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 in all software, including the small programs that 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.

Years from now you may be a software engineer writing large systems in a specialized area, perhaps computer graphics or artificial intelligence. Such futuristic applications will be exciting and stimulating, and within your work you will still see the phases of software development and fundamental data structures that you learn and practice now.

Here is a list of the phases of software development:

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

Do not 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 little 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.

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 where 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 to the Java language that is particularly suited to applications that can be moved from one machine to another over the Internet.

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

Throughout the evolution of Java, the designers 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.

There are many different Java development environments that you may successfully use 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.

you should already know how to write, compile, and run short Java programs

If you know how to use Java in these ways, then 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

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 to solve the problem; the **implementation** is the actual Java code to carry out the design.

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

```
TEMPERATURE CONVERSION
----------------------
Celsius     Fahrenheit
-50.00C
-40.00C
-30.00C
-20.00C
-10.00C
  0.00C
 10.00C
 20.00C
 30.00C
 40.00C
 50.00C
----------------------
```

The equivalent Fahrenheit temperatures will be computed and displayed on this side of the table.

An **algorithm** is a set of instructions for solving a problem. An algorithm for the temperature problem will print the conversion table. During the design of the algorithm, the details of a particular programming language can be distracting and obscure the simplicity of a solution. Therefore, during the design we generally write in English. We use a rather corrupted kind of English that mixes in Java when it's convenient. This mixture of English and a programming language is called **pseudocode**. When the Java code for a step is obvious, then the pseudocode may use Java. When a step is clearer in English, then we will use English. Keep in mind that the reason for pseudocode is to improve *clarity*.

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 discuss now.

Key Design Technique

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

Design Technique: Decomposing the Problem

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: (1) converting a temperature from Celsius degrees to Fahrenheit, and (2) 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 `celsius`, rounding to the nearest hundredth.
   2d. Print the letter C and some spaces.
   2e. Print `fahrenheit`, rounding to the nearest hundredth.
   2f. Print the letter F, and end the output line.
3. Print the line of dashes at the bottom of the table.

what makes a good decomposition?

The underlined steps (2b, 2c, and 2e) are the major subtasks that we have identified. 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.

code reuse

Steps 2c and 2e both print a number with a specified accuracy, so we plan to write just a single Java method that "prints a number" and use this method for both steps. This is an example of **code reuse**, in which we use a single method for several related tasks. In fact, programmers often produce collections of

related Java methods that are made available in packages to be reused over and over with many different application programs. Later we will write such a package of methods for producing nice output, including the "print a number" method. But for now, the nice package is unavailable, so we plan to write the printing method from scratch.

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.

In order for a problem decomposition to produce easily modified code, the Java methods that 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, then 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 incorporating information hiding involves specifying your methods' behavior using *preconditions* and *postconditions*, which we 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)
```

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 the irrelevant details. 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, the term *procedure* is a more general term than *method.* Computer scientists use the term procedure for any sequence of instructions, and so they 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. Of course, 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 (–273.16°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 that you may write. The format has five parts, which are illustrated here 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 >= -273.16`.

  Returns:
  the temperature `c` converted to Fahrenheit degrees

  Throws: `IllegalArgumentException`
  Indicates that `c` is less than the smallest Celsius temperature (–273.16).

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` is no less than the smallest valid Celsius temperature (–273.16°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 can provide a precondition and a returns condition. For more complex methods, the specfication can provide a precondition and a general postcondition.

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 section describes the exact meaning of *throwing an exception* to indicate that a precondition has failed.

Throwing an Exception to Indicate a Failed Precondition

It is a programming error to call `celsiusToFahrenheit` with an argument that is below –273.16. Celsius temperatures below this level have no physical meaning, they cannot be converted to Fahrenheit, and the method cannot provide a meaningful return value. Despite this warning, some chilly 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. For example, the implementation of `celsiusToFahrenheit` might have these statements:

```
if (c < -273.16)
   throw new IllegalArgumentException("Temperature too small.");
```

Simple exceptions such as this are easy to throw. You begin with the keyword "`throw`" and follow this pattern:

```
throw new ____________________ ("____________________");
```

This is the type of the exception that 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 within a method, the method immediately 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.

With a method's specification in place, you can think more about the design. Perhaps the method needs to be broken into yet smaller tasks, or perhaps the task is now small enough to start the implementation—the actual writing of Java code. The `celsiusToFahrenheit` problem is small enough to implement, though during the implementation you may need to finish small design tasks such as finding the conversion formula (a celsius temperature C is converted to a Fahrenheit temperature F with the formula $F = \frac{9}{5}C + 32$). The complete implementation might be something like Figure 1.1 on page 10. The implementation illustrates two programming tips that we'll discuss now.

Programming Tip: How and When to Throw Exceptions

TIP 

It is a programming error to call a method when the precondition is invalid. When a method detects an invalid precondition, then the method should throw an `IllegalArgumentException`. The exception usually includes a copy of the argument as part of its message. For example, the exception message in Figure 1.1 is:

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

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

FIGURE 1.1 Specification and Implementation of the `celsiusToFahrenheit` Method

Specification

- **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 >= -273.16`.

  Returns:
  the temperature `c` converted to Fahrenheit degrees

  Throws: `IllegalArgumentException`
  Indicates that `c` is less than the smallest Celsius temperature (–273.16).

Implementation

```
public static double celsiusToFahrenheit(double c)
{
   final double MINIMUM_CELSIUS = -273.16;

   if (c < MINIMUM_CELSIUS)
      throw new IllegalArgumentException("Argument " + c + " is too small.");
   return (9.0/5.0) * c + 32;
}
```

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 = -273.16;
```

This is a declaration of a `double` variable called `MINIMUM_CELSIUS`, which is given an initial value of –273.16. 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 –273.16 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 < –273.16)` instead, then a person reading our program would have to stop to remember that –273.16 is "absolute zero," the smallest Celsius temperature.

