting at the last card, look through the deck, from last to first, find the largest card and exchange it with the last card. Then go through the deck again starting at the next to the last card, find the next largest card, and exchange it with the next to the last card. Go to the next card, and so on. If you repeat this process 51 times, the deck will be completely sorted.

Let's design a recursive version of this algorithm. The algorithm we just described is like a head-and-tail algorithm in reverse, where the last card in the deck is like the head, and the cards before it are like the tail. After each pass or recursion, the last card will be in its proper location, and the cards before it will represent the unsorted portion of the deck. If we use parameter to represent *last*, then it will be moved one card to the left at each level of the recursion.

Figure [fig-selectsort] illustrates this process for an array of integers. The base case is reached when the *last* parameter is pointing to the first element in the array. An array with one element in it is already sorted. It needs no rearranging. The recursive case involves searching an ever-smaller portion of the array. This is represented in our design by moving *last* down one element to the left.

Figure [fig-selectionsort] provides a partial implementation of selection sort for an array of int. In this definition, the array is one parameter. The second parameter, int last, defines that portion of the array, from right to left, that is yet to be sorted. On the first call to this method, *last* will be arr.length – 1. On the second, it will be arr.length – 2, and so on. When *last* gets to be 0, the array will be sorted. Thus, in terms of the card deck analogy, *last* represents the last card in the unsorted portion of the deck.

```
  /**
   * selectionSort(arr,last)--Recursively sort arr starting
   *   at last element
   * Pre:  arr != null and 0 <= last < arr.length
   * Post: arr will be arranged so that arr[j] <= arr[k],
   *   for any j < k
   */
 private void selectionSort(int arr[], int last) {
  if (last > 0) {
    int maxLoc = findMax (arr, last);// Find the largest
    swap(arr, last, maxLoc);         // Swap it with last
    selectionSort(arr, last - 1);    // Move down the array
  }
 } // selectionSort()
```

Note how simply the selectionSort() method can be coded. Of course, this is because we have used separate methods to handle the tasks of finding the largest element and swapping the last element and the largest. This not only makes sense in terms of the divide-and-conquer principle, but we also already defined a swap() method in Chapter 9. So here is a good example of reusing code:

```
  /**
   * swap(arr, el1 el2) swaps el1 and el2 in the array, arr
   * Pre: arr is non null, 0 <= el1 < arr.length,
                           0 <= el2 < arr.length
   * Post: the locations of el1 and el2 are swapped in arr
   */
 private void swap(int arr[], int el1, int el2)  {
  int temp = arr[el1]; // Assign the first element to temp
  arr[el1] = arr[el2]; // Overwrite first with second
  arr[el2] = temp;     // Overwrite second with first(temp)
 } // swap()
```

The definition of the findMax() method is left as a self-study exercise.

As in the case of the search() method, we need to provide a public interface to the recursive selectionSort() method. We want to enable the user to sort an array just by calling sort(arr), where arr is the name of the array to be sorted. Define the sort() method.

Define an iterative version of the findMax(arr,N) method that is used in selectionSort(). Its goal is to return the location (index) of the largest integer between arr[0] and arr[N].

This page titled 10.4: Recursive Array Processing is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **12.4: Recursive Array Processing** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

10.5: Example- Drawing (Recursive) Fractals

[pg-sec-fractals] A **fractal** is a geometric shape that exhibits a recursive structure. When it is divided into parts, each part is a smaller version of the whole. Fractal patterns occur in many situations and places. For example, if you look at a graph of the Dow Jones Industrial Average (DJIA) over the past year, the graph for each day is similar to the graph of each month, which is similar to the graph of each year, and so on. Each part is a reduced-scale version of the whole. Fractals also occur throughout nature. If you look at a coastline from an airplane, the shape of each part of the coastline, no matter how small the scale, resembles the shape of the whole coastline. If you look at a tree, each branch of the tree is similar in shape to the whole tree.

So, fractal patterns are all around us. Because of their self-similarity and divisibility, fractals are well-suited for recursive programming. Drawing recursive patterns is also an excellent way to illustrate how to use parameters to create generality in method design. In this section, we will develop two simple patterns and incorporate them into a GUI.

Nested Squares

Earlier in this chapter, we developed a recursive definition for drawing a nested squares pattern (Fig. 12.2). Now let's develop a recursive method that actually draws the pattern. For this pattern, the base case is the drawing of the square. The recursive case, if more divisions are desired, is the drawing of smaller patterns within the square:

```
Draw a square.
If more divisions are desired
    draw a smaller version of pattern within square.
```

An important consideration for this algorithm is to specify precisely what we mean by "if more divisions are desired." In other words, how exactly do we control the recursion? In our earlier definition of the pattern, we used the length of the side to control the algorithm. When $side \geq 5$, we recursed.

Another more general way to do this is to describe the fractal structure in terms of its *levels*. For nested squares, the level-zero pattern would be just the basic square shape (Fig. 12.21). A level-one pattern would be the basic square shape plus an inner square, and so on. The higher the level, the more subdividing we do. Therefore, one way to control the recursion is to use a *level* parameter as the *recursion parameter*—as the parameter that controls the recursion:

```
Draw a square.
If the level is greater than 0,
    draw a smaller version of pattern within square.
```

What other parameters will we need for this method? If we're going to draw a rectangle, we'll need parameters for its *x*- and *y*-coordinates. We'll also need a parameter for the length of sides of the square. Another issue we need to decide is how much the length of the sides should change at each level. Should length change by a fixed amount, by a fixed ratio, or by some other factor? In order to allow this kind of flexibility, let's use another parameter for this value.

These design considerations suggest the method shown in Figure [fig-drawboxes]. Note that we must also provide a Graphics parameter so the method can use the drawRect() method to draw the square. As we decided, the level parameter controls the recursion. Note that its value is decreased by 1 in the recursive call. This will ensure that level will eventually reach 0, and recursion will stop.

```
/**
 * drawBoxes()---recursively draws pattern of nested
 *  squares with top left corner of outer square at
 *  (locX, locY) and dimensions of length side.
 * level (>= 0) is the recursion parameter (base: = 0)
 * delta is used to adjust the length of the side.
 */
private void drawBoxes(Graphics g, int level,
           int locX, int locY, int side, int delta) {
  g.drawRect(locX, locY, side, side );
  if (level > 0) {
    int newLocX = locX + delta; int newLocY = locY + delta;
    drawBoxes(g, level - 1, newLocX, newLocY,
                                  side - 2 * delta, delta);
  } // if
} // drawBoxes()
```

Finally, note the use of the delta parameter, which is used to change the length of the sides by a fixed amount, 2 * delta, at each level. It is also used to calculate the *x*- and *y*-coordinates for the location of the next level of boxes *(locX + delta, locY + delta)*. But delta's value remains constant through all the levels. This will lead to a pattern where the "gap" between nested squares is constant.

