CHAPTER 1

The Phases of Software Development

LEARNING OBJECTIVES

When you complete Chapter 1, you will be able to...

- use Javadoc to write a method's complete specification, including a precondition/postcondition contract.
- recognize quadratic, linear, and logarithmic runtime behavior in simple algorithms, and write big-O expressions to describe this behavior.
- create and recognize test data that is appropriate for a problem, including testing boundary conditions and fully exercising code.

CHAPTER CONTENTS

CHAPTER 1

The Phases of Software Development

Chapter the first which explains how, why, when, and where there was ever any problem in the first place

NOEL LANGLEY
The Land of Green Ginger

This chapter illustrates the phases of software development. These phases occur for all software, including the small programs you'll see in this first chapter. In subsequent chapters, you'll go beyond these small programs, applying the phases of software development to organized collections of data. These organized collections of data are called **data structures**, and the main topics of this book revolve around proven techniques for representing and manipulating such data structures.

Data Structure

A **data structure** is a collection of data, organized so that items can be stored and retrieved by some fixed techniques. For example, a Java array is a simple data structure that allows individual items to be stored and retrieved based on an index ([0], [1], [2]...) that is assigned to each item.

Throughout this book, you will be presented with many forms of data structures with organizations that are motivated by considerations such as ease of use or speed of inserting and removing items from the structure.

Years from now, you may be a software engineer writing large systems in a specialized area, perhaps computer graphics or artificial intelligence. Such futuristic application will be exciting and stimulating, and within your work you will still see the data structures that you learn and practice now. You will still be following the same phases of software development that you learned when designing and implementing your first programs. Here is a typical list of the software delvelopment phases:

The Phases of Software Development

- Specification of the task
- Design of a solution
- Implementation (coding) of the solution
- Analysis of the solution
- Testing and debugging
- Maintenance and evolution of the system
- Obsolescence

the phases blur into each other

You don't need to memorize this list; throughout the book, your practice of these phases will achieve far better familiarity than mere memorization. Also, memorizing an "official list" is misleading because it suggests that there is a single sequence of discrete steps that always occur one after another. In practice, the phases blur into each other; for instance, the analysis of a solution's efficiency may occur hand in hand with the design, before any coding. Or low-level design decisions may be postponed until the implementation phase. Also, the phases might not occur one after another. Typically, there is back and forth travel between the phases.

Most of the work in software development does not depend on any particular programming language. Specification, design, and analysis can all be carried out with few or no ties to a particular programming language. Nevertheless, when we get down to implementation details, we do need to decide on one particular programming language. The language we use in this book is **Java™ 2 Standard Edition 5.0 '(J2SE 5.0)**.

What You Should Know About Java Before Starting This Text

the origin of Java

The Java language was conceived by a group of programmers at Sun Microsystems in 1991. The group, led by James Gosling, had an initial design called Oak that was motivated by a desire for a single language in which programs could be developed and easily moved from one machine to another. Over the next four years, many other Sun programmers contributed to the project, and Gosling's Oak evolved into the Java language, including the current Java 2 Standard Edition 5.0. Java's goal of easily moving programs between machines was met by introducing an intermediate form called **byte codes**. To run a Java program, the program is first translated into byte codes; the byte codes are then given to a machine that runs a controlling program called the **Java Runtime Environment (JRE)**. Because the JRE is freely available for a wide variety of machines, Java programs can be moved from one machine to another. Because the JRE controls all Java programs, there is an added level of security that avoids potential problems from running unknown programs.

In addition to the original goals of program transportability and security, the designers of Java also incorporated ideas from other modern programming languages. Most notably, Java supports **object-oriented programming (OOP)** in a manner that was partly taken from the C++ programming language. OOP is a programming approach that encourages strategies of information hiding and component reuse. In this book, you will be introduced to these important OOP principles to use in your designs and implementations.

this book gives an introduction to OOP principles for information hiding and component reuse

All of the programs in this book have been developed and tested with Sun's **Java 2 Standard Development Kit (SDK)**, but many other Java programming environments may be successfully used with this text. You should be comfortable writing, compiling, and running short Java application programs in your environment. You should know how to use the Java primitive types (the number types, char, and boolean), and you should be able to use arrays.