To increase clarity, some programmers declare *all* constants as final variables. As rules go, this is a reasonable rule, particularly if the same constant appears in several different places of the program or if you plan to compile the program sometime with a different value for the constant. However, there is another side to the issue. Well-known formulas may be more easily recognized in their original form (using constants rather than artificially introduced names). For example, the conversion from Celsius to Fahrenheit is recognizable as $F = \frac{9}{5}C + 32$. Thus, Figure 1.1 uses the return statement shown here:

```
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.

The Method for Printing a Number

Our pseudocode for the temperature problem includes a step to print a number rounded to a specified accuracy. Here is a specification for a method that we can use to carry out this step:

- **printNumber**

  `public static void printNumber(double d, int minimumWidth, int fractionDigits)`

  Print a number to `System.out`, using a specified format.

  Parameters:

  `d` – the number to be printed

  `minimumWidth` – the minimum number of characters in the entire output

  `fractionDigits` – the number of digits to print on the right side of the decimal point

  Precondition:

  `fractionDigits` is not negative.

  Postcondition:

  The number `d` has been printed to `System.out`. This printed number is rounded to the specified number of digits on the right of the decimal. If `fractionDigits` is 0, then only the integer part of `d` is printed. If necessary, spaces appear at the front of the number to raise the total number of printed characters to the minimum. Additional formatting details are obtained from the current locale. For example, in the United States, a period is used for the decimal and commas are used to separate groups of integer digits.

  Throws: `IllegalArgumentException`

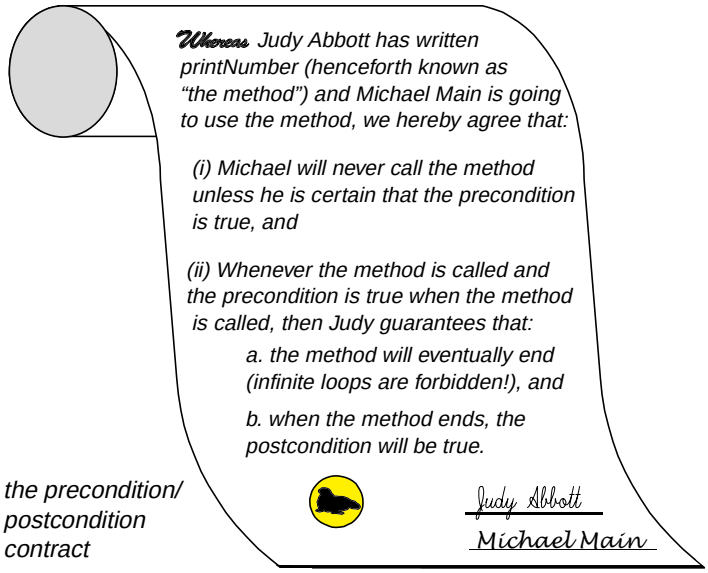
  Indicates that `fractionDigits` is negative.

  Example:

  ```
  printNumber(12345.27, 8, 1); // Prints 12,345.3 in the U.S.
  ```

Because `printNumber` is somewhat complex, we have added an example to the specification. An example helps a programmer quickly understand the intended use of a method.

With the specification in place, you can turn to the implementation—or if you are working in a programming team, you might ask another group member to implement the method. In fact, I asked several of my students to implement `printNumber`, using the specification shown earlier. The students researched how to round numbers and how to determine other formatting details. I decided to use one of the implementations, from a student named Judy Abbott. Her implementation of `printNumber` was interesting because it used several unfamiliar features of the Java language—unfamiliar to me anyway. Does this mean that I can't use the method? Of course not. The specification tells me how to use the method, even if I don't know the implementation details. In team situations, one programmer often does not know the implementation details of another programmer's work. In fact, sharing knowledge about how methods work can be counterproductive. Instead, the combination of 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 method and the programmer who writes that method. If programmers were lawyers, the contract might look like the scroll shown in the margin. As a programmer, the contract tells me precisely what Judy's method does. It states that *if* I make sure that the precondition is met when the method is called, *then* Judy ensures that the method returns with the postcondition satisfied.



the precondition/ postcondition contract

If you are curious about the `printNumber` programming, you can read Appendix B, which provides some input/output ideas for Java programming. But even without the knowledge of how Judy writes `printNumber`, we can write a program that uses Judy's method. In particular, we can now implement the temperature program as a Java application program with three methods:

- a `main` method that follows the pseudocode from page 4: This `main` method prints the temperature conversion table using `printNumber` and `celsiusToFahrenheit` to carry out the work of its subtasks.
- the `celsiusToFahrenheit` method.
- the `printNumber` method.

The Java application program with these three methods appears in Figure 1.2 on page 14, together with the specifications that we already wrote.

the Javadoc tool automatically produces nicely formatted information about a class

The specification at the start of Figure 1.2 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 actual implementation needs special **Javadoc comments**. These necessary comments are not listed in Figure 1.2, but the techniques for writing these comments are described in Appendix H. You should consider using Javadoc to produce similar specifications for your programs.

Self-Test Exercises

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, then 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. You may use the `printNumber` method from this section.

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 forbid a negative 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 1.2?

FIGURE 1.2 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.

(continued)

(FIGURE 1.2 continued)

- **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 >= -273.16`.

Returns:
the temperature `c` converted to Fahrenheit degrees

Throws: `IllegalArgumentException`
Indicates that `c` is less than the smallest Celsius temperature (–273.16).

Most of this information was automatically produced by the Javadoc tool. Appendix H describes how to use Javadoc to produce similar information for your programs.

- **printNumber**

