Chapter 8

Living with Java

This chapter deals with some miscellaneous issues: program output, formatting, errors, constants, and encapsulation.

8.1 Standard Output

The Java runtime environment (JRE) supplies a plethora of standard objects. The most widely used standard object is `System.out`. `System.out` is used to display information to the "console." The console is an output window often called *standard out*. Under Unix/Linux and Mac OS X, standard out corresponds to a shell. Under Microsoft Windows, it is called the command shell (`cmd.exe`). In the DrJava environment, standard out is displayed in the Console pane which is made visible by selecting the Console tab.

The `println()` method from `System.out` outputs its argument to the console. If executed within the DrJava interpreter, it outputs to the Interactions pane as well.

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> System.out.println(10);
10
\end{verbatim}
A quick check of the Console pane reveals:
\begin{Console}
10
\end{Console}
Here is a little more elaborate session:
\begin{Interactions}
Welcome to DrJava.  Working directory is /Users/rick/java
> int x = 2, y = 3;
> System.out.println("The sum of " + x + " and " + y +
                     " is " + (x + y));
The sum of 2 and 3 is 5
> System.out.println(x + " + " + y + " = " + (x + y));
2 + 3 = 5
```


The `System.out.print()` method works exactly like `println()`, except subsequent output will appear on the same line. Compare the following statements:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> System.out.println(10); System.out.println(20);
10
20
> System.out.print(10); System.out.print(20);
1020>
```

The `System.out.println()` and `System.out.print()` methods are overloaded since they will accept any primitive type or reference type.

Well-placed `println()` statements are invaluable for understanding what a program is doing when it is running.

8.2 Formatted Output

A companion method of `System.out.println()` is `System.out.printf()`. The `System.out.printf()` method provides formatted output for primitive types and strings. Special control codes denote display widths and decimal places. An example call is

```
System.out.printf("[%5d]-[%7.4f]: %s %n", 45, 3.14159, "Test values");
```

which outputs

```
[   45]-[ 3.1416]: Test values
```

The `printf()` method accepts a comma-separated list of arguments. The first argument, called the *format string*, is a string literal enclosed by quotation marks. The other arguments (if any) depend on formatting control codes within the format string. The contents of the format string are printed literally with the exception of the control codes which denote arguments that follow the format string. The first control code corresponds to the first argument after the format string, the second control code corresponds to the second argument after the format string, etc. Table 8.1 lists some commonly used control codes. The `printf()` method provides many other capabilities described in the class documentation. For example, if an optional zero appears before the width specifier in a numeric control code, as in

```
%04d
```

then the number is padded with leading zeros instead of leading blank spaces to make the number fill the specified width.

Notice in our example that the square brackets, dash, colon, and spaces appear as they do in the format string. The value 45 is displayed with three spaces preceding it to make a total of five spaces for the value. For 3.14159, a width of seven spaces is allotted for the entire value, and three decimal places are to be displayed. Note that the number of slots of characters between the displayed square brackets is seven, and the value is rounded to four decimal places. The string appears literally as it does within the quotes.

If the format string provides a control code like `%d` and `%f` that requires a trailing argument, the programmer must ensure that an argument of the appropriate type appears in its proper place in the list that follows the format

Control Code	Meaning
`%d`	Decimal (that is, base 10) integer (any integral type). An optional single number between the `%` and the `d` specifies field width for right justification.
`%f`	Floating point number (`float` or `double`). Optional field width and/or number of decimal places to display can appear between the `%` symbol and the `f`.
`%s`	String. Can be a string literal (as in this example) or a string variable.
`%n`	Platform-independent end-of-line character. The major operating systems (Unix/Linux, Microsoft Windows, and Macintosh) use different characters to signify a newline. This control code is translated into the proper newline character for the platform upon which the program is running.
`%%`	Percent symbol. Since `%` indicates that a control code symbol follows, this provides a way to print a literal percent symbol.

Table 8.1: Control codes used in the `System.out.printf()` format string

string. The compiler does *not* check for compliance. If too few trailing arguments are provided or the type of the argument does not agree with the control code, the program will generate a runtime error when the offending `printf()` statement is executed. Extra trailing arguments are ignored.