Trace through the drawBoxes() method and draw the level-four and level-five versions of the nested boxes pattern. Assume that the initial values for side and delta are 100 and 5, respectively, and the initial coordinates for (locX,locY) are (20,20).

The pattern shown in Figure 12.23 can be drawn by using delta as a fixed ratio of the length of the side, for example, 10 percent. Modify the drawBoxes() method to use delta in this way.

Write an iterative version of the drawBoxes() method. (*Hint*: On each iteration, you must change the *x*- and *y*-coordinates of the square's location and the length of its side.)

The Sierpinski Gasket

Let's return now to the *Sierpinski gasket* pattern that we introduced at the start of this chapter. This is a much more interesting fractal pattern (Fig. [fig-sierpinski]). The overall shape of the pattern is that of a triangle, but notice how the outer triangle is divided into three smaller triangles. Then each of those triangles are divided into three smaller triangles. If you continue this process of dividing and shrinking, you get the level-seven pattern shown here.

Let's develop a recursive method to draw this pattern. If we follow the same strategy we used in the nested squares example, we get the following algorithm:

```
Base case:       Draw a triangle.
Recursive Case:  If more divisions are desired,
     draw three smaller gaskets within the triangle.
```

For this pattern the base case is the drawing of the basic triangle. The recursive cases, if more divisions are desired, are the drawing of smaller gaskets within the triangle. Again we will use a level parameter to control the depth of the recursion. The higher the level, the more divisions will be drawn.

If we're going to draw a triangle shape, we need the coordinates of its three vertices—that is, an *x*- and *y*-coordinate for each vertex. Taken together, these design considerations suggest the method definition shown in Figure [fig-gasket].

-2pc

```
/**
 * drawGasket()---recursively draws the Sierpinski gasket
 *  pattern, with points (p1X, p1Y),(p2X, p2Y),(p3X, p3Y)
 *  representing the vertices of its enclosing triangle.
 * level (>= 0) is the recursion parameter (base: = 0)
 */
private void drawGasket(Graphics g, int lev, int p1X, int p1Y,
                   int p2X, int p2Y, int p3X, int p3Y) {
  g.drawLine(p1X, p1Y, p2X, p2Y);  // Draw a triangle
  g.drawLine(p2X, p2Y, p3X, p3Y);
  g.drawLine(p3X, p3Y, p1X, p1Y);
  if (lev > 0) { // If more levels, draw 3 smaller gaskets
    int q1X = (p1X + p2X) / 2;    int q1Y = (p1Y + p2Y) / 2;
    int q2X = (p1X + p3X) / 2;    int q2Y = (p1Y + p3Y) / 2;
    int q3X = (p2X + p3X) / 2;    int q3Y = (p2Y + p3Y) / 2;
    drawGasket(g, lev - 1, p1X, p1Y, q1X, q1Y, q2X, q2Y);
    drawGasket(g, lev - 1, p2X, p2Y, q1X, q1Y, q3X, q3Y);
    drawGasket(g, lev - 1, p3X, p3Y, q2X, q2Y, q3X, q3Y);
  } // if
} // drawGasket()
```

As we described earlier, we use the level parameter as the recursion parameter for this method, which controls the recursion. Note that each of the three recursive calls decreases the level by 1. This will ensure that eventually level will equal 0, and recursion will stop.

Note also how the three pairs of coordinates are used. Drawing a triangle is simple. Just draw three lines from (p1X,p1Y) to (p2X,p2Y), from (p2X,p2Y) to (p3X,p3Y), and from (p3X,p3Y) back to (p1X, p1Y). The most complicated part of the method is calculating the vertices for the three inner gaskets. If you look at Figure [fig-sierpinski] again, you'll notice that each of the inner triangles uses one of the vertices of the main triangle, plus the *midpoints* of the two adjacent sides. Thus, the triangle on the "left" uses the left vertex (p1X,p1Y), and the midpoints of the other two lines from (p1X,p1Y) to (p2X,p2Y) and from (p1X,p1Y) to (p3X,p3Y). As you might remember from high school math, the formula for computing the midpoint of the line segment $(x1, y1)$ to $(x2, y2)$ is

```
( (x1 + x2) / 2, (y1 + y2) / 2 )
```

This formula is used repeatedly to calculate the vertices of the three smaller gaskets.

This page titled 10.5: Example- Drawing (Recursive) Fractals is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **12.5: Example- Drawing (Recursive) Fractals** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

10.6: Object-Oriented Design- Tail Recursion

Although the drawBoxes() method is relatively simple to convert into an iterative version (see Self-Study Exercise 12.18), the same cannot be said for the drawGasket() method. It is clearly a case where the recursive approach makes the problem easier to solve.

One difference between drawBoxes() and drawGasket() is that drawBoxes() is an example of a tail-recursive method. A method is **tail recursive** if all of its recursive calls occur as the last action performed in the method. You have to be a bit careful about this definition. The recursive call in a tail-recursive method has to be the last *executed* statement. It needn't be the last statement appearing in the method's definition.

For example, the following method will print "Hello" *N* times. This method is tail recursive even though its last statement is not a recursive call:

```
public void printHello(int N) {
 if (N > 1) {
   System.out.println("Hello");
   printHello(N - 1); // The last executed statement
 } else
   System.out.println("Hello");
} // printHello()
```

This method is tail recursive because the last statement that will be executed, in its recursive cases, is the recursive call.

A tail-recursive method is relatively easy to convert into an iterative method. The basic idea is to make the recursion parameter into a loop variable, taking care to make sure the bounds are equivalent. Thus, the following iterative method will print "Hello" *N* times:

```
public void printHelloIterative(int N) {
    for (int k = N; k > 0; k--)
        System.out.println("Hello");
}
```

In this case, we use the parameter *N* to set the initial value of the loop variable, *k*, and we decrement *k* on each iteration. This is equivalent to what happens when we decrement the recursion parameter in the recursive call.

As you can see, recursive methods that are not tail recursive are much more complex. Just compare the drawGasket() and drawBoxes() methods. Yet it is precisely for these nontail-recursive algorithms that recursion turns out to be most useful. As you might expect, if you can't give a simple tail-recursive solution to a problem, the problem probably doesn't have a simple iterative solution either. Thus, the problems where we most need recursion are those where we can't give a simple tail-recursive or a simple iterative solution. And there are a lot of such problems, especially when you get into nonlinear data structures such as trees and graphs.

To gain some appreciation for this complexity, consider how difficult it would be to draw the Sierpinski gasket using an iterative approach. We could start by developing an outer for loop to account for the different levels in the pattern:

```
for (int k = lev; k > 0; k--) {
 drawGasket(g, lev - 1, p1X, p1Y, q1X, q1Y, q2X, q2Y);
 drawGasket(g, lev - 1, p2X, p2Y, q1X, q1Y, q3X, q3Y);
 drawGasket(g, lev - 1, p3X, p3Y, q2X, q2Y, q3X, q3Y);
}
```

But now each of the method calls within the body of this loop would have to be replaced by very complex loops. That would be a daunting task. So the lesson to be drawn from this observation is that recursion is most useful as a problem-solving technique for problems that don't yield to a simple iterative solution.