```
public static void printNumber(double d, int minimumWidth, int fractionDigits)
```

Print a number to `System.out`, using a specified format.

Parameters:
`d` – the number to be printed
`minimumWidth` – the minimum number of characters in the entire output
`fractionDigits` – the number of digits to print on the right side of the decimal point

Precondition:
`fractionDigits` is not negative.

Postcondition:
The number `d` has been printed to `System.out`. This printed number is rounded to the specified number of digits on the right of the decimal. If `fractionDigits` is 0, then only the integer part of `d` is printed. If necessary, spaces appear at the front of the number to raise the total number of printed characters to the minimum. Additional formatting details are obtained from the current locale. For example, in the United States, a period is used for the decimal, and commas are used to separate groups of integer digits.

Throws: `IllegalArgumentException`
Indicates that `fractionDigits` is negative.

Example:

```
printNumber(12345.27, 8, 1); // Prints 12,345.3 in the U.S.
```

(continued)

(FIGURE 1.2 continued)

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.

Java Application Program

```
// File: TemperatureConversion.java
// A Java application to print a
// temperature conversion table.
// Additional Javadoc information is available on pages 14-15 or at
//   http://www.cs.colorado.edu/~main/docs/TemperatureConversion.html

import java.text.NumberFormat; // Used in the printNumber method.

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
      final int    WIDTH       =     6; // Number of characters in output numbers
      final int    ACCURACY    =     2; // Number of digits to right of decimal point

      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);
         printNumber(celsius, WIDTH, ACCURACY);
         System.out.print("C        ");
         printNumber(fahrenheit, WIDTH, ACCURACY);
         System.out.println("F");
      }
      System.out.println("----------------------");
   }

   public static double celciusToFahrenheit(double c)
   {
      final double MINIMUM_CELSIUS = -273.16;
      if (c < MINIMUM_CELSIUS)
        throw new IllegalArgumentException("Argument " + c + " is too small.");
      return (9.0/5.0) * c + 32;
   }
```

(continued)

(FIGURE 1.2 continued)

```
    public static void printNumber(double d, int minimumWidth, int fractionDigits)
    {
        // Note: getNumberInstance( ) creates a NumberFormat object using local
        // information about the characters for a decimal point and separators.
        NumberFormat form = NumberFormat.getNumberInstance( );
        String output;
        int i;
        int length;

        // Set the number of digits to appear on the right of the decimal.
        if (fractionDigits < 0)
           throw new IllegalArgumentException("fractionDigits < 0:" + fractionDigits);
        form.setMinimumFractionDigits(fractionDigits);
        form.setMaximumFractionDigits(fractionDigits);

        // Round and format the number. I have added code to handle a Java bug that
        // occurs when fractionDigits is zero.
        output = form.format(d);
        length = output.length( );
        if (fractionDigits == 0 && length > 1 && output.charAt(length-2) == '.')
           output = "  " + output.substring(0,length-2);

        // Print any leading spaces and the number itself.
        for (i = output.length( ); i < minimumWidth; i++)
           System.out.print(' ');
        System.out.print(output);
    }
}
```

Output of the Application