The rest of this chapter will prepare you to tackle the topic of data structures in Java. Section 1.1 focuses on a technique for specifying program behavior, and

you'll also see some hints about design and implementation. Section 1.2 illustrates a particular kind of analysis: the running time analysis of a program. Section 1.3 provides some techniques for testing and debugging Java programs.

1.1 SPECIFICATION, DESIGN, IMPLEMENTATION

One begins with a list of difficult design decisions which are likely to change. Each module is then designed to hide such a decision from the others.

D. L. PARNAS
"On the Criteria to Be Used in Decomposing Systems into Modules"

you should already know how to write, compile, and run short Java programs in some programming environment

As an example of software development in action, let's examine the specification, design, and implementation for a particular problem. The **specification** is a precise description of the problem; the **design** phase consists of formulating the steps (or **algorithm**) to solve the problem; the **implementation** is the actual Java code to carry out the design.

```
TEMPERATURE CONVERSION
----------------------
Celsius     Fahrenheit
-50.00C
-40.00C     The equivalent
-30.00C     Fahrenheit
-20.00C     temperatures will be
-10.00C     computed and
  0.00C     displayed on this side
 10.00C     of the table.
 20.00C
 30.00C
 40.00C
 50.00C
----------------------
```

The problem we have in mind is to display a table for converting Celsius temperatures to Fahrenheit, similar to the table shown here. For a small problem, a sample of the desired output is a reasonable specification. Such a sample is *precise,* leaving no doubt as to what the program must accomplish. The next step is to design a solution.

An **algorithm** is a procedure or sequence of directions for solving a problem. For example, an algorithm for the temperature problem will tell how to produce the conversion table. An algorithm can be expressed in many different ways, such as in English, in a mixture of English and mathematical notation, or in a mixture of English with a programming language. This mixture of English and a programming language is called **pseudocode**. Using pseudocode allows us to avoid programming language details that may obscure a simple solution, but at the same time we can use Java code (or another language) when the code is clear. Keep in mind that the reason for pseudocode is to improve *clarity*.

Algorithm

An **algorithm** is a procedure or sequence of instructions for solving a problem. Any algorithm may be expressed in many different ways: in English, in a particular programming language, or (most commonly) in a mixture of English and programming called **pseudocode**.

We'll use pseudocode to design a solution for the temperature problem, and we'll also use the important design technique of decomposing the problem, which we'll discuss now.

Design Technique: Decomposing the Problem

Key Design Concept
Break down a task into a few subtasks; then decompose each subtask into smaller subtasks.

A good technique for designing an algorithm is to break down the problem at hand into a few subtasks, then decompose each subtask into smaller subtasks, then replace the smaller subtasks with even smaller subtasks, and so forth. Eventually the subtasks become so small that they are trivial to implement in Java or whatever language you are using. When the algorithm is translated into Java code, each subtask is implemented as a separate Java method. In other programming languages, methods are called "functions" or "procedures," but it all boils down to the same thing: The large problem is decomposed into subtasks, and subtasks are implemented as separate pieces of your program.

For example, the temperature problem has at least two good subtasks: converting a temperature from Celsius degrees to Fahrenheit and printing a number with a specified accuracy (such as rounding to the nearest hundredth). Using these two subproblems, the first draft of our pseudocode might look like this:

1. Display the labels at the top of the table.
2. For each line in the table (using variables `celsius` and `fahrenheit`):
   2a. Set `celsius` equal to the next Celsius temperature of the table.
   2b. `fahrenheit` = the `celsius` temperature converted to Fahrenheit.
   2c. Print one line of the output table with each temperature rounded to the nearest hundredth and labeled (by the letter C or F).
3. Print the line of dashes at the bottom of the table.

what makes a good decomposition?