Trace the drawGasket() method for levels two and three. Pick your own values for the three vertices.

Is the printReverse() method, discussed earlier, tail recursive? Explain.

Is the countChar() method, discussed earlier, tail recursive? Explain.

This page titled 10.6: Object-Oriented Design- Tail Recursion is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **12.6: Object-Oriented Design- Tail Recursion** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

10.7: Object-Oriented Design- Recursion or Iteration?

As we mentioned at the outset of this chapter, recursive algorithms require more computational overhead than iterative algorithms. We're now in a good position to appreciate why this is so.

A recursive algorithm incurs two kinds of overhead that are not incurred by an iterative algorithm: memory and CPU time. Both of these are direct results of the fact that recursive algorithms do a lot of method calling.

As we saw in our various traces, each time a method is called, a representation of the method call is placed on the *method call stack*. These often take the form of a *block* of memory locations, which can be quite large. The block must contain space for the method's local variables, including its parameters. Also, unless the method is void, the block must contain space for the method's return value. In addition it must contain a reference to the calling method, so it will know where to go when it is done. Figure [fig-methodblock] shows what the method call block would look like for the search() method.

In addition to the memory required, a method call also requires extra CPU time. Each time a method is called, Java must create a method call block, copy the method call arguments to the parameters in the block, create initial values for any local variables that are used by the method, and fill in the return address of the calling method. All of this takes time, and in the case of a recursive method, these steps are repeated at each level of the recursion.

Compare these memory and CPU requirements with what normally transpires for an iterative algorithm—an algorithm involving a loop. The loop structure usually occurs entirely within a method, so it doesn't incur either the memory or CPU overhead involved in recursion. Therefore, iterative algorithms are generally more efficient than recursive algorithms. One useful guideline, then, is when runtime performance and efficiency are of prime importance, you should use iteration instead of recursion.

On the other hand, recursive algorithms are much easier to design than the corresponding iterative algorithms for many problems. We tried to illustrate this point in our development of the Sierpinski gasket algorithm, but there are many other examples that we could have used. Given that programmer and designer time is the most expensive resource involved in software development, a recursive solution may be easier to develop and maintain than a corresponding iterative solution. And given the great cost of software development, a less efficient solution that is easier to develop, easier to understand, and easier to maintain may be preferable to a highly efficient algorithm that's difficult to understand. For some problems then, such as the Sierpinski gasket, a recursive algorithm may provide the best solution.

One final point that's worth making is that some *optimizing* compilers are able to convert recursive methods into iterative methods when they compile the program. The algorithms for doing this are well known. They are often subjects for study in a data structures course, so we won't go into them here. The resulting runtime programs will be just as efficient, in CPU time and memory, as if you had written iterative methods. The point is that if you have such a compiler, you really get the best of both worlds. You get the advantage of using recursion as a problem-solving and software development approach, and the compiler takes care of producing an efficient object program.

This page titled 10.7: Object-Oriented Design- Recursion or Iteration? is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **12.7: Object-Oriented Design- Recursion or Iteration?** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

10.8: Special Topic- Exploring the Mandelbrot Set

The Mandelbrot set is one of the most fascinating fractals. It is named after its discover, IBM mathematician Benoit Mandelbrot. The Mandelbrot set itself is the black, heart-shaped image shown in Figure [fig-mandel]. What makes the Mandelbrot set so interesting is that with the help of a Java GUI you can explore the set as if you were taking a trip through outer space. The most interesting regions to explore are those just along the boundary of the set. For example, notice that the boundary contains numerous circular shapes, each of which is itself studded with circular shapes. This is an example of the scaled self-similarity that we found to be so prevalent in recursive structures. By continually expanding the regions around the boundary, you'll find an infinite recursion of fascinating images and shapes. In some regions of the set you'll even find miniature replications of the set itself.

The Mandelbrot set is generated by an *iterated function system*. The mathematics underlying this fascinating object is quite accessible, and there are a number of online tutorials that explain how the set is generated and how the pictures are produced. Many of the Mandelbrot and fractal Web sites contain excellent Java applets that let you explore the Mandelbrot set as well as related sets. An excellent place to start your exploration would be David Joyce's award-winning Web site,

```
http://aleph0.clarku.edu/~djoyce/julia/
```

which contains references to a number of other good sites. For a tutorial on how the various Mandelbrot set-generating Java programs work, see

```
http://storm.shodor.org/mteach/
```

This page titled 10.8: Special Topic- Exploring the Mandelbrot Set is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **12.8: Special Topic- Exploring the Mandelbrot Set** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

10.9: From the Java Library- javax.swing.JComboBox

is a Swing component that combines a text field and a drop-down list (Fig. 12.28). It lets the user either type in a selection or choose a selection from a list that appears when the user requests it—a JComboBox's drop-down behavior is somewhat similar to a java.awt.Choice box.

A JComboBox can be used to represent a *drop-down menu*. When the user clicks on a JComboBox, a list of options drops down, and the user can select a particular option that is stored in the box's internal state (Fig. 12.29). The list of options associated with a JComboBox can be built beforehand and inserted into the component in a constructor, or items can be inserted one at a time by repeatedly using its addItem() method.

As Figure 12.28 shows, either an array or a vector of items can be passed to a constructor method to initialize the box's menu. The items stored in a JComboBox box are references to Objects, most commonly Strings that represent the name of the menu item. They are stored in the (zero indexed) order in which they are added. The addItem() method is used to add an individual Object to a JComboBox. By default, the first item added to a JComboBox will be the *selected* item until the user selects another item.

When the user makes a selection in a JComboBox, the item selected can be gotten either by its reference (getSelectedItem()) or by its position within the menu (getSelectedIndex()). There are also methods to setSelectedItem() and setSelectedIndex() that let you select an individual item either by its reference or its position. The addItemListener() method is used to designate some object as the listener for the ItemEvents that are generated whenever the user selects a menu option. Alternatively, the method lets you handle action events, such as when the user types a value into the box.

A JComboBox Example

As a simple example, let's design an graphical interface that can be used to display the fractal patterns we developed earlier. We want an interface that lets the user select from among the available patterns—we'll use the Sierpinski gasket and nested boxes for starters. In addition, the user should also be able to select different levels for the drawings, from 0 to 9. We want to present these options in two menus, with one JComboBox for each menu.

The first step is to declare and instantiate the JComboBoxes as instance variables:

```
private String items[] =
    {"Sierpinski Gasket","Nested Boxes"};
private JComboBox patterns = new JComboBox(items);
private JComboBox levels = new JComboBox();
```

Note that in this case we pass the constructor for the menu an entire array of items. If we hadn't done it this way, we would add individual items to the combo box in the JFrame's constructor RecursivePatterns(). In fact, that's how we'll initialize the levels menu:

```
for (int k=0; k < 10; k++)   // Add 10 levels
    levels.addItem(k + "" );
levels.setSelectedItem("4"); // Select default level
```

This loop would be placed in the JFrame's constructor, RecursivePatterns(). It adds strings representing levels 0 to 9 to the menu and initializes the box so that level four is showing as the default option.