```
TEMPERATURE CONVERSION
----------------------
Celsius     Fahrenheit
-50.00C        -58.00F
-40.00C        -40.00F
-30.00C        -22.00F
-20.00C         -4.00F
-10.00C         14.00F
  0.00C         32.00F
 10.00C         50.00F
 20.00C         68.00F
 30.00C         86.00F
 40.00C        104.00F
 50.00C        122.00F
----------------------
```

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?* We'll discuss these issues in this section. An example will start the discussion.

The Stair-Counting Problem

Suppose that 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 drawn 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.

decide what operations to count

Of course, each of these operations takes a certain amount of time, and making a mark may take a different amount of time than taking a step. But this doesn't concern us because we won't measure the actual time taken by the operations. Instead, we will ask: *How many operations are needed for each of the three techniques?*

For the first technique, you take 2689 steps down, another 2689 steps up, and you also make 2689 marks on the paper, for a total of 3×2689 operations—that is 8067 total operations.

For the second technique, there are also 2689 marks made on Judy's paper, but the total number of operations is considerably more. You start by going down one step and back up one step. Then down two and up two. Then down three and up three, and so forth. The total number of operations taken is:

Downward steps	=	3,616,705 (which is $1 + 2 + \ldots + 2689$)
Upward steps	=	3,616,705
Marks made	=	2689
Total operations	=	Downward steps + Upward steps + Marks made
	=	7,236,099

The third technique is the quickest of all: Only four marks are made on the paper (that is, we're counting one "mark" for each digit of 2689), and there is no

going up and down stairs. The number of operations used by each of the techniques is summarized here:

Technique 1	8067	operations
Technique 2	7,236,099	operations
Technique 3	4	operations

Doing a time analysis for a program is similar to the analysis of the stair-counting techniques. For a time analysis of a program, we do not usually measure the actual time taken to run the program because the number of seconds can depend on too many extraneous factors—such as the speed of the processor, and whether the processor is busy with other tasks. Instead, the analysis counts the number of operations required. There is no precise definition of what constitutes an **operation**, although an operation should satisfy your intuition of a "small step." An operation can be as simple as the execution of a single program statement. Or we could use a finer notion of operation that counts each arithmetic operation (addition, multiplication, etc.) and each assignment to a variable as a separate operation.

dependence on input size

For most programs, the number of operations depends on the program's input. For example, a program that sorts a list of numbers is quicker with a short list than with a long list. In the stairway example, we can view the Eiffel Tower as the input to the problem. In other words, the three different techniques all work on the Eiffel Tower, but the techniques also work on Toronto's CN Tower or on the stairway to the top of the Statue of Liberty or on any other stairway.

When a time analysis depends on the size of the input, then the time can be given as an expression, where part of the expression is the input's size. The time expressions for our three stair-counting techniques are:

Technique 1	$3n$
Technique 2	$n + 2\,(1 + 2 + \ldots + n)$
Technique 3	The number of digits in the number n

The expressions on the right give the number of operations performed by each technique when the stairway has n steps.

The expression for the second technique is not easy to interpret. It needs to be simplified in order to become a formula that we can easily compare to other formulas. So, let's simplify it. We start with the subexpression

$$(1 + 2 + \ldots + n)$$

simplification of the time analysis for Technique 2

There is a trick that will enable us to find a simplified form for this expression. The trick is to *compute twice the amount of the expression and then divide the result by 2*. Unless you've seen this trick before, it sounds crazy. But it works

FIGURE 1.3 Deriving a Handy Formula

$(1 + 2 + \ldots + n)$ can be computed by first computing the sum of twice $(1 + 2 + \ldots + n)$, as shown here:

$$\begin{array}{rcccccccc} & 1 & + & 2 & + \ldots + & (n-1) & + & n \\ + & n & + & (n-1) & + \ldots + & 2 & + & 1 \\ \hline & (n+1) & + & (n+1) & + \ldots + & (n+1) & + & (n+1) \end{array}$$

The sum is $n(n + 1)$, so $(1 + 2 + \ldots + n)$ is half this amount:

$$(1 + 2 + \ldots + n) = \frac{n(n+1)}{2}$$

fine. The trick is illustrated in Figure 1.3. Let's go through the computation of that figure step-by-step.

We write the expression $(1 + 2 + \ldots + n)$ twice and add the two expressions. But as you can see in Figure 1.3, we also use another trick: When we write the expression twice, *we write the second expression backwards.* After we write down the expression twice, we see the following:

$$\begin{array}{l} (1 + 2 + \ldots + n) \\ +(n + \ldots + 2 + 1) \end{array}$$

We want the sum of the numbers on these two lines. That will give us twice the value of $(1 + 2 + \ldots + n)$, and we can then divide by 2 to get the correct value of the subexpression $(1 + 2 + \ldots + n)$.