The underlined steps (2b and 2c) are the major subtasks. But aren't there other ways to decompose the problem into subtasks? What are the aspects of a good decomposition? One primary guideline is that the subtasks should help you produce short pseudocode—no more than a page of succinct description to solve the entire problem and ideally much less than a page. In your first designs, you can also keep in mind two considerations for selecting good subtasks: the potential for code reuse and the possibility of future changes to the program. Let's see how our subtasks embody these considerations.

the printf method

Step 2c is a form of a common task: printing some information with a specified format. This task is so common that the latest version of Java has included a new method, **`System.out.printf`**, that can be used by any program that produces formatted output. We'll discuss and use this function when we implement Step 2c.

code reuse

The printf method is an example of **code reuse**, in which a single method can be used by many programs for similar tasks. In addition to Java's many packages of reusable methods, programmers often produce packages of their own Java methods that are intended to be reused over and over with many different application programs.

easily modified code

Decomposing problems also produces a good final program in the sense that the program is easy to understand, and subsequent maintenance and modifications are relatively easy. For example, our temperature program might later be modified to convert to Kelvin degrees instead of Fahrenheit. Since the conversion task is performed by a separate Java method, most of the modification will be confined to this one method. Easily modified code is vital since real-world studies show that a large proportion of programmers' time is spent maintaining and modifying existing programs.

For a problem decomposition to produce easily modified code, the Java methods you write need to be genuinely separated from one another. An analogy can help explain the notion of "genuinely separated." Suppose you are moving a bag of gold coins to a safe hiding place. If the bag is too heavy to carry, you might divide the coins into three smaller bags and carry the bags one by one. Unless you are a character in a comedy, you would not try to carry all three bags at once. That would defeat the purpose of dividing the coins into three groups. This strategy works only if you carry the bags one at a time. Something similar happens in problem decomposition. If you divide your programming task into three subtasks and solve these subtasks by writing three Java methods, you have traded one hard problem for three easier problems. Your total job has become easier—provided that you design the methods separately. When you are working on one method, you should not worry about how the other methods perform their jobs. But the methods do interact. So when you are designing one method, you need to know something about what the other methods do. The trick is to know *only as much as you need but no more*. This is called **information hiding**. One technique for information hiding involves specifying your methods' behavior using *preconditions* and *postconditions*, which we'll discuss next.

How to Write a Specification for a Java Method

When you implement a method in Java, you give complete instructions for how the method performs its computation. However, when you are *using a method* in your pseudocode or writing other Java code, you only need to think about *what the method does*. You need not think about *how the method* does its work. For example, suppose you are writing the temperature-conversion program and you are told that the following method is available for you to use:

```
// Convert a Celsius temperature c to Fahrenheit degrees
public static double celsiusToFahrenheit(double c)
```

signature of a method

This information about the method is called its *signature*. A **signature** includes the method name (`celsiusToFahrenheit`), its parameter list (`double c`), its return type (a `double` number), and any modifiers (`public` and `static`).

In your program, you might have a `double` variable called `celsius` that contains a Celsius temperature. Knowing this description, you can confidently write the following statement to convert the temperature to Fahrenheit degrees, storing the result in a `double` variable called `fahrenheit`:

```
fahrenheit = celsiusToFahrenheit(celsius);
```

When you use the `celsiusToFahrenheit` method, you do not need to know the details of how the method carries out its work. You need to know *what* the method does, but you do not need to know *how* the task is accomplished.

procedural abstraction

When we pretend that we do not know how a method is implemented, we are using a form of information hiding called **procedural abstraction**. This simplifies your reasoning by abstracting away irrelevant details (that is, by hiding them). When programming in Java, it might make more sense to call it "method abstraction" since you are abstracting away irrelevant details about how a method works. However, computer scientists use the term *procedure* for any sequence of instructions, so they also use the term *procedural abstraction*. Procedural abstraction can be a powerful tool. It simplifies your reasoning by allowing you to consider methods one at a time rather than all together.

To make procedural abstraction work for us, we need some techniques for documenting what a method does without indicating how the method works. We could just write a short comment as we did for `celsiusToFahrenheit`. However, the short comment is a bit incomplete; for instance, the comment doesn't indicate what happens if the parameter `c` is smaller than the lowest Celsius temperature (–273.15°C, also called **absolute zero**). For better completeness and consistency, we will follow a fixed format that is guaranteed to provide the same kind of information about any method you may write. The format has five parts, which are illustrated below and on the next page for the `celsiusToFahrenheit` method.