8.3 Keyboard Input

DrJava's Interactions pane provides a *read-evaluate-print* loop:

1. The user's keystrokes are read by the interpreter.
2. Upon hitting Enter, the user's input is evaluated, if possible.
3. The result of the evaluation, if any, is then displayed.

This way of user interaction is built into DrJava and we do not need to provide code to make it work. On the other hand, standalone programs are completely responsible for obtaining user input at the right time and providing the proper feedback to the user. This interaction must be handled by statements within the program.

We have seen how `System.out.print()`, `System.out.println()`, and `System.out.printf()` can be used to provide output. User input can be handled by `Scanner` objects. The `Scanner` class is found in the `java.util` package and must be imported for use within programs. `SimpleInput` (8.1) shows how a `Scanner` object can be used:

```
import java.util.Scanner;

public class SimpleInput {
    public void run() {
        int val1, val2;
        Scanner scan = new Scanner(System.in);
        System.out.print("Please enter an integer: ");
```



```
        val1 = scan.nextInt();
        System.out.print("Please enter another integer: ");
        val2 = scan.nextInt();
        System.out.println(val1 + " + " + val2 + " = " + (val1 + val2));
    }
}
```

Listing 8.1: `SimpleInput`—adds two integers

Compile this class and try it out:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/Documents/src/java/DrJava
> new SimpleInput().run();
Please enter an integer: 34
Please enter another integer: 10
34 + 10 = 44
```

The Interactions pane prints the prompt message (*Please enter an integer:*) without its familiar "**>**" prompt. After entering the values, their sum is then printed. Notice that you can enter both numbers together at the first prompt, separated by at least one space, and then press Enter. The first `nextInt()` grabs the first integer from the keyboard buffer, and the next call to `nextInt()` grabs the remaining value left over on the same line.

The `Scanner`'s constructor here accepts a standard `System.in` object which is associated with the keyboard. It can be associated with other input streams such as text files, but we will not consider those options here.

`Scanner` objects have a variety of methods to read the different kinds of values from the keyboard. Some of these methods are shown in Table 8.2:

Some Methods of the `Scanner` Class
`int nextInt()` Returns the integer value typed in by the user. This method produces a runtime error if the key sequence typed in cannot be interpreted as an `int`.
`double nextDouble()` Returns the double-precision floating point value typed in by the user. This method produces a runtime error if the key sequence typed in cannot be interpreted as a `double`.
`String next()` Returns the next string token typed in by the user. Tokens are separated by spaces.
`String nextLine()` Returns the string typed in by the user including any spaces.

Table 8.2: A subset of the methods provided by the `Scanner` class

8.4 Source Code Formatting

Program comments are helpful to human readers but ignored by the compiler. The way the source code is formatted is also important to human readers but is of no consequence to the compiler. Consider

`ReformattedCircleCalculator` (8.2), which is a reformatted version of `CircleCalculator` (4.1):

```
                                        public
                                        class
ReformattedCircleCalculator{final
                                               private double PI=3.14159
;public double circumference
                                                                  (double