Now, rather than proceed in the most obvious way, we instead add pairs of numbers from the first and second lines. We add the 1 and the n to get $n + 1$. Then we add the 2 and the $n - 1$ to again get $n + 1$. We continue until we reach the last pair consisting of an n from the top line and a 1 from the bottom line. All the pairs add up to the same amount, namely $n + 1$. Now that is handy! We get n numbers, and all the numbers are the same, namely $n + 1$. So the total of all the numbers on the preceding two lines is

$$n(n + 1)$$

The value of twice the expression is n multiplied by $n + 1$. We are now essentially done. The number we computed is twice the quantity we want. So, to

obtain our simplified formula, we only need to divide by 2. The final simplification is thus

$$(1 + 2 + \ldots + n) = \frac{n(n+1)}{2}$$

We will use this simplified formula to rewrite the Technique 2 time expression, but you'll also find that the formula occurs in many other situations. The simplification for the Technique 2 expression is as follows:

Number of operations for Technique 2

$= n + 2(1 + 2 + \ldots + n)$

$= n + 2\left(\frac{n(n+1)}{2}\right)$ *Plug in the formula for* $(1 + 2 + \ldots + n)$

$= n + n(n+1)$ *Cancel the* 2*s*

$= n + n^2 + n$ *Multiply out*

$= n^2 + 2n$ *Combine terms*

So, Technique 2 requires $n^2 + 2n$ operations.

simplification of time analysis for Technique 3

The number of operations for Technique 3 is just the number of digits in the integer n when written down in the usual way. The usual way of writing down numbers is called **base 10 notation**. As it turns out, the number of digits in a number n, when written in base 10 notation, is approximately equal to another mathematical quantity known as the **base 10 logarithm** of n. The notation for the base 10 logarithm of n is written:

$\log_{10} n$

base 10 notation and base 10 logarithms

The base 10 logarithm does not always give a whole number. For example, the actual base 10 logarithm of 2689 is about 3.43 rather than 4. If we want the actual number of digits in an integer n, we need to carry out some rounding. In particular, the exact number of digits in a positive integer n is obtained by rounding $\log_{10} n$ downward to the next whole number, and then adding 1. The notation for rounding down and adding 1 is obtained by adding some marks to the logarithm notation as follows:

$\lfloor \log_{10} n \rfloor + 1$

This is all fine if you already know about logarithms, but what if some of this is new to you? For now, you can simply define the notation to mean *the number of*

digits in the base 10 numeral for n. You can do this because if others use any of the other accepted definitions for this formula, they will get the same answers that you do. You will be right! (And they will also be right.) In Section 10.3 of this book, we will show that the various definitions of the logarithm function are all equivalent. For now, we will not worry about all that detail. We have larger issues to discuss first. The table of the number of operations for each technique can now be expressed more concisely as shown here:

Technique 1	$3n$
Technique 2	$n^2 + 2n$
Technique 3	$\lfloor \log_{10} n \rfloor + 1$

Big-*O* Notation

The time analyses we gave for the three stair-counting techniques were very precise. They computed the exact number of operations for each technique. But such precision is sometimes not needed. Often it is enough to know in a rough manner how the number of operations is affected by the input size. In the stair example, we started by thinking about a particular tower, the Eiffel Tower, with a particular number of steps. We expressed our formulas for the operations in terms of *n*, which stood for the number of steps in the tower. Now suppose that we apply our various stair-counting techniques to a tower with ten times as many steps as the Eiffel Tower. If *n* is the number of steps in the Eiffel Tower, then this taller tower will have $10n$ steps. The number of operations needed for Technique 1 on the taller tower increases tenfold (from $3n$ to $3 \times (10n) = 30n$); the time for Technique 2 increases approximately 100-fold (from about n^2 to about $(10n)^2 = 100n^2$); and Technique 3 increases by only one operation (from the number of digits in *n* to the number of digits in $10n$, or to be very concrete, from the 4 digits in 2689 to the 5 digits in 26,890). We can express this kind of information in a format called **big-*O* notation**. The symbol *O* in this notation is the letter O, so big-*O* is pronounced "big Oh."

We will describe three common examples of the big-*O* notation. In these examples, we use the notion of "the largest term in a formula." Intuitively, this is the term with the largest exponent on *n*, or the term that grows the fastest as *n* itself becomes larger. For now, this intuitive notion of "largest term" is enough. Here are the examples:

quadratic time
$O(n^2)$

Quadratic Time. If the largest term in a formula is no more than a constant times n^2, then the algorithm is said to be "**big-*O* of** n^2," written $O(n^2)$, and the algorithm is called **quadratic**. In a quadratic algorithm, doubling the input size makes the number of operations increase by approximately fourfold (or less). For a concrete example, consider Technique 2, requiring $n^2 + 2n$ operations. A 100-step tower requires 10,200 operations (that is, $100^2 + 2 \times 100$). Doubling the tower to 200 steps increases the time by approximately fourfold, to 40,400 operations (that is, $200^2 + 2 \times 200$).

linear time O(n)

Linear Time. If the largest term in the formula is a constant times n, then the algorithm is said to be "**big-*O* of** n," written $O(n)$, and the algorithm is called **linear**. In a linear algorithm, doubling the input size makes the time increase by approximately twofold (or less). For example, a formula of $3n + 7$ is linear, so that $3 \times 200 + 7$ is about twice $3 \times 100 + 7$.