♦ **celsiusToFahrenheit**

`public static double celsiusToFahrenheit(double c)`

Convert a temperature from Celsius degrees to Fahrenheit degrees.

Parameters:
`c` – a temperature in Celsius degrees

Precondition:
`c >= -273.15.`

Returns:
the temperature `c` converted to Fahrenheit degrees

Throws: `IllegalArgumentException`
Indicates that `c` is less than the smallest Celsius temperature (–273.15).

This documentation is called the method's **specification**. Let's look at the five parts of this specification.

1. Short Introduction. The specification's first few lines are a brief introduction. The introduction includes the method's name, the complete heading (`public static double celsiusToFahrenheit(double c)`), and a short description of the action that the method performs.

2. Parameter Description. The specification's second part is a list of the method's parameters. We have one parameter, `c`, which is a temperature in Celsius degrees.

3. Precondition. A **precondition** is a condition that is supposed to be true when a method is called. The method is not guaranteed to work correctly unless the precondition is true. Our method requires that the Celsius temperature `c` be no less than the smallest valid Celsius temperature (–273.15°C).

4. The Returns Condition or Postcondition. A **returns** condition specifies the meaning of a method's return value. We used a returns condition for `celsiusToFahrenheit`, specifying that the method "returns the temperature `c` converted to Fahrenheit degrees." More complex methods may have additional effects beyond a single return value. For example, a method may print values or alter its parameters. To describe such effects, a general *postcondition* can be provided instead of just a returns condition. A **postcondition** is a complete statement describing what will be true when a method finishes. If the precondition was true when the method was called, then the method will complete and the postcondition will be true when the method completes. The connection between a precondition and a postcondition is given here:

A Method's Precondition and Postcondition

A **precondition** is a statement giving the condition that is supposed to be true when a method is called. The method is not guaranteed to perform as it should unless the precondition is true.

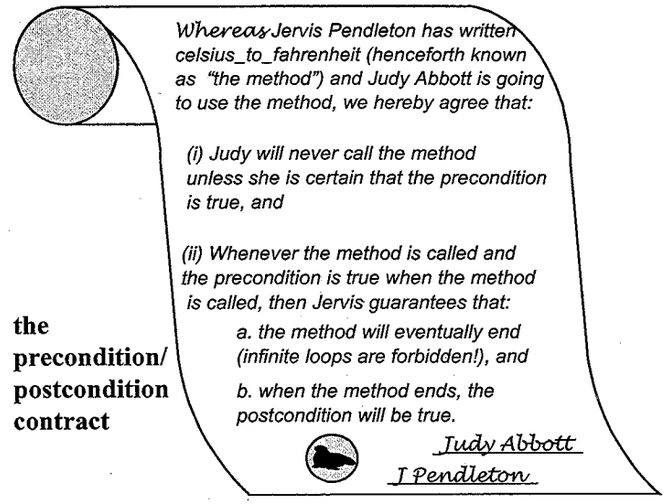
A **postcondition** is a statement describing what will be true when a method call is completed. If the method is correct and the precondition was true when the method was called, then the method will complete, and the postcondition will be true when the method's computation is completed.

For small methods that merely return a calculated value, the specification provides a precondition and a returns condition. For more complex methods, the specification provides a precondition and a general postcondition.

programming teams

Preconditions and postconditions are even more important when a group of programmers work together. In team situations, one programmer often does not know how a function written by another programmer works. In fact, sharing knowledge about how a function works can be counterproductive. Instead, the precondition and postcondition provide all the interaction that's needed. In effect, the precondition/postcondition pair forms a contract between the programmer who uses a function and the programmer who writes that function. To aid the explanation of this "contract," we'll give these two programmers names. Judy is the head of programming team that is writing a large piece of software. Jervis is one of her programmers, who writes various functions for Judy to use in large programs. If Judy and Jervis were lawyers, the contract might look like the scroll shown in the margin. As a programmer, the contract tells them precisely what the function does. It states that if Judy makes sure that the precondition is met when the function is called, then Jervis ensures that the function returns with the postcondition satisfied.