radius) {return 2*PI
                                          *
radius;}public double
                                                                area(double
radius) {return PI*radius*radius;}}
```

Listing 8.2: `ReformattedCircleCalculator`—A less than desirable formatting of source code

To an experienced Java programmer `CircleCalculator` is easier to understand more quickly than `ReformattedCircleCalculator`. The elements of `CircleCalculator` are organized better. What are some distinguishing characteristics of `CircleCalculator`?

- Each statement appears on its own line. A statement is not unnecessarily split between two lines of text. Visually, one line of text implies one action (statement) to perform.
- Every close curly brace aligns vertically with the line that ends with the corresponding open curly brace. This makes it easier to determine if the curly braces match and nest properly. This becomes very important as more complex programs are tackled since curly braces frequently can become deeply nested.
- Contained elements like the `circumference()` method within the class and the statements within the `circumference()` method are indented several spaces. This visually emphasizes the fact that the elements are indeed logically enclosed.
- Spaces are used to spread out statements. Space around the assignment operator (=) and addition operator (+) makes it easier to visually separate the operands from the operators and comprehend the details of the expression.

Most software organizations adopt a set of *style guidelines*, sometimes called *code conventions*. These guidelines dictate where to indent and by how many spaces, where to place curly braces, how to assign names to identifiers, etc. Programmers working for the organization are required to follow these style guidelines for code they produce. This allows a member of the development team to more quickly read and understand code that someone else has written. This is necessary when code is reviewed for correctness or when code must be repaired or extended, and the original programmer is no longer with the development team.

One good set of style guidelines for Java can be found at Sun's web site:

`http://www.javasoft.com/codeconv/index.html`

The source code in this book largely complies with Sun's code conventions.

8.5 Errors

Beginning programmers make mistakes writing programs because of inexperience in programming in general or because of unfamiliarity with a programming language. Seasoned programmers make mistakes due to carelessness or because the proposed solution to a problem is faulty and the correct implementation of an incorrect solution will not produce a correct program. A programming error falls under one of three categories:

- compile-time error
- runtime error
- logic error

Compile-time errors. A compile-time error results from the misuse of the language. A *syntax error* is a common compile-time error. For example, in English one can say

The boy walks quickly.

This sentence uses correct syntax. However, the sentence

The boy walk quickly.

is not correct syntactically: the number of the subject (singular form) disagrees with the number of the verb (plural form). It contains a syntax error. It violates a grammatical rule of the English language. Similarly, the Java statement

```
x = 10 + 2;
```

is syntactically correct because it obeys the rules for the structure of an assignment statement described in § 2.2. However, consider replacing this assignment statement with a slightly modified version:

Interactions

```
> 10 + 2 = x;
Error: Bad left expression in assignment
```

When the interpreter attempts to compile `BadAssignment`, it issues an error message.[1]

Compilers have the reputation for generating cryptic error messages. They seem to provide little help as far as novice programmers are concerned. The message issued here is actually very useful. The "left expression" is the expression to the left of the assignment operator (=). It makes no sense to try to change the value of 10 + 2—it is always 12.

`CircleCalculatorWithMissingBrace` (8.3) contains another kind of syntax error. All curly braces must be properly matched, but in `CircleCalculatorWithMissingBrace` the curly brace that closes the `circumference()` method is missing.

```
public class CircleCalculatorWithMissingBrace {
    final double PI = 3.14159;
```

[1] The messages listed here are generated by DrJava's compiler. Other compilers should issue similar messages, but they may be worded differently.