logarithmic time O(log n)

Logarithmic Time. If the largest term in the formula is a constant times a logarithm of n, then the algorithm is "**big-*O* of the logarithm of** n," written $O(\log n)$, and the algorithm is called **logarithmic**. (The base of the logarithm may be base 10, or possibly another base. We'll talk about the other bases in Section 10.3.) In a logarithmic algorithm, doubling the input size will make the time increase by no more than a fixed number of new operations, such as one more operation, or two more operations—or in general by c more operations, where c is a fixed constant. For example, Technique 3 for stair-counting has a logarithmic time formula. And doubling the size of a tower (perhaps from 500 stairs to 1000 stairs) never requires more than one extra operation.

Using big-*O* notation, we can express the time requirements of our three stair-counting techniques as follows:

Technique 1	$O(n)$
Technique 2	$O(n^2)$
Technique 3	$O(\log n)$

order of an algorithm

When a time analysis is expressed with big-*O*, the result is called the **order** of the algorithm. We want to reinforce one important point: Multiplicative constants are ignored in the big-*O* notation. For example, both $2n$ and $42n$ are linear formulas, so both are expressed as $O(n)$, ignoring the multiplicative constants 2 and 42. As you can see, this means that a big-*O* analysis loses some information about relative times. Nevertheless, a big-*O* analysis does provide some useful information for comparing algorithms. The stair example illustrates the most important kind of information provided by the order of an algorithm.

> The order of an algorithm generally is more important than the speed of the processor.

For example, using the quadratic technique (Technique 2) the fastest stair climber in the world is still unlikely to do better than a slowpoke—provided that the slowpoke uses one of the faster techniques. In an application such as sorting a list, a quadratic algorithm can be impractically slow on even moderately sized lists, regardless of the processor speed. To see this, notice the comparisons showing actual numbers for our three stair-counting techniques, which are shown in Figure 1.4.

FIGURE 1.4 Number of Operations for Three Techniques

Number of stairs (n)	Logarithmic $O(\log n)$ Technique 3, with $\lfloor \log_{10} n \rfloor + 1$ operations	Linear $O(n)$ Technique 1, with $3n$ operations	Quadratic $O(n^2)$ Technique 2, with $n^2 + 2n$ operations
10	2	30	120
100	3	300	10,200
1000	4	3000	1,002,000
10,000	5	30,000	100,020,000

Time Analysis of Java Methods

The principles of the stair-climbing example can be applied to counting the number of operations required by code written in a high-level language such as Java. As an example, consider the method implemented in Figure 1.5 on page 26. The method searches through an array of numbers to determine whether a particular number occurs.

As with the stair-climbing example, the first step of the time-analysis is to decide precisely what we will count as a single operation. For Java, a good choice is to count the total number of Java operations (such as an assignment, an arithmetic operation, or the < comparison). If a method calls other methods, then we would also need to count the operations that are carried out in the other methods.

for our first analysis, the number that we are searching for does not occur in the array

With this in mind, let's do a first analysis of the `search` method for the case where the array's length is a non-negative integer n, and (just to be difficult) the number that we are searching for does not occur in the array. How many operations does the search method carry out in all? Our analysis has three parts:

1. When the for-loop starts, there are two operations: an assignment to initialize the variable `i` to 0, and an execution of the test to determine whether `i` is less than `data.length`.
2. We then execute the body of the loop, and because the number that we are searching for does not occur, we will execute this body n times. How many operations occur during each execution of the loop body? We could count this number, but let's just say that each execution of the loop body requires k operations, where k is some number around 3 or 4 (including the work at the end of the loop where `i` is incremented and the termination test is executed). If necessary, we'll figure out k later, but for now it is enough to know that we execute the loop body n times, and each execution takes k operations, for a total of kn operations.
3. After the loop finishes, there is one more operation (a return statement).

The total number of operations is now $kn + 3$. The +3 is from the two operations before the loop and the one operation after the loop. Regardless of how big k is, this formula is always linear time. So, in the case where the sought-after number does not occur, the search method takes linear time. In fact, this is a frequent pattern that we summarize here:

Frequent Linear Pattern

A loop that does a fixed amount of operations n times requires $O(n)$ time.

FIGURE 1.5 Specification and Implementation of a `search` Method

Specification

- **search**

  `public static boolean search(double[ ] data, double target)`

  Notice that there is no precondition.

  Search an array for a specified number.

  Parameters:

  `data` – an array of double numbers

  `target` – a particular number that we are searching for

  Returns:

  `true` (to indicate that `target` occurs somewhere in the array), or `false` (to indicate that `target` does not occur in the array)

Implementation

