Appendix A

Documentation and Programming Style

Contents

Prerequisite

Some knowledge of Java

Most programs are used many times and are changed either to fix bugs or to accommodate new demands by the user. If the program is not easy to read and to understand, it will not be easy to change. It might even be impossible to change without heroic efforts. Even if you use your program only once, you should pay some attention to its readability. After all, you will have to read the program to debug it.

In this appendix, we discuss three techniques that can help make your program more readable: meaningful names, indenting, and comments.

Naming Variables and Classes

A.1 Names without meaning are almost never good variable names. The name you give to a variable should suggest what the variable is used for. If the variable holds a count of something, you might name it `count`. If the variable holds a tax rate, you might name it `taxRate`.

In addition to choosing names that are meaningful and legal in Java, you should follow the normal practice of other programmers. That way it will be easier for them to read your code and to combine your code with their code, when you work on a project with more than one person. By convention, each variable name begins with a lowercase letter, and each class name begins with an uppercase letter. If the name consists of more than one word, use a capital letter at the beginning of each word, as in the variable `numberOfTries` and the class `StringBuffer`.

Use all uppercase letters for named constants to distinguish them from other variables. Use the underscore character to separate words, if necessary, as in `INCHES_PER_FOOT`.

Indenting

A.2 A program has a structure: Smaller parts are within larger parts. You use indentation to indicate this structure and thereby make your program easier to read. Although Java ignores any indentation you use, indenting consistently is essential to good programming style.

Each class begins at the left margin and uses braces to enclose its definition. For example, you might write

```
public class CircleCalculation
{
   . . .
} // end CircleCalculation
```

The data fields and methods appear indented within these braces, as illustrated in the following simple program:

```
public class CircleCalculation
{
   public static final double PI = Math.PI;
   public static void main(String[] args)
   {
      double radius; // In inches
      double area;   // In square inches
      . . .
   } // end main
} // end CircleCalculation
```

Within each method, you indent the statements that form the method's body. These statements in turn might contain compound statements that are indented further. Thus, the program has statements nested within statements.

Each level of nesting should be indented from the previous level to show the nesting more clearly. The outermost structure is not indented at all. The next level is indented. The structure nested within that is double indented, and so on. Typically, you should indent two or three spaces at each level of indentation. You want to see the indentation clearly, but you want to be able to use most of the line for the Java statement.

If a statement does not fit on one line, you can write it on two or more lines. However, when you write a single statement on more than one line, you should indent the successive lines more than the first line, as in the following example:

```
System.out.println("The volume of a sphere whose radius is " +
                   radius + " inches is " + volume +
                   " cubic inches.");
```

Ultimately, you need to follow the rules for indenting—and for programming style in general—given by your instructor or project manager. In any event, you should indent consistently within any one program.

Comments

A.3 The documentation for a program describes what the program does and how it does it. The best programs are **self-documenting**. That is, their clean style and well-chosen names make the program's purpose and logic clear to any programmer who reads the program. Although you should strive for such self-documenting programs, your programs will also need a bit of explanation to make them completely clear. This explanation can be given in the form of **comments**.

Comments are notations in your program that help a person understand the program, but that are ignored by the compiler. Many text editors automatically highlight comments in some way, such as showing them in color. In Java, there are several ways of forming comments.

Single-Line Comments

A.4 To write a comment on a single line, begin the comment with two slashes `//`. Everything after the slashes until the end of the line is treated as a comment and is ignored by the compiler. This form is handy for short comments, such as

```
String sentence; // Spanish version
```

If you want a comment of this kind to span several lines, each line must contain the symbols `//`.

Comment Blocks

A.5 Anything written between the matching pair of symbols `/*` and `*/` is a comment and is ignored by the compiler. This form is not typically used to document a program, however. Instead, it is handy during debugging to temporarily disable a group of Java statements. Java programmers do use the pair `/**` and `*/` to delimit comments written in a certain form, as you will see in Segment A.7.

When to Write Comments

A.6 It is difficult to explain just when you should write a comment. Too many comments can be as bad as too few. Too many comments can hide the really important ones. Too few comments can leave a reader baffled by things that were obvious to you. Just remember that you also will read your program. If you read it next week, will you remember what you did just now?

Every program file should begin with an explanatory comment. This comment should give all the important information about the file: what the program does, the name of the author, how to contact the author, the date that the file was last changed, and in a course, what the assignment is. Every method should begin with a comment that explains the method.