```
    double circumference(double radius) {
        return 2 * PI * radius;
    double area(double radius) {
        return PI * radius * radius;
    }
}
```

Listing 8.3: `CircleCalculatorWithMissingBrace`—A missing curly brace

Figure 8.1 shows the result of compiling `CircleCalculatorWithMissingBrace` (8.3). As shown in Figure 8.1,

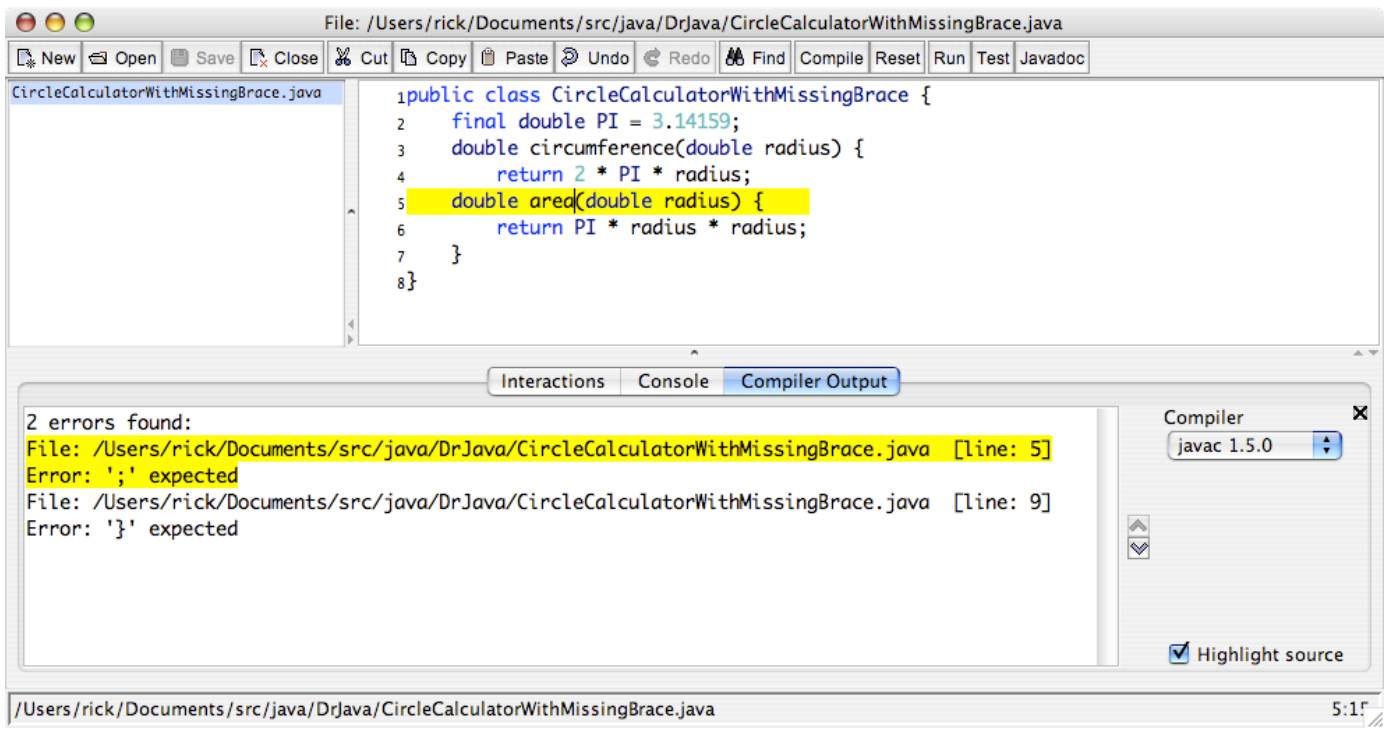

Figure 8.1: Result of compiling a Java source file with a compile-time error

an attempt to compile this code results in the following messages in the CompilerOutput pane:

Compiler Output

```
2 errors found:
File:
CircleCalculatorWithMissingBrace.java
[line: 5]
Error: ';' expected
File:
CircleCalculatorWithMissingBrace.java
[line: 9]
Error: '}' expected
```

This report is an example of how the compiler does indeed detect an error, but it does not appear to be the error that we expected! The first error it reports is on Line 5, which is in fact where the problem lies. The compiler

faithfully attempts the translate into bytecode the source code provided by the programmer, and it reports errors only when encounters an illegal situation. In Java, it is illegal to define a method directly within another method. Said another way, Java does not permit nested method definitions. When the compiler is scanning Line 5, it assumes the programmer is attempting to declare a `double` variable named `area` within the `circumference()` method, which is legal to do. When it sees the left parenthesis, it then detects an attempt to define a method named `area` inside another method (`circumference()`), which is illegal. Had the programmer instead used a semicolon (`;`), the declaration would have been legal. The compiler then trudged on, eventually determining that the three opening braces (`{`) it counted were not properly matched up with an equal number of closing braces. (It ran out of source code on Line 9 while only finding two matching closing braces.)

`MissingDeclaration` (8.4) contains another error of omission. It is an error to use a variable that has not been declared.

```
class MissingDeclaration {
    int attemptToUseUndeclaredVariable() {
        x = 2;
        return x;
    }
}
```

Listing 8.4: `MissingDeclaration`—A missing declaration

`MissingDeclaration` uses variable `x`, but `x` has not been declared. This type of compile-time error is not a syntax error, since all of the statements that are present are syntactically correct. The compiler reports

Compiler Output