```
public static boolean search(double[ ] data, double target)
{
   int i;

   for (i = 0; i < data.length; i++)
   {  // Check whether the target is at data[i].
      if (data[i] == target)
         return true;
   }

   // The loop finished without finding the target.
   return false;
}
```

Examples:

Suppose that the data array has the five numbers {2, 4, 6, 8, 10}. Then `search(data, 10)` *returns true, but* `search(data, 42)` *returns false.*

Later you will see additional patterns, resulting in quadratic, logarithmic, and other times. In fact, in Chapter 11 you will rewrite the `search` method in a way that uses an array that is sorted from smallest to largest, but requires only logarithmic time.

Worst-Case, Average-Case and Best-Case Analyses

The `search` method has another important feature: For any particular array size *n*, the number of required operations can differ depending on the exact parameter values. For example, with *n* equal to 100, the target could be 27 and the very first array element could also be 27—so the loop body executes just one time. On the other hand, maybe the number 27 doesn't occur until `data[99]` and the loop body executes the maximum number of times (*n* times). In other words, for any fixed *n*, different possible parameter values result in a different number of operations. When this occurs, we usually count the *maximum* number of required operations for inputs of a given size. Counting the maximum number of operations is called the **worst-case** analysis. In fact, the worst case for the `search` method occurs when the sought-after number is not in the array, which is the reason that we used the "not in array" situation in our previous analysis.

worst-case analysis

During a worst-case time analysis, you may sometimes find yourself unable to provide an exact count of the number of operations. If the analysis is a worst-case analysis, you may estimate the number of operations, always making sure that your estimate is on the high side. In other words, the actual number of operations must be guaranteed to be less than the estimate that you use in the analysis.

In Chapter 11, when we begin the study of searching and sorting, you'll see two other kinds of time-analysis: **average-case** analysis, which determines the average number of operations required for a given *n*, and **best-case** analysis, which determines the fewest number of operations required for a given *n*.

Self-Test Exercises

9. Each of the following is a formula for the number of operations in some algorithm. Express each formula in big-*O* notation.

   a. $n^2 + 5n$ d. $100n + 5$

   b. $3n^2 + 5n$ e. $5n + 3n^2$

   c. $(n + 7)(n - 2)$ f. the number of digits in $2n$

10. Write code for a method that computes the sum of all the numbers in an integer array. If the array's length is zero, then the sum should also be zero. Do a big-*O* time analysis of your method.

1.3 TESTING AND DEBUGGING

> *Always do right. This will gratify some people, and astonish the rest.*
>
> MARK TWAIN
> To the Young People's Society, February 16, 1901

program testing

Program testing occurs when you run a program and observe its behavior. Each time you execute a program on some input, you are testing to see how the program works for that particular input, and you are also testing to see how long the program takes to complete. Part of the science of *software engineering* is the systematic construction of a set of test inputs that is likely to discover errors, and such test inputs are the topic of this section.

Choosing Test Data

To serve as good test data, your test inputs need two properties.

Properties of Good Test Data

1. You must know what output a correct program should produce for each test input.
2. The test inputs should include those inputs that are most likely to cause errors.

Do not take the first property lightly—you must choose test data for which you know the correct output. Just because a program compiles, runs, and produces output that looks about right does not mean the program is correct. If the correct answer is 3278 and the program outputs 3277, then something is wrong. How do you know the correct answer is 3278? The most obvious way to find the correct output value is to work it out with pencil and paper using some method other than that used by the program. To aid you in doing this, you might choose test data for which it is easy to calculate the correct answer, perhaps by using smaller input values or by using input values for which the answer is well known.

Boundary Values

We will focus on two approaches for finding test data that is most likely to cause errors. The first approach is based on identifying and testing inputs called *boundary values*, which are particularly apt to cause errors. A **boundary value** of a problem is an input that is one step away from a different kind of behavior.

For example, recall the `dateCheck` method from the first self-test exercise on page 13. It has the following precondition:

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

Two boundary values for `dateCheck` are January 1, 1900 (since one step below this date is illegal) and December 31, 2099 (since one step above this date is illegal). If we expect the method to behave differently for "leap days," then we should try days such as February 28, 2000 (just before a leap day), February 29, 2000 (a leap day), and March 1, 2000 (just after a leap day).

test zero as a boundary value

Frequently zero has special behavior, so it is a good idea to consider zero to be boundary values whenever it is a legal input. For example, consider the `search` method from Figure 1.5 on page 26. This method should be tested with a data array that contains no elements (`data.length` is 0). For example:

```
double[ ] EMPTY = new double[0]; // An array with no elements

// Searching the EMPTY array should always return false.
if (search(EMPTY, 0))
   System.out.println("Wrong answer for an empty array.");
else
   System.out.println("Right answer for an empty array.");
```

test 1 and –1 as boundary values

The numbers 1 and –1 also have special behavior in many situations, so they should be tested as boundary values whenever they are legal input. For example, the `search` method should be tested with an array that contains just one element (`data.length` is 1). In fact, it should be tested twice with a one-element array: once when the target is equal to the element, and once when the target is different from the element.

boundary values are "one step away from different behavior"

In general, there is no precise definition of a boundary value, but you should develop an intuitive feel for finding inputs that are "one step away from different behavior."

Test Boundary Values

If you cannot test all possible inputs, at least test the boundary values. For example, if legal inputs range from zero to one million, then be sure to test input 0 and input 1000000. It is a good idea also to consider 0, 1, and –1 to be boundary values whenever they are legal input.

Fully Exercising Code

The second widely used testing technique requires intimate knowledge of how a program has been implemented. The technique, called **fully exercising code**, is stated with two rules:

1. Make sure that each line of your code is executed at least once by some of your test data. For example, there might be a portion of your code that is only handling a rare situation. Make sure that this rare situation is included among your set of test data.
2. If there is some part of your code that is sometimes skipped altogether, then make sure that there is at least one test input that actually does skip this part of your code. For example, there might be a loop where the body is sometimes executed zero times. Make sure that there is a test input that causes the loop body to be executed zero times.

profiler

Some Java programming environments have a software tool called a **profiler** to help fully exercise code. A typical profiler will generate a listing indicating how many times each method was called, so you can easily check that each method has been executed at least once. Some profilers offer more complete information, telling how often each individual statement of your program was executed. This can help you spot parts of your program that were not tested.

Fully Exercising Code

1. Make sure that each line of your code is executed at least once by some of your test data.
2. If there is part of your code that is sometimes skipped altogether, then make sure that there is at least one test input that actually does skip this part of your code.

Programming Tip: How to Debug

Finding a test input that causes an error is only half the problem of testing and debugging. After an erroneous test input is found, you still must determine exactly why the "bug" occurs, and "debug the program." When you have found an error, there is an impulse to dive right in and start changing code. It is tempting to look for suspicious parts of your code and change these suspects to something "that might work better."

Avoid the temptation.

An impulsive change to suspicious code almost always makes matters worse. Instead, you must discover *exactly* why a test case is failing and limit your changes to *corrections of known errors.* Once you have corrected a known error, all test cases should be rerun.

Tracking down the exact reason why a test case is failing can be difficult. For large programs, tracking down errors is nearly impossible without the help of a software tool called a **debugger.** A debugger executes your code one line at a time, or it may execute your code until a certain condition arises. Using a debugger, you can specify what conditions should cause the program execution to pause. You can also keep a continuous watch on the location of the program execution and on the values of specified variables.

Debugging Tips

1. Never start changing suspicious code in the hope that the change "might work better."
2. Instead, discover *exactly* why a test case is failing and limit your changes to *corrections of known errors*.
3. Once you have corrected a known error, all test cases should be rerun.

Use a software tool called a *debugger* to help track down exactly why an error occurs.

Self-Test Exercises

11. Suppose you write a program that accepts as input any integer in the range –20 through 20, and outputs the number of digits in the input integer. What boundary values should you use as test inputs?
12. Suppose you write a program that accepts a single line as input, and outputs a message telling whether or not the line contains the letter A, and whether or not it contains more than three A's. What is a good set of test inputs?
13. What does it mean to "fully exercise" code?

CHAPTER SUMMARY

- The first step in producing a program is to write down a precise description of what the program is supposed to do.
- *Pseudocode* is a mixture of Java (or some other programming language) and English (or some other natural language). Pseudocode is used to express algorithms so that you are not distracted by details about Java syntax.
- One good method for specifying what a method is supposed to do is to provide a *precondition* and *postcondition* for the method. These form a

contract between the programmer who uses the method and the programmer who writes the method.

- *Time analysis* is an analysis of how many operations an algorithm requires. Often, it is sufficient to express a time analysis in big-*O* notation, which is the *order* of an algorithm. The order analysis is often enough to compare algorithms and estimate how running time is affected by changing input size.
- Three important examples of big-*O* analyses are *linear* (i.e., $O(n)$), *quadratic* (i.e., $O(n^2)$), and *logarithmic* (i.e., $O(\log n)$).
- An important testing technique is to identify and test *boundary values.* These are values that lie on a boundary between different kinds of behavior for your program.
- A second important testing technique is to ensure that your test cases are *fully exercising* the code. A software tool called a *profiler* can aid in fully exercising code.
- During debugging, you should discover exactly why a test case is failing and limit your changes to corrections of known errors. Once you have corrected a known error, all test cases should be rerun. Use a software tool called a *debugger* to help track down exactly why an error occurs.

Solutions to Self-Test Exercises

1. The method returns 7 on July 22, 2003. On both July 30, 2003 and February 1, 2004 the method returns 0 (since July 29, 2003 has already passed).

2. Yes. In order to use a method, all you need to know is the specification.

3.
```
System.out.print("$");
printNumber(boodle, 12, 2);
```

4. printSqrt

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

Prints the square root of a number.

Parameters:

`x` – any non-negative double number

Precondition:

`x >= 0`.

Postcondition:

The positive square root of `x` has been printed to standard output.

Throws: `IllegalArgumentException`

Indicates that `x` is negative.

5. sphereVolume

`public static double sphereVolume(double radius)`
Computes the volume of a sphere.

Parameters:
`radius`– any non-negative double number

Precondition:
`radius >= 0`.

Returns:
the volume of a sphere with the specified radius

Throws: `IllegalArgumentException`
Indicates that `radius` is negative.

6.
```
if (x < 0)
  throw new IllegalArgumentException
  ("x is negative: " + x);
```

7. A `final` variable is given an initial value when it is declared, and this value will never change while the program is running.

8. Most of the specification was automatically produced using the Javadoc tool described in Appendix H.

9. Part d is linear (i.e., $O(n)$); part f is logarithmic (i.e., $O(\log n)$); all of the others are quadratic (i.e., $O(n^2)$).

10. Here is a specification for the method, along with an implementation:

sum

`public static int sum(int[ ] a)`
Computes the sum of the numbers in an array.

Parameters:
`a` – an array whose numbers are to be summed

Returns:
the sum of the numbers in the array (which is zero if the array has no elements)

```
public static int sum(int[ ] a)
{
  int answer, i;

  answer = 0;
  for (i = 0; i < a.length; i++)
     answer += a[i];
  return answer;
}
```

Our solution uses `answer += i`, which causes the current value of `i` to be added to what's already in `answer`.

For a time analysis, let n be the array size. There are two assignment operations (`i = 0` and `answer = 0`). The `<` test is executed $n + 1$ times (the first n times it is `true`, and the final time, with `i` equal to n, it is `false`). The `++` and `+=` operations are each executed n times, and an array subscript (`a[i]`) is taken n times. The entire code is $O(n)$.

11. As always, 0, 1, and –1 are boundary values. In this problem, –20 (smallest value) and 20 (largest value) are also boundary values. Also 9 and 10 (since the number changes from a single digit to two digits) and –9 and –10. (By the way, this particular problem is small enough that it would be reasonable to test *all* legal inputs, rather than just testing the boundary values.)

12. You should include an empty line (with no characters before the carriage return) and lines with 0, 1, 2, 3 A's. Also include a line with 4 A's (the smallest case with more than three) and a line with more than 4 A's. For the lines with 1 or more A's, include lines that have only the A's, and also include lines that have A's together with other characters. Also test the case where all the A's appear at the front or the back of the line.

13. See the box "Fully Exercising Code" on page 30.