Assertions and Loop Invariants
CSC 207, “Algorithms and Object-Oriented Design”
Department of Computer Science
Grinnell College
October 10, 2018

Assertions

In computer programming, an assertion is a statement that tests whether a precondition for the successful completion of some operation has been met, and interrupts the execution of the program if it is not. It can be placed at the beginning of a method to ensure (for example) that appropriate arguments have been supplied to the method, or in the middle of a method, perhaps immediately in front of a loop, to ensure that the statements that precede it have established the preconditions for the successful execution of the loop. An assertion can also be placed immediately before a return-statement to confirm that the value that is about to be returned satisfies the postconditions of the method.

Java supports assertions in the form of assert-statements, which have two forms. The first consists of just the keyword assert, a Boolean expression, and a semicolon. When the assert-statement is executed, the Boolean expression is evaluated. If its value is true, control simply passes to the next statement, but if it is false, an AssertionError is raised.

The second form is similar, except that a colon is placed after the Boolean expression and a second expression is placed between the colon and the semicolon. In this case, if the value of first expression is false, the second expression is evaluated, and its value is packaged into the AssertionError. Typically the value of the second expression is a String, and that String will appear in the error message that the run-time system prints after the program has crashed.

By default, assert-statements are not executed at all; the virtual machine skips over them. To arrange for the virtual machine to execute them, you must enable assertions. On the command line, you do this by adding the option -ea (“enable assertions”) to the java command, just before the name of the class you want to execute. In Eclipse, select Run Configurations from the Run menu, click on the Arguments tab, and put the option -ea into the text field labelled “VM Arguments.”

Loop Invariants

A loop invariant is a condition that is true at the beginning of every execution of the body of a loop and also true at the end of the last execution of the body of the loop. Typically the body of the loop contains some statements that cause the invariant to be broken, followed by some statements that restore the invariant for the next execution of the loop body. For instance, in the code

```java
    total = 0;
    for( int i = 0; i < N; i++ )
        total += lst.get(i);
```
(from page 255 of the textbook) the statement that total is equal to the sum of the first i elements of lst is a loop invariant. It is (vacuously) true just before the first execution of the loop body, when total is 0, i is 0, and no elements of the list have yet been considered. If lst has, say, seven elements, the loop invariant is still true just before the sixth execution of the loop body, when i is 5, because the first five elements of the list (in positions 0 through 4) will have been successively added to total by that time. Finally, the loop invariant is also true at the end of the seventh execution of the loop, when i is 7, and all seven elements of lst have been added to total.

The importance of a loop invariant, especially in cases where the programmer does not know in advance how many times the loop body will execute in any particular run of the code, is that it serves as a kind of checkpoint for verifying that the code is correct, analogous to the preconditions
and postconditions of a method. Formulating and checking the loop invariant for a tricky or elaborate loop makes it possible to program the details inside the loop with greater confidence and less chance for making errors.

Because the loop invariant is true at the end of the last iteration of the loop body, it can also be used (along with the negation of the loop entry test) as the precondition for the subsequent steps of the method. In the example above, we know that we can safely return total as the sum of the elements of the list immediately after leaving the for-loop, because the loop invariant tells us that total is the sum of the first \( i \) elements of the list, and the fact that the loop entry test is now false tells us that \( i \) is no longer less than \( N \), that is, that we have seen all of the elements of the list.