```
2 errors found:
File:
MissingDeclaration.java
[line: 3]
Error: cannot find symbol
symbol  : variable x
location: class MissingDeclaration
File:
MissingDeclaration.java
[line: 4]
Error: cannot find symbol
symbol  : variable x
location: class MissingDeclaration
s
```

Here, the compiler flags both lines that attempt to use the undeclared variable `x`. One "error," forgetting to declare `x`, results in two error messages. From the programmer's perspective only one error exists—the missing declaration. The compiler, however, cannot determine what the programmer intended; it simply notes the two uses of the variable that has never been declared, printing two separate error messages. It is not uncommon for beginning programmers to become disheartened when the compiler reports 50 errors and then be pleasantly surprised when one simple change to their code removes all 50 errors!

The message “cannot find symbol” means the compiler has no record of the symbol (variable `x`) ever being declared. The compiler needs to know the type of the variable so it can determine if it is being used properly.

Runtime errors. The compiler ensures that the structural rules of the Java language are not violated. It can detect, for example, the malformed assignment statement and the use of a variable before its declaration. Some violations of the language cannot be detected at compile time, however. Consider `PotentiallyDangerousDivision` (8.5):

```
class PotentiallyDangerousDivision {
    //  Computes the quotient of two values entered by the user.
    int divide(int dividend, int divisor) {
            return dividend/divisor;
    }
}
```

Listing 8.5: `PotentiallyDangerousDivision`—Potentially dangerous division

The expression

```
dividend/divisor
```

is potentially dangerous. Since the two variables are of type `int`, integer division will be performed. This means that division will be performed as usual, but fractional results are dropped; thus, $15 \div 4 = 3$, not 3.75, and rounding is not performed. If decimal places are needed, floating point numbers are required (§ 2.1).

This program works fine (ignoring the peculiar behavior of integer division), until zero is entered as a divisor:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> d = new PotentiallyDangerousDivision();
> d.divide(15, 4)
3
> d.divide(15, 0)
ArithmeticException: / by zero
  at PotentiallyDangerousDivision.divide(PotentiallyDangerousDiv...
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAcc...
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingM...
  at java.lang.reflect.Method.invoke(Method.java:585)
```

Division by zero is undefined in mathematics and is regarded as an error in a Java program. This type of error, however, cannot be detected by the compiler since the value of the divisor is not known until the program is executing (runtime). It would be inappropriate (and certainly annoying) if the compiler issued a message in every instance where division by zero is a possibility. Fortunately, the Java runtime environment checks for various errors such as division by zero. When such an error is encountered, a message is displayed and the program’s execution is normally terminated. This action is called *throwing an exception*. The message indicates that the problem is division by zero and gives the source file and line number of the offending statement.

Logic errors. Consider the effects of replacing the statement


```
    return dividend/divisor;
```

in `PotentiallyDangerousDivision` (8.5) with the statement:

```
    return divisor/dividend;
```

The program compiles with no errors. It runs, and unless a value of zero is entered for the dividend, no runtime errors arise. However, the answer it computes is in general not correct. The only time the correct answer is printed is when `dividend` = `divisor`. The program contains an error, but neither the compiler nor the runtime environment is able detect the problem. An error of this type is known as a *logic error*.

Beginning programmers tend to struggle early on with compile-time errors due to their unfamiliarity with the language. The compiler and its error messages are actually the programmer's best friend. As the programmer gains experience with the language and the programs written become more complicated, the number of syntax errors decrease and the number of logic errors increase. Unfortunately, both the compiler and runtime environments are powerless to provide any insight into the nature and/or location of logic errors. Logic errors, therefore, tend to be the most difficult to find and repair. Tools such as debuggers are frequently used to help locate and fix logic errors, but these tools are far from automatic in their operation.

The runtime exceptions mentioned in § 8.5 arise from logic errors. In the division by zero example, programmers can take steps to ensure such division by zero does not occur. `BetterDivision` (8.6) shows how it might be done.

```
class BetterDivision {
    //  Computes the quotient of two values entered by the user.
    int divide(int dividend, int divisor) {
        if (divisor == 0) {
            System.out.println("Warning!  Division by zero, result " +
                               "is invalid");
            return 2147483647;     //  Return closest int to infinity!
        } else {
            return dividend/divisor;
        }
    }
}
```

Listing 8.6: `BetterDivision`—works around the division by zero problem

`BetterDivision` (8.6) avoids the division by zero runtime error that causes the program to terminate prematurely, but it still alerts the user that there is a problem. Another application may handle the situation in a different way; for example, use some default value for `divisor` instead of zero.