Our next step is to designate the JFrame as the ItemListener for both menus—that is, the JFrame is named as the object that will handle the events that occur in the JComboBoxes. Then we add the JComboBox component to the JFrame:

```
controls.add(levels);    // Control panel for menus
controls.add(patterns);
                                 // Add the controls
getContentPane().add(controls, "North");
                            // And the drawing panel
getContentPane().add(canvas, "Center");
              // Register the menus with a listener
levels.addItemListener(this);
patterns.addItemListener(this);
```

Note that we use a separate controls panel (a JPanel) for the two menus and a canvas panel (another JPanel) for the drawings.

The next step is to implement the itemStateChanged() method to handle the user's selections. Whenever the user selects an item from a JComboBox menu, an ItemEvent is generated. In order to handle these events, the program must implement the ItemListener interface, which consists of the single method itemStateChanged(). This method is invoked automatically whenever the user selects an item from one of the JComboBoxes:

```
public void itemStateChanged(ItemEvent e) {
    canvas.setPattern(patterns.getSelectedIndex(),
                          levels.getSelectedIndex());
    repaint();
}
```

The itemStateChanged() method has the same general form as the actionPerformed() method, except that its parameter is an . For this example, the program uses the getSelectedIndex() method to get the selected pattern and the selected level by their respective item numbers within the menus. It then passes these values along to the canvas object, which takes care of the drawing. Finally, the method invokes the repaint() method. Because the JFrame is a container, this will cause all of its components to be repainted as well.

Figure [fig-seqdiag] illustrates the sequence of events that occurs when an item is selected from a JComboBox. The complete implementation for the program is given in Figure [fig-recursivepatterns].

-3pc

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class RecursivePatterns extends JFrame implements ItemListener  {
  private String choices[] = {"Sierpinski Gasket", "Nested Boxes"};
  private JComboBox patterns = new JComboBox(choices); // Pattern choices
  private JComboBox levels = new JComboBox();          // Level choices
  private Canvas canvas = new Canvas();                // Drawing panel
  private JPanel controls = new JPanel();

  public RecursivePatterns() {
    for (int k=0; k < 10; k++)            // Add 10 levels
      levels.addItem(k + "" );
    patterns.setSelectedItem(choices[0]); // Initialize menus
    levels.setSelectedItem("4");

    canvas.setBorder(BorderFactory.createTitledBorder("Drawing Canvas"));
    controls.add(levels);         // Control panel for menus
    controls.add(patterns);
    getContentPane().add(controls,"North"); // Add controls
    getContentPane().add(canvas,"Center");  // Add drawing panel
    levels.addItemListener(this);   // Register menus with listener
    patterns.addItemListener(this);
    setSize(canvas.WIDTH,canvas.HEIGHT+controls.getSize().width);
  } // init()

  public void itemStateChanged(ItemEvent e) {
    canvas.setPattern(patterns.getSelectedIndex(),
                                  levels.getSelectedIndex());
    repaint();                          // Repaint the JFrame
  } // itemStateChanged()

   public static void main(String args[])
   {
      JFrame f = new RecursivePatterns();
      f.setVisible(true);
   }
} // RecursivePatterns
```

The actual drawing of the fractal patterns is handled by the canvas JPanel component, whose design is shown in Figure [fig-canvas] and whose implementation is given in Figure [fig-canvaspanel]. All of the drawing is done in the paintComponent() method. Because the canvas is contained within the JFrame, the paintComponent() method is called automatically whenever the JFrame repaints itself. Notice how the switch statement uses the pattern that the user chose to call the corresponding drawing method. You can see from this switch statement that a JComboBox's items are *zero indexed*.

-7pc

```
import javax.swing.*;
import java.awt.*;

public class Canvas extends JPanel {
  private static final int GASKET = 0, BOXES = 1;
  public static final int WIDTH=400, HEIGHT=400;
  private final int HBOX=10, VBOX=50, BOXSIDE=200, BOXDELTA=10;
  private final int gP1X = 10;  private final int gP1Y = 280; // Initial
  private final int gP2X = 290; private final int gP2Y = 280; // gasket
  private final int gP3X = 150; private final int gP3Y = 110; // points
  private int pattern = 0 ;                     // Current pattern
  private int level = 4;                        // Current level

  public Canvas() {
    setSize(WIDTH, HEIGHT);
  }

  public void setPattern(int pat, int lev) {
    pattern = pat;
    level = lev;
  }
```

```
    public void paintComponent(Graphics g) {
        g.setColor(getBackground());   // Redraw the panel's background
        g.drawRect(0, 0, WIDTH, HEIGHT);
        g.setColor(getForeground());
        switch (pattern) {
        case GASKET:
            drawGasket(g, level, gP1X, gP1Y, gP2X, gP2Y, gP3X, gP3Y);
            break;
        case BOXES:
            drawBoxes(g, level, HBOX, VBOX, BOXSIDE, BOXDELTA );
            break;
        } // switch
    } // paintComponent()