Before long, we will provide both the specification and the implementation of the `celsiusToFahrenheit` method. But keep in mind that it's only the specification that we need to know how to use the method.



the precondition/ postcondition contract

5. The "throws" List. It is always the responsibility of the programmer who *uses* a method to ensure that the precondition is valid. Calling a method without ensuring a valid precondition is a programming error. Once the precondition fails, the method's behavior is unpredictable—the method *could* do anything at all. Nonetheless, the person who writes a method should make every effort to avoid the more unpleasant behaviors, even if the method is called incorrectly. As part of this effort, the first action of a method is often to check that its precondition has been satisfied. If the precondition fails, then the method *throws an exception*. You may have used exceptions in your previous programming, or maybe not. In either case, the next programming tip describes the exact meaning of *throwing an exception* to indicate that a precondition has failed.

PROGRAMMING TIP

THROW AN EXCEPTION TO INDICATE A FAILED PRECONDITION

It is a programming error to call a method when the precondition is invalid. For example, `celsiusToFahrenheit` should not be called with an argument that is below `-273.15`. Despite this warning, some programmer may try `celsiusToFahrenheit(-1000)` or `celsiusToFahrenheit(-273.17)`. In such a case, our `celsiusToFahrenheit` method will detect that the precondition has been violated, immediately halt its own work, and pass a "message" to the calling program to indicate that an illegal argument has occurred. Such messages for serious programming errors are called **exceptions**. The act of halting your own work and passing a message to the calling program is known as **throwing an exception**.

how to throw an exception

The Java syntax for throwing an exception is simple. You begin with the keyword `throw` and follow this pattern:

```
throw new ____________ ("____________");
```

This is the type of exception we are throwing. To begin with, all of our exceptions will be the type IllegalArgumentException, which is provided as part of the Java language. This type of exception tells a programmer that one of the method's arguments violated a precondition.

This is an error message that will be passed as part of the exception. The message should describe the error in a way that will help the programmer fix the programming error.

what happens when an exception is thrown?

When an exception is thrown in a method, the method stops its computation. A new "exception object" is created, incorporating the indicated error message. The exception, along with its error message, is passed up to the method or program that made the illegal call in the first place. At that point, where the illegal call was made, there is a Java mechanism to "catch" the exception, try to fix the error, and continue with the program's computation. You can read about exception catching in Appendix C. However, exceptions that arise from precondition violations should never be caught because they indicate programming errors that must be fixed. When an exception is not caught, the program halts, printing the error message along with a list of the method calls that led to the exception. This error message can help the programmer fix the programming error.

Now you know the meaning of the specification's "throws list." It is a list of all the exceptions that the method can throw, along with a description of what causes each exception. Certain kinds of exceptions must also be listed in a method's implementation, after the parameter list, but an `IllegalArgumentException` is listed only in the method's specification.

Temperature Conversion: Implementation

Our specification and design are now in place. The subtasks are small enough to implement, though during the implementation you may need to finish small design tasks such as finding the conversion formula. In particular, we can now implement the temperature program as a Java application program with two methods:

- a `main` method that follows the pseudocode from page 5: This `main` method prints the temperature conversion table using the method `celsiusToFahrenheit` to carry out some of its work.
- the `celsiusToFahrenheit` method.

The Java application program with these two methods appears in Figure 1.1. The program produces the output from our initial specification on page 4.

A few features of the implementation may be new to you, so we will discuss these in some programming tips and after the figure.

FIGURE 1.1 Specification and Implementation for the Temperature Conversion Application

Class TemperatureConversion

❖ **public class TemperatureConversion**

The `TemperatureConversion` Java application prints a table converting Celsius to Fahrenheit degrees.

Specification

♦ **main**

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

The `main` method prints a Celsius-to-Fahrenheit conversion table. The `String` arguments (`args`) are not used in this implementation. The bounds of the table range from –50C to +50C in 10-degree increments.

♦ **celsiusToFahrenheit**

`public static double celsiusToFahrenheit(double c)`

Convert a temperature from Celsius degrees to Fahrenheit degrees.

Parameters:

`c` – a temperature in Celsius degrees

Precondition:

`c >= -273.15`.

