Chapter 2

Comments, Conditions, and Assertions

Concepts:
- ▷ Preconditions
- ▷ Postconditions
- ▷ Assertions
- ▷ Copyrighting code

```
/* This is bogus code.
Wizards are invited to improve it.  */
```
—Anonymous

CONSIDER THIS: WE CALL OUR PROGRAMS "CODE"! The features of computer languages, including Java, are designed to help express algorithms in a manner that a machine can understand. Making a program run more efficiently often makes it less understandable. If language design was driven by the need to make the program readable by programmers, it would be hard to argue against programming in English.

Okay, perhaps French!

A *comment* is a carefully crafted piece of text that describes the state of the machine, the use of a variable, or the purpose of a control construct. Many of us, though, write comments for the same reason that we exercise: we feel guilty. You feel that, if you do not write comments in your code, you "just *know*" something bad is going to happen. Well, you are right. A comment you write today will help you out of a hole you dig tomorrow.

Ruth Krauss: "A hole is to dig."

All too often comments are hastily written after the fact, to help understand the code. The time spent thinking seriously about the code has long since passed, and the comment might not be right. If you write comments beforehand, while you are designing your code, it is more likely your comments will describe what you want to do as you carefully think it out. Then, when something goes wrong, the comment is there to help you figure out the code. In fairness, the code and the comment have a symbiotic relationship. Writing one or the other does not really feel complete, but writing both provides you with the redundancy of concept: one lucid and one as clear as Java.

The one disadvantage of comments is that, unlike code, they cannot be checked. Occasionally, programmers come across comments such as "If you think you understand this, you don't!" or "Are you reading this?" One could, of course, annotate programs with mathematical formulas. As the program is compiled, the mathematical comments are distilled into very concise descriptions of what should be going on. When the output from the program's code does not match the result of the formula, something is clearly wrong with your logic. But *which* logic? The writing of mathematical comments is a level of detail most programmers would prefer to avoid.

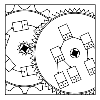
Semiformal convention: a meeting of tie haters.

A compromise is a semiformal convention for comments that provide a reasonable documentation of *when* and *what* a program does. In the code associated with this book, we see one or two comments for each method or function that describe its purpose. These important comments are the *precondition* and *postcondition*.

2.1 Pre- and Postconditions

The *precondition* describes, as succinctly as possible in your native tongue, the conditions under which a method may be called and expected to produce correct results. Ideally the precondition expresses the *state* of the program. This state is usually cast in terms of the parameters passed to the routine. For example, the precondition on a square root function might be

```
// pre: x is nonnegative
```



sqrt

The authors of this square root function expect that the parameter is not a negative number. It is, of course, legal in Java to call a function or method if the precondition is not met, but it might not produce the desired result. When there is no precondition on a procedure, it may be called without failure.

The *postcondition* describes the state of the program once the routine has been completed, *provided the precondition was met*. Every routine should have some postcondition. If there were not a postcondition, then the routine would not change the state of the program, and the routine would have no effect! Always provide postconditions.

Pre- and postconditions do not force you to write code correctly. Nor do they help you find the problems that *do* occur. They can, however, provide you with a uniform method for documenting the programs you write, and they require more thought than the average comment. More thought put into programs lowers your average blood pressure and ultimately saves you time you might spend more usefully playing outside, visiting museums, or otherwise bettering your mind.

2.2 Assertions

In days gone by, homeowners would sew firecrackers in their curtains. If the house were to catch fire, the curtains would burn, setting off the firecrackers. It was an elementary but effective fire alarm.

And the batteries never needed replacing.

An *assertion* is an assumption you make about the state of your program. In Java, we will encode the assertion as a call to a function that verifies the state of the program. That function does nothing if the assertion is true, but it halts your program with an error message if it is false. It is a firecracker to sew in your program. If you sew enough assertions into your code, you will get an early warning if you are about to be burned by your logic.