   /** drawGasket()---recursively draws the Sierpinski
    *  gasket pattern, with points (p1X, p1Y), (p2X, p2Y), (p3X, p3Y)
    *  representing the vertices of its enclosing triangle.
    * level (>= 0) is the recursion parameter (base case: level  0)
    */
    private void drawGasket(Graphics g, int lev, int p1X, int p1Y,
                   int p2X, int p2Y, int p3X, int p3Y) {
        g.drawLine(p1X, p1Y, p2X, p2Y);  // Draw a triangle
        g.drawLine(p2X, p2Y, p3X, p3Y);
        g.drawLine(p3X, p3Y, p1X, p1Y);
        if (lev > 0) { // If more levels, draw 3 smaller gaskets
            int q1X = (p1X + p2X) / 2;    int q1Y = (p1Y + p2Y) / 2;
            int q2X = (p1X + p3X) / 2;    int q2Y = (p1Y + p3Y) / 2;
            int q3X = (p2X + p3X) / 2;    int q3Y = (p2Y + p3Y) / 2;
            drawGasket(g, lev - 1, p1X, p1Y, q1X, q1Y, q2X, q2Y);
            drawGasket(g, lev - 1, p2X, p2Y, q1X, q1Y, q3X, q3Y);
            drawGasket(g, lev - 1, p3X, p3Y, q2X, q2Y, q3X, q3Y);
        }
    } // drawGasket()
   /** drawBoxes()---recursively draws pattern of nested squares
    *  with (locX, locY) the top left corner of outer the square and
    *  side being the length square's side.
    * level (>= 0) is the recursion parameter (base case: level  0)
    * delta is used to adjust the length of the side.
    */
    private void drawBoxes(Graphics g, int level,
           int locX, int locY, int side, int delta) {
        g.drawRect(locX, locY, side, side );
        if (level > 0) {
            int newLocX = locX + delta; int newLocY = locY + delta;
            drawBoxes(g, level - 1, newLocX, newLocY,
                                       side - 2 * delta, delta);
        }
    } // drawBoxes()
} // Canvas
```

base case

computational overhead

head-and-tail algorithm

iterative method

last-in-first-out (LIFO)

method call stack

recursion parameter

recursive case

recursive definition

recursive method

self-similarity

tail recursive

A *recursive definition* is one that defines the *n*th case of a concept in terms of the $(n-1)$st case plus a limiting condition. It is based on the idea of breaking a problem up into smaller, self-similar problems.

A *recursive method* is one that calls itself. It is usually defined in terms of a *base case* or limiting case, which stops the recursive process, and a recursive case, which breaks the method into a smaller, self-similar copy of itself. A *recursion parameter* is generally used to control the recursion.

An iterative algorithm is one that uses some kind of loop as its control structure. Any algorithm that can be done iteratively can also be done recursively, and vice versa.

Because method calling is relatively costly both in terms of memory used and CPU time involved, a recursive algorithm is generally less efficient than an iterative one that does the same thing.

In designing recursive algorithms, the *base case* defines a limit. Each level of recursion should make progress toward the limit, and the algorithm should eventually reach the limit. The limit is usually expressed in terms of the *recursion parameter*.

A recursive method is *tail recursive* if and only if each of its recursive calls is the last action executed by the method.

A Swing JComboBox component is used to represent a GUI drop-down menu.

The output produced by mystery(0) would be 0 1 2 3 4 5 6. The output produced by mystery(100) would be 100.

The output produced by mystery(5) would be: 5 4 3, and so on. In other words, this is an infinite recursion.

```
Definition: twoToN(N), N >= 0
  1, if N == 0                  // Base case
  2 * twoToN(N - 1),  N > 0     // Recursive case
```

The function x^n is known as the power function:

```
Definition: power(X,N), N >= 0
  1, if N == 0                   // Base case
  X * power(X, N - 1),  N > 0    // Recursive case
```

Yes, the two definitions for nested boxes are equivalent. Suppose the square starts out with a side of 20. The definition given in the exercise will also draw squares with sides of 20, 15, 10, 5.

A recursive definition for the pattern in Figure 12.4:

```
Draw a square with side, s.
Inscribe a circle with diameter, s.
If s > 5,
  Draw a smaller version of same pattern. // Recursive case
```

The printString2("hello") method will print: “olleh.”

A definition for countDown():

```
 /** countDown(N) recursively prints a countdown
  *  beginning at N and ending at 1
  * @param N >= 1
  * Base case: N == 0
  */
void countDown(int N) {
    if (N == 0)                     // Base case
        System.out.println("blastoff");
    else {
        System.out.print(N + ", "); // Recursive case
        countDown(N - 1);
    }
} // countDown()
```

A revised definition for countDown():

```
 /** countDown(N) recursively prints a countdown
  *  beginning at N, counting every other number, 10 8 6 ...
  *  and ending at "blastoff"
  * @param N >= 1
  * Base case: N <= 0
  */
void countDown(int N) {
    if (N <= 0)                     // Base case
        System.out.println("blastoff");
    else {
        System.out.print(N + ", "); // Recursive case
        countDown(N - 2 );
    }
} // countDown()
```

A method to sum the numbers from 1 to *N*.

```
int sum(int N) {
    if (N == 0)
        return 0;
    else
        return N + sum(N-1);
}
```

A method to change each blank within a string to two blanks.

```
String addBlanks(String s) {
  if (s.length() == 0)
     return "";
  else if (s.charAt(0) == ' ')
     return ' ' + s.charAt(0) + addBlanks(s.substring(1));
  else
     return s.charAt(0) + addBlanks(s.substring(1));
}
```

A method to print out all possible outcomes for a chess player playing N games. printOutcomes(str, N) will print all outcomes for the next N games given that results for previous games are stored in the string named str.

```
 public static void printOutcomes(String str, int N){
     if (N = 1){ // Base case: win, lose, or draw one game
         System.out.println(str + "W");
         System.out.println(str + "L");
         System.out.println(str + "D");
     } else {  // Recursive case
         printOutcomes(str + "W", N - 1);
         printOutcomes(str + "L", N - 1);
         printOutcomes(str + "D", N - 1);
     } //else
 }// printOutcomes()
```

```
public static void main(String args[]) {
  int numbers[] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18};
  Searcher searcher = new Searcher();
  for (int k = 0; k <= 20; k++) {
    int result = searcher.search(numbers, k);
    if (result != -1)
      System.out.println(k + " found at " + result);
    else
      System.out.println(k + " is not in the array ");
  } // for
} // main()
```

The sort() method is used as a public interface to the recursive selectionSort() method:

```
/** sort(arr) sorts the int array, arr
 *  Pre: arr is not null
 *  Post: arr will be arranged so that arr[j] <= arr[k]
 *     for any j < k
 */
public void sort(int arr[]) {
    selectionSort(arr, arr.length - 1);
              // Just call the recursive method
}
```

An iterative version of findMax():

```
 /** findMax (arr,N) returns the index of the largest
  *  value between arr[0] and arr[N], N >= 0.
  *  Pre: 0 <= N <= arr.length -1
  *  Post: arr[findMax()]>=arr[k] for k between 0 and N.
  */
private int findMax(int arr[], int N) {
    int maxSoFar = 0;
    for (int k = 0; k <= N; k++)
        if (arr[k] > arr[maxSoFar])
            maxSoFar = k;
    return maxSoFar;
} // findMax()
```

Levels four and five of the nested boxes pattern are shown in Figure 12.34.

The following method will reduce the length of the side by delta percent at each level of recursion. The spacing between the boxes will vary by a constantly decreasing amount.