Within methods, you need comments to explain any nonobvious details. Notice the poor comments on the following declarations of the variables `radius` and `area`:

```
double radius; // The radius
double area;   // The area
```

Because we chose descriptive variable names, these comments are obvious. But rather than simply omitting these comments, can we write something that is not obvious? What units are used for the radius? Inches? Feet? Meters? Centimeters? We will add a comment that gives this information, as follows:

```
double radius; // In inches
double area;   // In square inches
```

Java Documentation Comments

A.7 The Java language comes with a utility program named **`javadoc`** that will generate HTML documents that describe your classes. These documents tell people who use your program or class how to use it, but they omit all the implementation details.

The program `javadoc` extracts the header for your class, the headers for all public methods, and comments that are written in a certain form. No method bodies and no private items are extracted.

For `javadoc` to extract a comment, the comment must satisfy two conditions:

- The comment must occur immediately before a public class definition or the header of a public method.
- The comment must begin with `/**` and end with `*/`.

Segment A.12 contains an example of a comment in this style.

You can insert HTML commands in your comments so that you gain more control over —`javadoc`, but that is not necessary and we have not done so in this book.

A.8 **Tags.** Comments written for `javadoc` usually contain special **tags** that identify such things as the programmer and a method's parameters and return value. Tags begin with the symbol `@`. We will describe only four tags in this appendix.

The tag `@author` identifies the programmer's name and can appear in all classes and interfaces. One tag can list one name or several comma-separated names. You can write several such tags—one per author—or omit this tag entirely. Because the documentation produced by `javadoc` ignores the `@author` tag, some organizations do not use it.

The other tags of interest to us are used with methods. They must appear in the following order within a comment that precedes a method's header:

```
@param
@return
@throws
```

We will describe each of these tags next.

A.9 **The `@param` tag.** You must write a `@param` tag for every parameter in a method. You should list these tags in the order in which the parameters appear in the method's header. After the `@param` tag, you give the name and description of the parameter. Typically, you use a phrase instead of a sentence to describe the parameter, and you mention the parameter's data type first. Do not use punctuation between the parameter name and its description, as `javadoc` inserts one dash when creating its documentation.

For example, the comments

```
@param code      The character code of the ticket category.
@param customer  The string that names the customer.
```

will produce the following lines in the documentation:

```
code - The character code of the ticket category.
customer - The string that names the customer.
```

A.10 **The `@return` tag.** You must write a `@return` tag for every method that returns a value, even if you have already described the value in the method's description. Try to say something more specific about this value here. This tag must come after any `@param` tags in the comment. Do not use this tag for `void` methods and constructors.

A.11 **The `@throws` tag.** Next, if a method can throw a checked exception, you name it by using a `@throws` tag, even if the exception also appears in a `throws` clause in the method's header. You can list unchecked exceptions if a client might reasonably catch them. (As you will learn in Appendix B, a client of a class is a program component that uses the class.) Include a `@throws` tag for each exception, and list them alphabetically by name.

A.12 **Example.** Here is a sample `javadoc` comment for a method. We usually begin such comments with a brief description of the method's purpose. This is our convention; `javadoc` has no tag for it.

```
/** Adds a new entry to a roster.
    @param newEntry     The object to be added to the roster.
    @param newPosition  The position of newEntry within the roster.
    @return  True if the addition is successful.
    @throws  RosterException if newPosition < 1 or newPosition > 1 + the length
             of the roster. */
public boolean add(Object newEntry, int newPosition) throws RosterException
```

The documentation that `javadoc` prepares from the previous comment appears as follows:

add

```
public boolean add(java.lang.Object newEntry,
                   int newPosition)
            throws RosterException
```

Adds a new entry to a roster.

Parameters:

```
newEntry - The object to be added to the roster.
newPosition - The position of newEntry within the roster.
```

Returns:

True if the addition is successful.

Throws:

`RosterException` - if newPosition < 1 or newPosition $> 1 +$ the length of the roster.

To save space in this book, we sometimes omit portions of a comment that we would include in our actual programs. For example, some methods might have only a description of their purpose, and some might have only a `@return` tag. Note that `javadoc` accepts these abbreviated comments.

Further details about `javadoc` are available at `docs.oracle.com/javase/8/docs/technotes/tools/unix/javadoc.html`.