Errors that escape compiler detection (runtime errors and logic errors) are commonly called *bugs*. Because of the inability of the compiler to detect these problems, such bugs are the major source of frustration for developers.

8.6 Constants Revisited

As mentioned earlier (§ 2.2), the proper use of constants can make Java code more readable. Constants have other advantages besides readability. Here we consider the advantages.

Why not use a literal value instead of a symbolic constant (for example, 3.14159 vs. `PI`)? A defined constant has several advantages over a literal value:

- The symbolic constant is potentially more readable. For example, it may be better to use the constant `MOLE` than to use its literal value `6.023e23` since the literal number may easily blend in with other literal values in an expression and the symbolic constant stands out with clear meaning.
- The symbolic constant hides specific details that are not relevant to the task at hand. For example, the area of a circle is $A = \pi r^2$. If the constant `PI` is defined somewhere, we can write the assignment

  ```
  area = PI * radius * radius;
  ```

  without worrying about how many decimal places of precision to use for `PI`.
- The specific information about the value of the constant is located in exactly one place. If the value must be changed, the programmer need only change one line in the program instead of possibly hundreds of lines throughout the program where the value is used. Consider a software system that must perform a large number of mathematical computations (any program that uses graphics extensively would qualify). Some commercial software systems contain several million lines of code. Calculations using the mathematical constant π could appear hundreds or thousands of times within the code. Suppose the program currently uses 3.14 for the approximation, but users of the program demand more precision, so the developers decide to use 3.14159 instead. Without constants, the programmer must find every occurrence of 3.14 in the code and change it to 3.14159. It is a common mistake to change most, but not all, values, thus resulting in a program that has errors. If a constant is used instead, like

  ```
  final double PI = 3.14;
  ```

  and the symbolic constant `PI` is used throughout the program where π is required, then only the constant initialization need be changed to update `PI`'s precision.

  One might argue: the same effect can be accomplished with an editor's search-and-replace feature. Just go ahead and use the literal value 3.14 throughout the code. When it must be changed, use the global search-and-replace to change every occurrence of 3.14 to 3.14159. This approach, however, is not a perfect solution. Some occurrences of this text string should *not* be replaced! Consider the statement:

  ```
  gAlpha = 109.33381 * 93.14934 - entryAngle;
  ```

  that could be buried somewhere in the thousands of lines of code. The editor's search-and-replace function would erroneously convert this statement to

  ```
  gAlpha = 109.33381 * 93.14159934 - entryAngle;
  ```

  This error is particularly insidious, since the intended literal value (the approximation for π) is not changed dramatically but only slightly. In addition, calculations based on `gAlpha`'s value, based in how the assignment statement was changed, may be off by only a small amount, so the bug may not be detected for some time. If the editor allowed the programmer to confirm before replacement, then the programmer must visually inspect each potential replacement, think about how the string is being used in context, and then decide whether to replace it or not. Obviously this process would be prone to human error. The proper use of symbolic constants eliminates this potential problem altogether.