```
private void  drawBoxes(Graphics g, int level, int locX,
                  int locY, int side, int delta) {
    g.drawRect(locX, locY, side, side );
    if (level > 0) {
      int dside = side * delta / 100; // Percent delta
      int newLocX = locX + dside;
      int newLocY = locY + dside;
      drawBoxes(g, level - 1, newLocX, newLocY,
                        side - 2 * dside, delta);
    }
} // drawBoxes()
```

```
private void drawBoxesIterative(Graphics g, int level,
            int locX, int locY, int side, int delta) {
 for (int k = level; k >= 0; k--) {
  g.drawRect(locX, locY, side, side );  // Draw a square
  locX += delta;              // Calculate new location
  locY += delta;
  side -= 2 * delta;          // Calculate new side length
 }
} // drawBoxes()
```

The level two and three gaskets are shown in Figure [fig-sierpinski23].

The printReverse() method is not tail recursive because in that method the recursive call is not the last statement executed.

The countChar() method is tail recursive. The recursive calls are not the last statements in the method definition. However, each of the recursive calls would be the last statement executed by the method.

Explain the difference between the following pairs of terms:

Iteration and *recursion*.

Recursive method and *recursive definition*.

Base case and *recursive case*.

Head and *tail*.

Tail and *nontail* recursive.

Describe how the *method call stack* is used during a method call and return.

Why is a recursive algorithm generally less efficient than an iterative algorithm?

A tree, such as a maple tree or pine tree, has a recursive structure. Describe how a tree's structure displays *self-similarity* and *divisibility*.

Write a recursive method to print each element of an array of double.

Write a recursive method to print each element of an array of double from the last to the first element.

Write a recursive method that will concatenate the elements of an array of String into a single String delimited by blanks.

Write a recursive method that is passed a single int parameter, $N \geq 0$, and prints all the odd numbers between 1 and *N*.

Write a recursive method that takes a single int parameter and prints the sequence of even numbers between *N* down to 0.

Write a recursive method that takes a single int parameter $N \geq 0$ and prints the multiples of 10 between 0 and *N*.

Write a recursive method to print the following geometric

```
#
# #
# # #
# # # #
# # # # #
```

Write recursive methods to print each of the following

```
# # # # # # # # #      # # # # # # # # #
  # # # # # # # #      # # # # # # # #
    # # # # # # #      # # # # # # #
      # # # # # #      # # # # # #
        # # # # #      # # # # #
          # # # #      # # # #
            # # #      # # #
              # #      # #
                #      #
```

Write a recursive method to print all multiples of *M* up to * *N*.

Write a recursive method to compute the sum of grades stored in an array.

Write a recursive method to count the occurrences of a substring within a string.

Write a recursive method to remove the HTML tags from a string.

Implement a recursive version of the Caesar.decode() method from Chapter 8.

The Fibonacci sequence (named after the Italian mathematician Leonardo of Pisa, ca. 1200) consists of the numbers 0,1,1,2,3,5,8,13,… in which each number (except for the first two) is the sum of the two preceding numbers. Write a recursive method fibonacci(N) that prints the first *N* Fibonacci numbers.

Write a recursive method to rotate a String by *N* characters to the right. For example, rotateR("hello", 3) should return "llohe."

Write a recursive method to rotate a String by *N* characters to the left. For example, rotateL("hello", 3) should return "lohel."

Write a recursive method to convert a String representing a binary number to its decimal equivalent. For example, binTodecimal("101011") should return the int value 43.

A palindrome is a string that is equal to its reverse—"mom," "i," "radar" and "able was i ere i saw elba." Write a recursive boolean method that determines whether its String parameter is a palindrome.

Challenge: Incorporate a drawBinaryTree() method into the RecursivePatterns program. A level-one binary tree has two branches. At each subsequent level, two smaller branches are grown from the endpoints of every existing branch. The geometry is easier if you use 45-degree angles for the branches. Figure 12.36 shows a level-four binary tree drawn upside down.

Challenge: Towers of Hanoi. According to legend, some Buddhist monks were given the task of moving 64 golden disks from one diamond needle to another needle, using a third needle as a backup. To begin with, the disks were stacked one on top of the other from largest to smallest (Fig. 12.37). The rules were that only one disk can be moved at a time and that a larger disk can never go on top of a smaller one. The end of the world was supposed to occur when the monks finished the task!

Write a recursive method, move(int N, char A, char B, char C), that will print out directions the monks can use to solve the towers of Hanoi problem. For example, here's what it should output for the three-disk case, move(3, "A", "B", "C"):