Returns:

the temperature `c` converted to Fahrenheit degrees

Throws: `IllegalArgumentException`

Indicates that `c` is less than the smallest Celsius temperature (–273.15).

These specifications were automatically produced by the Javadoc tool. Appendix H describes how to use Javadoc to produce similar information.

(continued)

(FIGURE 1.1 continued)

Java Application Program

The actual implementation includes Javadoc comments that are omitted here but are listed in Appendix H. The comments allow Javadoc to produce nicely formatted information automatically. See Appendix H to read about using Javadoc for your programs.

```
// File: TemperatureConversion.java from
// www.cs.colorado.edu/~main/applications/
// A Java application to print a
// temperature conversion table.
// Additional Javadoc information is available on page 11 or at
//   http://www.cs.colorado.edu/~main/docs/TemperatureConversion.html

public class TemperatureConversion
{
   public static void main(String[ ] args)
   {
      // Declare values that control the table's bounds.
      final double TABLE_BEGIN = -50.0; // The table's first Celsius temperature
      final double TABLE_END   =  50.0; // The table's final Celsius temperature
      final double TABLE_STEP  =  10.0; // Increment between temperatures in table

      double celsius;                   // A Celsius temperature
      double fahrenheit;                // The equivalent Fahrenheit temperature

      System.out.println("TEMPERATURE CONVERSION");
      System.out.println("----------------------");
      System.out.println("Celsius     Fahrenheit");
      for (celsius = TABLE_BEGIN; celsius <= TABLE_END; celsius += TABLE_STEP)
      {  // The for-loop has set celsius equal to the next Celsius temperature of the table.
         fahrenheit = celsiusToFahrenheit(celsius);
         System.out.printf("%6.2fC", celsius);
         System.out.printf("%14.2fF\n", fahrenheit);
      }
      System.out.println("----------------------");
   }

   public static double celsiusToFahrenheit(double c)
   {
      final double MINIMUM_CELSIUS = -273.15;
      if (c < MINIMUM_CELSIUS)
        throw new IllegalArgumentException("Argument " + c + " is too small.");
      return (9.0/5.0) * c + 32;
   }
}
```


PROGRAMMING TIP

USE JAVADOC TO WRITE SPECIFICATIONS

the Javadoc tool automatically produces nicely formatted information about a class

The specification at the start of Figure 1.1 was produced in an interesting way. Most of it was automatically produced from the Java code by a tool called **Javadoc**. With a web browser, you can access this specification over the Internet at: `http://www.cs.colorado.edu/~main/docs/TemperatureConversion.html`.

To use Javadoc, the .java file that you write needs special **Javadoc comments**. These necessary comments are not listed in Figure 1.1, but Appendix H is a short manual on how to write these comments. Before you go on to Chapter 2, you should read through the appendix and be prepared to produce Javadoc comments for your own programs.

PROGRAMMING TIP

USE FINAL VARIABLES TO IMPROVE CLARITY

The method implementation in Figure 1.1 has a local variable declared this way:

```
final double MINIMUM_CELSIUS = -273.15;
```

This is a declaration of a `double` variable called `MINIMUM_CELSIUS`, which is given an initial value of −273.15. The keyword `final`, appearing before the declaration, makes `MINIMUM_CELSIUS` more than just an ordinary declaration. It is a **final variable**, which means that its value will never be changed while the program is running. A common programming style is to use all capital letters for the names of final variables. This makes it easy to determine which variables are final and which may have their values changed.

There are several advantages to defining `MINIMUM_CELSIUS` as a final variable rather than using the constant −273.15 directly in the program. Using the name `MINIMUM_CELSIUS` makes the comparison `(c < MINIMUM_CELSIUS)` easy to understand; it's clear that we are testing whether `c` is below the minimum valid Celsius temperature. If we used the direct comparison `(c < −273.15)` instead, a person reading our program would have to stop to remember that −273.15 is "absolute zero," the smallest Celsius temperature.