8.7 Encapsulation

So far we have been using the `public` and `private` qualifiers for class members without much discussion about their usage. Recall `RYGTrafficLight` (7.4) from § 7.2. It nicely modeled traffic light objects and provided a methods so that clients can interact with the lights in well-defined but restricted ways. The state of each traffic light object is determined by exactly one integer: its `color` instance variable. This variable is well protected from direct access by clients:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> RYGTrafficLight t = new RYGTrafficLight();
> t.color = 182;
Error: This object cannot be modified
```

When developing complex systems, allowing indiscriminate access to an object's internals can be disastrous. It is all too easy for a careless, confused, or inept programmer to change an object's state in such a way as to corrupt the behavior of the entire system. A malicious programmer may intentionally tweak one or more objects to wreak havoc with the system. If the software system controls a medical device or military missile system, the results can be deadly.

Java provides several ways to protect the internals of an object from the outside world, but the simplest strategy is the one we have been using: fields and methods, generically referred to as class members, can be qualified as either `public` or `private`. These qualifiers mean the following:

- `public`—the class member is freely accessible by code in any class anywhere. `public` is the least restrictive qualifier.
- `private`—the class member is accessible only by code within the class itself. This is the most restrictive qualifier.

Said another way, `private` information is invisible to the outside world (clients of the class), but `public` information is freely accessible.[2] The compiler enforces the inaccessibility of `private` members. In traffic light objects, client code cannot modify `color` directly either accidentally or on purpose.

The following represent rules of thumb for using access specifiers:

- Data, that is, instance variables, generally should be `private`.
- Methods that constitute a service to be provided to clients should be `public`.
- Methods that serve other methods within the class but are not meant to be used outside the class should be `private`.

Why would a programmer intentionally choose to limit access to parts of an object? Restricting access obviously limits the client's control over the objects it creates. While this may appear to be a disadvantage at first glance, this access restriction actually provides a number of advantages:

- **Flexibility in implementation**. A class can be conceptually separated into two parts:

[2] It is not possible to protect an object from code within itself. If it appears that parts of a class should be protected from some of its methods, the class should be split up into multiple classes with suitable restrictions among the split up classes.

1. **Interface—the public part**. Clients see and can use the public part of an object. The public methods and public variables of a class constitute the *interface* of the class. A class's interface specifies *what* it does.
2. **Implementation—the hidden part**. Clients cannot see any `private` methods or `private` variables. Class developers are free to do whatever they want with the private parts of the class. A class's implementation specifies *how* it accomplishes what it needs to do.

We would like our objects to be black boxes: clients shouldn't need to know *how* they work, but clients rely on *what* they can do.

Many real-world objects follow this design philosophy. Consider a digital wristwatch. Its display gives its user the current time and, perhaps, date. It can produce different output in different modes; for example, elapsed time in stopwatch mode. It presents to its user only a few buttons for changing modes, starting and stopping stopwatches, and setting the time. *How* it does what is does is irrelevant to its user; its user is concerned with *what* it does. Its user risks great peril by opening the watch and looking at its intricate internal details. The user is meant to interact with the watch only through its interface—the display and buttons.

Similarly, an automobile presents an accelerator pedal to its user. The user knows that pushing the pedal makes the car go faster. That the pedal is connected to the fuel system (and possibly other systems, like cruise control) through a cable or other type of linkage is of concern only to the automotive designer or mechanic. Most drivers prefer to be oblivious to the under-the-hood details.

Changing the interface of a class disturbs client code that has been written to use objects of that class. For example, in the `RYGTrafficLight` class (7.4), the `draw()` method is `public` and is therefore part of the interface of the `RYGTrafficLight` class. If, after the `RYGTrafficLight` class has been made available to clients, the authors of the `RYGTrafficLight` class decide to eliminate the `draw()` method or rewrite it to accept parameters, then any existing client code that uses `draw()` according to its original definition will no longer be correct. We say the change in `RYGTrafficLight`'s interface *breaks* the client code. Class authors have no flexibility to alter the interface of a class once the class has been released for clients to use. On the other hand, altering the private information in a class will not break existing client code that uses that class, since private class information is invisible to clients. Therefore, a class becomes less resilient to change as more of its components become exposed to clients. To make classes as flexible as possible, which means maximizing the ability to make improvements to the class in the future, hide as much information as possible from clients.

- **Reducing programming errors**. Parts of a class that are `private` cannot be misused by client code since the client cannot see the `private` parts of a class. Properly restricting client access can make it impossible for client code to put objects into an undefined state. In fact, if a client can coax an object into an illegal state via the class interface, then the class has not been implemented properly. For example, recall the `color` instance variable of `RYGTrafficLight` (7.4). The decision to make `color` private means that the only way color can assume a value outside the range of $1 \ldots 3$ is if one of the `RYGTrafficLight`'s methods is faulty. Clients can never place a traffic light into an illegal state.