```
Move 1 disk from A to B.
Move 1 disk from A to C.
Move 1 disk from B to C.
Move 1 disk from A to B.
Move 1 disk from C to A.
Move 1 disk from C to B.
Move 1 disk from A to B.
```

This page titled 10.9: From the Java Library- javax.swing.JComboBox is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Ralph Morelli & Ralph Wade via source content that was edited to the style and standards of the LibreTexts platform.

- **12.9: From the Java Library- javax.swing.JComboBox** by Ralph Morelli & Ralph Wade is licensed CC BY 4.0. Original source: http://turing.cs.trincoll.edu/~ram/jjj.

Index

Glossary

Sample Word 1 | Sample Definition 1

Detailed Licensing

Some of the pages in this book are re-mixes of open content from other books in the LibreTexts Library. Sometimes sections from several sources are used on the same page and there may be no unique open license. Where there is a conflict, the licensing may appear as *undeclared*. On any page, by toggling AutoAttibution on the Tools menu, you can display the detailed attribution on a paragraph level. Material from different books will be color coded and the source displayed by mousing over the paragraph.

Content in the Front and Back Matter, directories and similar can be uncopywriteable, being simple listings and these are also shown as *undeclared*.

Overview

Title: CS 113 Programming in Java (Milan)

Webpages: 109

All licenses found:

- CC BY 4.0: 75.2% (82 pages)
- Undeclared: 24.8% (27 pages)

By Page

- CS 113 Programming in Java (Milan) - *CC BY 4.0*
  - Front Matter - *Undeclared*
    - TitlePage - *Undeclared*
    - InfoPage - *Undeclared*
    - Table of Contents - *Undeclared*
    - Licensing - *Undeclared*
  - 1: Java Program Design and Development - *CC BY 4.0*
    - 1.1: Introduction - *CC BY 4.0*
    - 1.2: Designing Good Programs - *CC BY 4.0*
    - 1.3: Designing a Riddle Program - *CC BY 4.0*
    - 1.4: Java Language Elements - *CC BY 4.0*
    - 1.5: Editing, Compiling, and Running a Java Program - *CC BY 4.0*
    - 1.6: From the Java Library- System and PrintStream - *CC BY 4.0*
  - 2: Objects- Defining, Creating, and Using - *CC BY 4.0*
    - 2.1: Introduction - *CC BY 4.0*
    - 2.2: Using String Objects - *CC BY 4.0*
    - 2.3: Drawing Shapes with the Graphics Object (Optional) - *CC BY 4.0*
    - 2.4: Class Definition - *CC BY 4.0*
    - 2.5: Case Study- Simulating a Two-Person Game - *CC BY 4.0*
    - 2.6: From the Java Library- java.util.Scanner - *CC BY 4.0*
  - 3: Methods- Communicating with Objects - *CC BY 4.0*
    - 3.1: Introduction - *Undeclared*
    - 3.2: Passing Information to an Object - *CC BY 4.0*
    - 3.3: Constructor Methods - *CC BY 4.0*
    - 3.4: Retrieving Information from an Object - *CC BY 4.0*
    - 3.5: Passing a Value and Passing a Reference - *CC BY 4.0*
    - 3.6: Flow of Control- Control Structures - *CC BY 4.0*
    - 3.7: Testing an Improved OneRowNim - *CC BY 4.0*
    - 3.8: From the Java Library- java.lang.Object - *CC BY 4.0*
    - 3.9: Object-Oriented Design- Inheritance and Polymorphism - *CC BY 4.0*
    - 3.10: Drawing Lines and Defining Graphical Methods (Optional) - *CC BY 4.0*
  - 4: Input/Output- Designing the User Interface - *CC BY 4.0*
    - 4.1: Introduction - *Undeclared*
    - 4.2: The User Interface - *CC BY 4.0*
    - 4.3: A Command-line Interface - *CC BY 4.0*
    - 4.4: A Graphical User Interface (GUI) - *CC BY 4.0*
    - 4.5: Case Study- The One Row Nim Game - *CC BY 4.0*
    - 4.6: From the Java Library- java.io.File and file input (Optional) - *CC BY 4.0*
  - 5: Java Data and Operators - *CC BY 4.0*
    - 5.1: Introduction - *Undeclared*
    - 5.2: Boolean Data and Operators - *CC BY 4.0*
    - 5.3: Numeric Data and Operators - *CC BY 4.0*
    - 5.4: From the Java Library java.lang.Math - *CC BY 4.0*
    - 5.5: Numeric Processing Examples - *CC BY 4.0*
    - 5.6: From the Java Library- java.text.NumberFormat - *Undeclared*
    - 5.7: Character Data and Operators - *Undeclared*
    - 5.8: Example- Character Conversions - *Undeclared*

- 5.9: Problem Solving = Representation + Action - *Undeclared*

- 6: Control Structures - *CC BY 4.0*
  - 6.1: Introduction - *Undeclared*
  - 6.2: Flow of Control- Repetition Structures - *CC BY 4.0*
  - 6.3: Counting Loops - *CC BY 4.0*
  - 6.4: Example- Car Loan - *CC BY 4.0*
  - 6.5: Graphics Example- Drawing a Checkerboard - *CC BY 4.0*
  - 6.6: Conditional Loops - *CC BY 4.0*
  - 6.7: Example- Computing Averages - *CC BY 4.0*
  - 6.8: Example- Data Validation - *CC BY 4.0*
  - 6.9: Principles of Loop Design - *CC BY 4.0*
  - 6.10: The switch Multiway Selection Structure - *CC BY 4.0*
  - 6.11: Object-Oriented Design- Structured Programming - *CC BY 4.0*
  - 6.12: Special Topic- What Can Be Computed? - *CC BY 4.0*
- 7: Strings and String Processing - *CC BY 4.0*
  - 7.1: Introduction - *Undeclared*
  - 7.2: String Basics - *CC BY 4.0*
  - 7.3: Finding Things Within a String - *CC BY 4.0*
  - 7.4: Example- Keyword Search - *CC BY 4.0*
  - 7.5: From the Java Library- StringBuffer - *CC BY 4.0*
  - 7.6: Retrieving Parts of Strings - *CC BY 4.0*
  - 7.7: Example- Processing Names and Passwords - *CC BY 4.0*
  - 7.8: Processing Each Character in a String - *CC BY 4.0*
  - 7.9: Comparing Strings - *CC BY 4.0*
  - 7.10: From the Java Library- StringTokenizer - *CC BY 4.0*
  - 7.11: Handling Text in a Graphics Context (Optional) - *CC BY 4.0*
- 8: Arrays and Array Processing - *CC BY 4.0*
  - 8.1: Introduction - *CC BY 4.0*
  - 8.2: One-Dimensional Arrays - *CC BY 4.0*
  - 8.3: Simple Array Examples - *CC BY 4.0*
  - 8.4: Example- Counting Frequencies of Letters - *CC BY 4.0*
  - 8.5: Array Algorithms- Sorting - *CC BY 4.0*
  - 8.6: Array Algorithms- Searching - *CC BY 4.0*
  - 8.7: Two-Dimensional Arrays - *Undeclared*
  - 8.8: Multidimensional Arrays (Optional) - *Undeclared*
  - 8.9: Object-Oriented Design- Polymorphic Sorting (Optional) - *Undeclared*
  - 8.10: From the Java Library- java.util.Vector - *Undeclared*
  - 8.11: Case Study- An N-Player Computer Game - *Undeclared*
  - 8.12: A GUI-Based Game (Optional Graphics) - *Undeclared*
- 9: Exceptions- When Things Go Wrong - *CC BY 4.0*
  - 9.1: Introduction - *Undeclared*
  - 9.2: Handling Exceptional Conditions - *CC BY 4.0*
  - 9.3: Java's Exception Hierarchy - *CC BY 4.0*
  - 9.4: Handling Exceptions Within a Program - *CC BY 4.0*
  - 9.5: Error Handling and Robust Program Design - *CC BY 4.0*
  - 9.6: Creating and Throwing Your Own Exceptions - *CC BY 4.0*
  - 9.7: From the Java Library- javax.swing.JOptionPane - *CC BY 4.0*
- 10: Recursive Problem Solving - *CC BY 4.0*
  - 10.1: Introduction - *Undeclared*