To increase clarity, some programmers declare all constants as final variables. This is a good rule, particularly if the same constant appears in several different places in the program or if you plan to later compile the program with a different value for the constant. However, well-known formulas may be more easily recognized in their original form. For example, the conversion from Celsius to Fahrenheit is recognizable as $F = \frac{9}{5}C + 32$. Thus, Figure 1.1 uses this statement:

```
return (9.0/5.0) * c + 32;
```

This return statement is clearer and less error prone than a version that uses final variables for the constants $\frac{9}{5}$ and 32.

Advice: Use final variables instead of constants. Make exceptions when constants are clearer or less error prone. When in doubt, write both solutions and interview several colleagues to decide which is clearer.

Programming Tip

Make Exception Messages Informative

If the celsiusToFahrenheit method detects that its parameter, `c`, is too small, then it throws an `IllegalArgumentException`. The message in the exception is:

```
"Argument "+ c + " is too small."
```

The parameter, `c`, is part of the message. If a programmer attempts to call `celsiusToFahrenheit(-300)`, the message will be "Argument -300 is too small."

Programming Tip

Format Output with System.out.printf

Our pseudocode for the temperature problem includes a step to print the Celsius and Fahrenheit temperatures rounded to a specified accuracy. Java 1.5 added a new method, `System.out.printf`, that we can use to print formatted output such as rounded numbers. The method, which is based on a similar function in the C language, always has a "format string" as its first argument. The format string tells how subsequent arguments should be printed to `System.out` (which is usually the computer monitor). For example, you could create an appropriate format string and print two numbers called age and height:

```
System.out.println(format string, age, height);
```

Two things can appear in a format string:

1. Most ordinary characters are just printed as they appear in the format string. For example, if the character F appears in the format string, then this will usually just print an F. Some sequences of characters are special, such as \n (which moves down to start the next new line). The complete list of special characters is listed in Appendix B.
2. Parts of the format string that begin with the % character are called **format specifiers**. Each format specifier indicates how the next item should be printed. Some of the common format specifiers are shown here:

Format specifier	Causes the next item to be printed as...
%d	...a decimal (base 10) whole number
%f	...a floating point number with six digits after the decimal point
%g	...a floating point number using scientific notation for large exponents
%s	...a string

The format specifiers have several additional options that are described in Appendix B. For our program, we used a format specifier of the form `%6.2f`, which means that we want to print a floating point number using six total spaces, two of which are after the decimal point. Within these six spaces, the number will be rounded to two decimal points and right justified. For example, the number `42.129` will print these six characters (the first of which is a blank space):

	4	2	.	1	3

In the format specifier `%6.2f`, the number 6 is the ***field width***, and the number 2 is the ***precision***.

As an example, the call `System.out.println("%14.2fF\n", fahrenheit)` prints the value of the fahrenheit variable using a field width of 14 and a precision of 2. After printing this number, the character F and a new line will appear.

Self-Test Exercises for Section 1.1

Each section of this book finishes with a few self-test exercises. Answers to these exercises are given at the end of each chapter. The first exercise refers to a method that Judy has written for *you* to use. Here is the specification:

♦ **dateCheck**

`public static int dateCheck(int year, int month, int day)`

Compute how long until a given date will occur.

Parameters:

`year` – the year for a given date
`month` – the month for a given date (using 1 for Jan, 2 for Feb, etc.)
`day` – the day of the month for the given date

Precondition:

The three arguments are a legal year, month, and day of the month in the years 1900 to 2099.

Returns:

If the given date has been reached on or before today, the return value is zero. Otherwise, the return value is the number of days until the given date returns.

Throws: `IllegalArgumentException`

Indicates the arguments are not a legal date in the years 1900 to 2099.

1. Suppose you call the method `dateCheck(2003, 7, 29)`. What is the return value if today is July 22, 2003? What if today is July 30, 2003? What about February 1, 2004?
2. Can you use `dateCheck` even if you don't know how it is implemented?
3. Suppose that `boodle` is a `double` variable. Write two statements that will print `boodle` to `System.out` in the following format: `$ 42,567.19` The whole dollars part of the output can contain up to nine characters, so this example has three spaces before the six characters of `42,567`. The fractional part is rounded to the nearest hundredth.