Principle 5 *Test assertions in your code.*

The `Assert` class provides several functions to help you test the state of your program as it runs:

Assert

```
public class Assert
{
    static public void pre(boolean test, String message)
    // pre: result of precondition test
    // post: does nothing if test true, otherwise abort w/message

    static public void post(boolean test, String message)
    // pre: result of postcondition test
    // post: does nothing if test true, otherwise abort w/message

    static public void condition(boolean test, String message)
    // pre: result of general condition test
    // post: does nothing if test true, otherwise abort w/message

    static public void invariant(boolean test, String message)
    // pre: result of an invariant test
    // post: does nothing if test true, otherwise abort w/message

    static public void fail(String message)
    // post: throws error with message
}
```

Each of `pre`, `post`, `invariant`, and `condition` methods test to see if its first argument—the assertion—is true. The `message` is used to indicate the condition tested by the assertion. Here's an example of a check to make sure that the precondition for the `sqrt` function was met:

```
public static double sqrt(double x)
// pre: x is nonnegative
// post: returns the square root of x
{
    Assert.pre(x >= 0,"the value is nonnegative.");
    double guess = 1.0;
    double guessSquared = guess * guess;

    while (Math.abs(x-guessSquared) >= 0.00000001) {
        // guess is off a bit, adjust
        guess += (x-guessSquared)/2.0/guess;
        guessSquared = guess*guess;
    }
    return guess;
}
```

Should we call `sqrt` with a negative value, the assertion fails, the message is printed out, and the program comes to a halt. Here's what appears at the display:

```
structure.FailedPrecondition:
Assertion that failed: A precondition: the value is nonnegative.
        at Assert.pre(Assert.java:17)
        at sqrt(examples.java:24)
        at main(examples.java:15)
```

The first two lines of this message indicate that a precondition (that `x` was nonnegative) failed. This message was printed within `Assert.pre` on line 17 of the source, found in `Assert.java`. The next line of this *stack trace* indicates that the call to `Assert.pre` was made on line 24 of `examples.java` at the start of the `sqrt` function. This is the first line of the `sqrt` method. The problem is (probably) on line 15 of the main procedure of `examples.java`. Debugging our code should probably start in the `main` routine.

Beginning with Java 1.4, assertion testing is part of the formal Java language specification. The `assert` keyword can be used to perform many of the types of checks we have described. If, however, you are using an earlier version of Java, or you expect your users may wish to use a version of Java before version 1.4, you may find the `Assert` class to be a more portable approach to the testing of the conditions of one's code. A feature of language-based assertion testing is that the tests can be automatically removed at compile time when one feels secure about the way the code works. This may significantly improve performance of classes that heavily test conditions.

2.3 Craftsmanship

If you *really* desire to program well, a first step is to take pride in your work—pride enough to sign your name on everything you do. Through the centuries, fine furniture makers signed their work, painters finished their efforts by dabbing on their names, and authors inscribed their books. Programmers should stand behind their creations.

Computer software has the luxury of immediate copyright protection—it is a protection against piracy, and a modern statement that you stand behind the belief that what you do is worth fighting for. If you have crafted something as best you can, add a comment at the top of your code:

```
// Image compression barrel for downlink to robotic cow tipper.
// (c) 2001, 2002 duane r. bailey
```

If, of course, you *have* stolen work from another, avoid the comment and consider, heavily, the appropriate attribution.

2.4 Conclusions

Effective programmers consider their work a craft. Their constructions are well considered and documented. Comments are not necessary, but documentation makes working with a program much easier. One of the most important comments you can provide is your name—it suggests you are taking credit *and* responsibility for things you create. It makes our programming world less anonymous and more humane.

Special comments, including conditions and assertions, help the user and implementor of a method determine whether the method is used correctly. While it is difficult for compilers to determine the "spirit of the routine," the implementor is usually able to provide succinct checks of the sanity of the function. Five minutes of appropriate condition description and checking provided by the implementor can prevent hours of debugging by the user.

I've done my time!

Self Check Problems

Solutions to these problems begin on page 442.

2.1 Why is it necessary to provide pre- and postconditions?

2.2 What can be assumed if a method has no precondition?

2.3 Why is it not possible to have a method with no postcondition?

2.4 Object orientation allows us to hide unimportant details from the user. Why, then, must we put pre- and postconditions on hidden code?

Problems

Solutions to the odd-numbered problems begin on page 457.

2.1 What are the pre- and postconditions for the `length` method of the `java.lang.String` class?

2.2 What are the pre- and postconditions for `String`'s `charAt` method?

2.3 What are the pre- and postconditions for `String`'s `concat` method?

2.4 What are the pre- and postconditions for the `floor` function in the `java.lang.Math` class?

2.5 Improve the comments on an old program.

2.6 Each of the methods of `Assert` (`pre`, `post`, and `condition`) takes the same parameters. In what way do the methods function differently? (Write a test program to find out!)