- **Hiding complexity**. Objects can provide a great deal of functionality. Even though a class may provide a fairly simple interface to clients, the services it provides may require a significant amount of complex code to accomplish their tasks. One of the challenges of software development is dealing with the often overwhelming complexity of the task. It is difficult, if not impossible, for one programmer to be able to comprehend at one time all the details of a large software system. Classes with well-designed interfaces and hidden implementations provide a means to reduce this complexity. Since `private` components of a class are hidden, their details cannot contribute to the complexity the client programmer must manage. The client programmer need not be concerned with exactly how an object works, but the details that make the object work are present nonetheless. The trick is exposing details only when necessary.

  - **Class designer**. The class implementer must be concerned with the hidden implementation details of the class. The class implementer usually does not have to worry about the context in which the class will be used because the class may be used in many different contexts. From the perspective of the class designer, the complexity of the client code that may use the class is therefore eliminated.
  - **Applications developer using a class**. The developer of the client code must be concerned with the details of the application code being developed. The application code will use objects. The details of the class these objects represent are of no concern to the client developers. From the perspective of the client code designer, the complexity of the code within classes used by the client code is therefore eliminated.

This concept of information hiding is called *encapsulation*. Details are exposed only to particular parties and only when appropriate. In sum, the proper use of encapsulation results in

- software that is more flexible and resilient to change,
- software that is more robust and reliable, and
- software development that is more easily managed.

8.8 Summary

- `System.out.println()` is used to display output to the console.
- `System.out.printf()` is used to display formatted output.
- `System.out.printf()` uses a format string and special control codes to justify and format numerical data.
- Coding conventions make it easier for humans to read source code, and ultimately humans must be able to easily read code in order to locate and fix bugs and extend programs.
- The three kinds of errors encountered by programmers are compile-time, runtime, and logic errors.
- The compiler notes compile-time errors; programs with compile-time errors cannot be compiled into bytecode.
- Runtime errors cannot be detected by the compiler but arise during the program's execution; runtime errors terminate the program's execution abnormally.
- Logic errors are usually the most difficult to fix. No error messages are given; the program simply does not run correctly.
- Symbolic constants should be used where possible instead of literal values. Symbolic constants make a program more readable, they shield the programmer from irrelevant information, and they lead to more maintainable code.
- Encapsulation hides code details in such a way so that details are only available to parties that need to know the details.
- The proper use of encapsulation results in code that is more flexible, robust, and easily managed.
- The public part of an object (or class) is called its interface.
- The private part of an object (or class) is called its implementation.

8.9 Exercises

1. What simple statement can be used to print the value of a variable named `x` on the console window?
2. The following control codes can be used in the format string of the `System.out.printf()` method call: `%d`, `%f`, `%s`, `%n`, `%%`. What do each mean?
3. Look at the documentation for `System.out.printf()` and find at least five control codes that are not mentioned in this chapter. What is the meaning of each of these codes?
4. Suppose `f` is a `double`. How can we print the value of `f` occupying 10 character positions, rounded to two decimal places, with leading zeroes? Provide the necessary statement and check with the following values: -2.007, 0.000003, and 1234.
5. Since the compiler is oblivious to it, why it consistent, standard source code formatting important?
6. List four generally accepted coding standards that are commonly used for formatting Java source.
7. Look at Sun's code convention website and list two suggested coding proactices that are not mentioned in this chapter.
8. What are the three main kinds of errors that Java programs exhibit? Since programmers can more easily deal with some kinds of errors than others, rank the three kinds of errors in order of increasing difficulty to the programmer. Why did you rank them as you did?
9. What kind of error would result from the following programming mistakes, if any?
   (a) Forgetting to end a statement in a method with a semicolon.
   (b) Using a local variable in a method without declaring it.
   (c) Printing a declared local variable in a method without initializing it.
   (d) Using an instance variable in a method without declaring it.
   (e) Printing a declared instance variable in a method without initializing it.
   (f) Omitting a `return` statement in a non-`void` method.
   (g) Integer division by zero.
   (h) Floating-point division by zero.
   (i) Calling an infinitely-recursive method, such as:

```
public int next(int n) {
    return next(n + 1);
}
```

   First try to answer without using a computer and then check your answers by intentionally making the mistakes and seeing what errors, if any, actually occur.
10. List three advantages of using symbolic constants instead of literal values.
11. What is encapsulation, and how is it beneficial in programming?
12. Should instance variables generally be `public` or `private`?
13. How do you decide whether a method should be declared `public` or `private`?
14. How is a class's interface distinguished from its implementation?