4. Consider a method with this heading:

   ```
   public static void printSqrt(double x)
   ```

   The method prints the square root of x to the standard output. Write a reasonable specification for this method and compare your answer to the solution at the end of the chapter. The specification should forbid a negative argument.

5. Consider a method with this heading:

   ```
   public static double sphereVolume(double radius)
   ```

   The method computes and returns the volume of a sphere with the given radius. Write a reasonable specification for this method and compare your answer to the solution at the end of the chapter. The specification should require a positive radius.

6. Write an if-statement to throw an `IllegalArgumentException` when x is less than zero. (Assume that x is a `double` variable.)
7. When can a `final` variable be used?
8. How was the specification produced at the start of Figure 1.1?
9. What are the components of a Java signature?
10. Write a Java statement that will print two variables called age and height. The age should be printed as a whole number using ten output spaces; the height should be printed using twelve output spaces and three decimal digits. Label each part of the output as "Age" and "Height." Use Appendix B if necessary.

1.2 RUNNING TIME ANALYSIS

Time analysis consists of reasoning about an algorithm's speed. *Does the algorithm work fast enough for my needs? How much longer does the algorithm take when the input gets larger? Which of several different algorithms is fastest?* These questions can be asked at any stage of software development. Some time analysis is useful to analyze an algorithm before any implementation is done to avoid wasted work of implementing inappropriately slow solutions. Further analysis can be carried out during or after an implementation. This section discusses time analysis, starting with an example that involves no implementation in the usual sense.

The Stair-Counting Problem

Suppose you and your friend Judy are standing at the top of the Eiffel Tower. As you gaze out over the French landscape, Judy turns to you and says, "I wonder how many steps there are to the bottom?" You, of course, are the ever-accommodating host, so you reply, "I'm not sure...but I'll find out." We'll look at three techniques that you could use and analyze the time requirements of each.

Technique 1: Walk Down and Keep a Tally. In the first technique, Judy gives you a pen and a sheet of paper. "I'll be back in a minute," you say as you dash down the stairs. Each time you take a step down, you make a mark on the sheet of paper. When you reach the bottom, you run back up, show Judy the piece of paper, and say, "There are this many steps."

Technique 2: Walk Down, but Let Judy Keep the Tally. In the second technique, Judy is unwilling to let her pen or paper out of her sight. But you are undaunted. Once more you say, "I'll be back in a minute," and you set off down the stairs. But this time you stop after one step, lay your hat on the step, and run back to Judy. "Make a mark on the piece of paper!" you exclaim. Then you run back to your hat, pick it up, take one more step, and lay the hat down on the second step. Then back up to Judy: "Make another mark on the piece of paper!" you say. You run back down the two stairs, pick up your hat, move to the third step, and lay down the hat. Then back up the stairs to Judy: "Make another mark!" you tell her. This continues until your hat reaches the bottom, and you speed back up the steps one more time. "One more mark, please." At this point, you grab Judy's piece of paper and say, "There are this many steps."

Technique 3: Jervis to the Rescue. In the third technique, you don't walk down the stairs at all. Instead, you spot your friend Jervis by the staircase, holding the sign shown here. The translation is *There are 2689 steps in this stairway (really!)*. So you take the paper and pen from Judy, write the number 2689, and hand the paper back to her, saying, "There are this many steps."

This is a silly example, but even so, it does illustrate the issues that arise when performing a time analysis for an algorithm or program. The first issue is deciding exactly how you will measure the time spent carrying out the work or executing the program. At first glance, the answer seems easy: For each of the three stair-counting techniques, just measure the actual time it takes to carry out the work. You could do this with a stopwatch. But there are some drawbacks to measuring actual time. Actual time can depend on various irrelevant details such as whether you or somebody else carried out the work. The actual elapsed time may vary from person to person, depending on how fast each person can run the stairs. Even if we decide that *you* are the runner, the time may vary depending on other factors such as the weather, what you had for breakfast, and what other things are on your mind.

So, instead of measuring the actual elapsed time, we count certain operations that occur while carrying out the work. In this example, we will count two kinds of operations:

1. Each time you walk up or down one step, that is one operation.
2. Each time you or Judy marks a symbol on the paper, that is also one operation.