  - 10.2: Recursive Definition - *CC BY 4.0*
  - 10.3: Recursive String Methods - *CC BY 4.0*
  - 10.4: Recursive Array Processing - *CC BY 4.0*
  - 10.5: Example- Drawing (Recursive) Fractals - *CC BY 4.0*
  - 10.6: Object-Oriented Design- Tail Recursion - *CC BY 4.0*
  - 10.7: Object-Oriented Design- Recursion or Iteration? - *CC BY 4.0*
  - 10.8: Special Topic- Exploring the Mandelbrot Set - *CC BY 4.0*
  - 10.9: From the Java Library- javax.swing.JComboBox - *CC BY 4.0*
- Back Matter - *Undeclared*
  - Index - *Undeclared*
  - Glossary - *Undeclared*
  - Detailed Licensing - *Undeclared*
  - Detailed Licensing - *Undeclared*

Detailed Licensing

Overview

Title: CS 113 Programming in Java (Milan)

Webpages: 109

All licenses found:

- CC BY 4.0: 75.2% (82 pages)
- Undeclared: 24.8% (27 pages)

By Page

- CS 113 Programming in Java (Milan) - *CC BY 4.0*
  - Front Matter - *Undeclared*
    - TitlePage - *Undeclared*
    - InfoPage - *Undeclared*
    - Table of Contents - *Undeclared*
    - Licensing - *Undeclared*
  - 1: Java Program Design and Development - *CC BY 4.0*
    - 1.1: Introduction - *CC BY 4.0*
    - 1.2: Designing Good Programs - *CC BY 4.0*
    - 1.3: Designing a Riddle Program - *CC BY 4.0*
    - 1.4: Java Language Elements - *CC BY 4.0*
    - 1.5: Editing, Compiling, and Running a Java Program - *CC BY 4.0*
    - 1.6: From the Java Library- System and PrintStream - *CC BY 4.0*
  - 2: Objects- Defining, Creating, and Using - *CC BY 4.0*
    - 2.1: Introduction - *CC BY 4.0*
    - 2.2: Using String Objects - *CC BY 4.0*
    - 2.3: Drawing Shapes with the Graphics Object (Optional) - *CC BY 4.0*
    - 2.4: Class Definition - *CC BY 4.0*
    - 2.5: Case Study- Simulating a Two-Person Game - *CC BY 4.0*
    - 2.6: From the Java Library- java.util.Scanner - *CC BY 4.0*
  - 3: Methods- Communicating with Objects - *CC BY 4.0*
    - 3.1: Introduction - *Undeclared*
    - 3.2: Passing Information to an Object - *CC BY 4.0*
    - 3.3: Constructor Methods - *CC BY 4.0*
    - 3.4: Retrieving Information from an Object - *CC BY 4.0*
    - 3.5: Passing a Value and Passing a Reference - *CC BY 4.0*
    - 3.6: Flow of Control- Control Structures - *CC BY 4.0*
    - 3.7: Testing an Improved OneRowNim - *CC BY 4.0*
    - 3.8: From the Java Library- java.lang.Object - *CC BY 4.0*
    - 3.9: Object-Oriented Design- Inheritance and Polymorphism - *CC BY 4.0*
    - 3.10: Drawing Lines and Defining Graphical Methods (Optional) - *CC BY 4.0*
  - 4: Input/Output- Designing the User Interface - *CC BY 4.0*
    - 4.1: Introduction - *Undeclared*
    - 4.2: The User Interface - *CC BY 4.0*
    - 4.3: A Command-line Interface - *CC BY 4.0*
    - 4.4: A Graphical User Interface (GUI) - *CC BY 4.0*
    - 4.5: Case Study- The One Row Nim Game - *CC BY 4.0*
    - 4.6: From the Java Library- java.io.File and file input (Optional) - *CC BY 4.0*
  - 5: Java Data and Operators - *CC BY 4.0*
    - 5.1: Introduction - *Undeclared*
    - 5.2: Boolean Data and Operators - *CC BY 4.0*
    - 5.3: Numeric Data and Operators - *CC BY 4.0*
    - 5.4: From the Java Library java.lang.Math - *CC BY 4.0*
    - 5.5: Numeric Processing Examples - *CC BY 4.0*
    - 5.6: From the Java Library- java.text.NumberFormat - *Undeclared*
    - 5.7: Character Data and Operators - *Undeclared*
    - 5.8: Example- Character Conversions - *Undeclared*
    - 5.9: Problem Solving = Representation + Action - *Undeclared*
  - 6: Control Structures - *CC BY 4.0*
    - 6.1: Introduction - *Undeclared*
    - 6.2: Flow of Control- Repetition Structures - *CC BY 4.0*
    - 6.3: Counting Loops - *CC BY 4.0*
    - 6.4: Example- Car Loan - *CC BY 4.0*
    - 6.5: Graphics Example- Drawing a Checkerboard - *CC BY 4.0*
    - 6.6: Conditional Loops - *CC BY 4.0*
    - 6.7: Example- Computing Averages - *CC BY 4.0*
    - 6.8: Example- Data Validation - *CC BY 4.0*
    - 6.9: Principles of Loop Design - *CC BY 4.0*

- 6.10: The switch Multiway Selection Structure - *CC BY 4.0*
- 6.11: Object-Oriented Design- Structured Programming - *CC BY 4.0*
- 6.12: Special Topic- What Can Be Computed? - *CC BY 4.0*

- 7: Strings and String Processing - *CC BY 4.0*
  - 7.1: Introduction - *Undeclared*
  - 7.2: String Basics - *CC BY 4.0*
  - 7.3: Finding Things Within a String - *CC BY 4.0*
  - 7.4: Example- Keyword Search - *CC BY 4.0*
  - 7.5: From the Java Library- StringBuffer - *CC BY 4.0*
  - 7.6: Retrieving Parts of Strings - *CC BY 4.0*
  - 7.7: Example- Processing Names and Passwords - *CC BY 4.0*
  - 7.8: Processing Each Character in a String - *CC BY 4.0*
  - 7.9: Comparing Strings - *CC BY 4.0*
  - 7.10: From the Java Library- StringTokenizer - *CC BY 4.0*
  - 7.11: Handling Text in a Graphics Context (Optional) - *CC BY 4.0*
- 8: Arrays and Array Processing - *CC BY 4.0*
  - 8.1: Introduction - *CC BY 4.0*
  - 8.2: One-Dimensional Arrays - *CC BY 4.0*
  - 8.3: Simple Array Examples - *CC BY 4.0*
  - 8.4: Example- Counting Frequencies of Letters - *CC BY 4.0*
  - 8.5: Array Algorithms- Sorting - *CC BY 4.0*
  - 8.6: Array Algorithms- Searching - *CC BY 4.0*
  - 8.7: Two-Dimensional Arrays - *Undeclared*
  - 8.8: Multidimensional Arrays (Optional) - *Undeclared*
  - 8.9: Object-Oriented Design- Polymorphic Sorting (Optional) - *Undeclared*
  - 8.10: From the Java Library- java.util.Vector - *Undeclared*
  - 8.11: Case Study- An N-Player Computer Game - *Undeclared*
  - 8.12: A GUI-Based Game (Optional Graphics) - *Undeclared*
- 9: Exceptions- When Things Go Wrong - *CC BY 4.0*
  - 9.1: Introduction - *Undeclared*
  - 9.2: Handling Exceptional Conditions - *CC BY 4.0*
  - 9.3: Java's Exception Hierarchy - *CC BY 4.0*
  - 9.4: Handling Exceptions Within a Program - *CC BY 4.0*
  - 9.5: Error Handling and Robust Program Design - *CC BY 4.0*
  - 9.6: Creating and Throwing Your Own Exceptions - *CC BY 4.0*
  - 9.7: From the Java Library- javax.swing.JOptionPane - *CC BY 4.0*
- 10: Recursive Problem Solving - *CC BY 4.0*
  - 10.1: Introduction - *Undeclared*
  - 10.2: Recursive Definition - *CC BY 4.0*
  - 10.3: Recursive String Methods - *CC BY 4.0*
  - 10.4: Recursive Array Processing - *CC BY 4.0*
  - 10.5: Example- Drawing (Recursive) Fractals - *CC BY 4.0*
  - 10.6: Object-Oriented Design- Tail Recursion - *CC BY 4.0*
  - 10.7: Object-Oriented Design- Recursion or Iteration? - *CC BY 4.0*
  - 10.8: Special Topic- Exploring the Mandelbrot Set - *CC BY 4.0*
  - 10.9: From the Java Library- javax.swing.JComboBox - *CC BY 4.0*
- Back Matter - *Undeclared*
  - Index - *Undeclared*
  - Glossary - *Undeclared*
  - Detailed Licensing - *Undeclared*
  - Detailed Licensing - *Undeclared*