2.7 What are the pre- and postconditions for `java.lang.Math.asin` class?

2.5 Laboratory: Using Javadoc Commenting

Objective. To learn how to generate formal documentation for your programs.

Discussion. The Javadoc program[1] allows the programmer to write comments in a manner that allows the generation web-based documentation. Programmers generating classes to be used by others are particularly encouraged to consider using Javadoc-based documentation. Such comments are portable, web-accessible, and they are directly extracted from the code.

In this lab, we will write documentation for an extended version of the `Ratio` class we first met in Chapter 1.

Comments used by Javadoc are delimited by a `/** */` pair. Note that there are two asterisks in the start of the comment. Within the comment are a number of keywords, identified by a leading "at-sign" (`@`). These keywords identify the purpose of different comments you right. For example, the text following an `@author` comment identifies the programmer who originally authored the code. These comments, called Javadoc comments, appear *before* the objects they document. For example, the first few lines of the `Assert` class are:

```
package structure;
/**
 * A library of assertion testing and debugging procedures.
 * <p>
 * This class of static methods provides basic assertion testing
 * facilities.  An assertion is a condition that is expected to
 * be true at a certain point in the code.  Each of the
 * assertion-based routines in this class perform a verification
 * of the condition and do nothing (aside from testing side-effects)
 * if the condition holds.  If the condition fails, however, the
 * assertion throws an exception and prints the associated message,
 * that describes the condition that failed.  Basic support is
 * provided for testing general conditions, and pre- and
 * postconditions. There is also a facility for throwing a
 * failed condition for code that should not be executed.
 * <p>
 * Features similar to assertion testing are incorporated
 * in the Java 2 language beginning in SDK 1.4.
 * @author duane a. bailey
 */
public class Assert
{
   . . .
}
```

For each class you should provide any class-wide documentation, including `@author` and `@version`-tagged comments.

[1] Javadoc is a feature of command-line driven Java environments. Graphical environments likely provide Javadoc-like functionality, but pre- and postcondition support may not be available.

Within the class definition, there should be a Javadoc comment for each instance variable and method. Typically, Javadoc comments for instance variables are short comments that describe the role of the variable in supporting the class state:

```
/**
 * Size of the structure.
 */
int size;
```

Comments for methods should include a description of the method's purpose. A comment should describe the purpose of each parameter (`@param`), as well as the form of the value returned (`@return`) for function-like methods. Programmers should also provide pre- and postconditions using the `@pre` and `@post` keywords.[2] Here is the documentation for a square root method.

```
/**
 *
 * This method computes the square root of a double value.
 * @param x The value whose root is to be computed.
 * @return The square root of x.
 * @pre x >= 0
 * @post computes the square root of x
 */
```

To complete this lab, you are to

1. Download a copy of the `Ratio.java` source from the *Java Structures* website. This version of the `Ratio` class does not include full comments.
2. Review the code and make sure that you understand the purpose of each of the methods.
3. At the top of the `Ratio.java` file, place a Javadoc comment that describes the class. The comment should describe the features of the class and an example of how the class might be used. Make sure that you include an `@author` comment (use your name).
4. Run the documentation generation facility for your particular environment. For Sun's Java environment, the Javadoc command takes a parameter that describes the location of the source code that is to be documented:

```
javadoc prog.java
```

[2] In this book, where there are constraints on space, the pre- and postconditions are provided in non-Javadoc comments. Code available on the web, however, is uniformly commented using the Javadoc comments. Javadoc can be upgraded to recognize pre- and postconditions; details are available from the *Java Structures* website.

The result is an `index.html` file in the current directory that contains links to all of the necessary documentation. View the documentation to make sure your description is formatted correctly.

5. Before each instance variable write a short Javadoc comment. The comment should begin with `/**` and end with `*/`. Generate and view the documentation and note the change in the documentation.

6. Directly before each method write a Javadoc comment that includes, at a minimum, one or two sentences that describe the method, a `@param` comment for each parameter in the method, a `@return` comment describing the value returned, and a `@pre` and `@post` comment describing the conditions.

   Generate and view the documentation and note the change in the documentation. If the documentation facility does not appear to recognize the `@pre` and `@post` keywords, the appropriate Javadoc doclet software has not been installed correctly. More information on installation of the Javadoc software can be found at the *Java Structures* website.

Notes: