Program Correctness

Defensive programming is like defensive driving: you take the attitude that you never know what the other drivers – or methods – are going to do, so you do what you can to prevent other people's mistakes from hurting you. When programming, this means checking that parameters passed to your methods are valid and that the results of your methods are reasonable.

Invariants

Recall that a *predicate* is any statement that can be true or false. An *invariant* is a predicate that expected to be true at a particular point in your program but is not directly guaranteed by the language. For example, in the following code we expect `i % 3 == 2` to be true in the last `else` block of the `if` statement, but that's not guaranteed in the same way that the first two blocks have guarantees. If we want to increase our confidence about our program's correctness (and narrow down the set of possible bugs), we'd like to have Java check our invariants for us, at least until we're satisfied in the correctness of our code. (In fact, this invariant is violated if `i` is ever negative.)

```
if (i % 3 == 0) {
  ...
} else if (i % 3 == 1) {
  ...
} else { // i % 3 == 2 here
  ...
}
```

There are several kinds of invariants, and Java provides different mechanisms to check them.

Preconditions of Public Methods

A *precondition* is an invariant that says something is true immediately before a method (or some other unit of code) runs. Although we haven't been requiring it in this course, preconditions of public methods should be carefully specified in the documentation of a function. For example, in our implementation of `HashSet`, a precondition of `add(o)` was that `o != null`. The proper way to enforce preconditions of public methods in Java is to throw a runtime exception (such as `IllegalArgumentException`, `IndexOutOfBoundsException`, or `NullPointerException`). Recall that a runtime exception is a subclass of `RuntimeException` and that such exceptions do not need to be declared in the `throws` clause of a method.

For example, our `HashSet` `add` method looked like:

```
public boolean add(Object o) {
  if (o == null) throw new NullPointerException("nulls disallowed");
  if (size == capacity)
    throw new RuntimeException("hash table full");
  ...
}
```

Class Invariants

A *class invariant* (the name isn't important) is an invariant that applies to all instances of a class at all times, except when an instance is transitioning from one state to another. For example, in a priority queue implemented as a sorted list, a class invariant is that the list is always sorted in increasing order by the keys of the elements.

EXERCISE: Make a list of other class invariants we've seen. Think of lists, binary trees, binary search trees, hash tables, stacks, queues, and heaps.

EXERCISE: For each of the invariants you've listed, would you be able to write a method that determines whether the invariant is properly satisfied? Write pseudocode for a method to check the invariant of a binary seach tree.

Although we're using Java 1.3 in this course, Java 1.4 provides an `assert` statement that can be used to verify such invariants. The basic syntax is

```
assert e;
```

where `e` is a boolean expression. If the expression evaluates to `false`, the `assert` statement will throw an `AssertionError`. Recall that `Errors`, as opposed to `Exceptions`, are normally not caught. This is appropriate here; if your queue implementation expects the list to be sorted, and you suddenly discover that the list isn't sorted, there's really no graceful way to recover because you don't know when the problem occurred.

Preconditions and Postconditions

Naturally, a *postcondition* is an invariant that is expected to be true at the end of a method (or other unit of code). For example, in a method written to sort an array, a postcondition would be that the array is sorted.

Any class invariants would normally be *both* preconditions and postconditions of *all* of the public methods in your class. For example, the heap order property should be satisfied before and after every `enqueue` and `dequeue` operation of our heap implementation. To check this I would write a method `isOrderPropertySatisfied`, and then at the beginning and end of `enqueue` and `dequeue` I would write

```
assert isOrderPropertySatisfied();
```

I would also place this assertion at the end of the constructor.

EXERCISE: Why would it be less appropriate to throw runtime exceptions when such a class invariant is found violated?

For the same reason, `assert` *statements are the proper way to handle preconditions of private methods and postconditions (of any methods).*

If your language lacks built-in assertions, you can always fake it with an `if` and a `throw`. But an important feature of the assertion facilities provided by most languages is that they can be turned off. Some assertions can be very expensive to check (such as checking that your list is sorted). Once you are satisfied that your program works, you can disable assertions, and then they won't take any time at runtime. In Java 1.4, you can disable assertions with the `-disableassertions` option to the Java runtime. (More advanced usage lets you selectively enable and disable assertions.)

Better than disabling assertions when you're done is to comment out trivial or expensive assertions but leave others in the code so that if a bug remains the program at least won't continue running blindly.

Because assertions can be disabled, you should never place in an `assert` statement any code that affects the correctness of your method.

How does this relate to the unit testing we did?

The purpose of unit testing is to verify the *externally visible* behavior of a class (or a larger unit of code). Unit testing is also called *functional testing*. Recall that we wrote unit tests in another class (such as `BoggleBoardTest`), so it didn't have access to private variables. All a unit test can do is call public constructors and public methods. Unit testing is also called *black-box testing* because the test doesn't care about how the class is implemented; we could completely change the implementation of `BoggleBoard` so that different class invariants apply, and our `BoggleBoardTest` should still work.

Assertions and other defensive programming techniques are a finer level of testing because we use them to verify the state of the program at individual lines within a class. If we change the implementation of `BoggleBoard`, we'd have to change the assertions because the class invariants would be different.

Both forms of testing are important.

Loop Invariants

Loop invariants are a special kind of invariant. They state what is true before and after each iteration of a loop. They are usually more subtle than the other kinds of invariants so they tend to require more thought per line of code. But they are absolutely crucial – if your loop doesn't properly maintain a reasonable loop invariant (even if you don't know what it is), your loop won't work.

Let's start with an iterative factorial function:

```
int fact(int n) {
  int r = 1, i = 1;
  while (i <= n) {
    r *= i;
    i++;
  }
  return r;
}
```

In order for `result` to have a meaningful value when the loop terminates, it should have a meaningful value after each iteration. We express this as the *loop invariant*. In this case, the loop invariant is that $r = (i-1)!$. The following table shows the values of r and i after each iteration:

After iteration	r	i
0	1	1
1	1	2
2	2	3
3	6	4
4	24	5
n	$n!$	$n+1$

As long as $r = (i-1)!$ before each iteration of the loop, the loop will update r and i so that i is one larger and $r = (i-1)!$ still. So when the loop terminates, $r = (i-1)!$ and $i = n+1$.

Let's try another example. Recall the `isIncreasing` method we wrote on `Lists` during the first week. Since that version was recursive, we'll write an iterative version that operates on arrays of integers:

```
boolean isIncreasing(int a[]) {
  int i = 0; // index into array a
  int m = Integer.MIN_VALUE; // maximum seen so far
  while (i < a.length) {
    if (a[i] <= m) {
      m = a[i];
      i++;
    } else {
      return false;
    }
  }
  return true;
}
```

What is invariant over all iterations? After each iteration we know that a particular part of the array is sorted. In particular, $a[0 \ldots i-1]$ is in increasing order with a maximum value no larger than m. (I don't say "equal to m" because that's not true after 0 iterations.) How can we be sure that we don't have an off-by-one error in the loop? Because after the last iteration of the loop i is equal to the length of the array, so $a[0 \ldots i-1]$ is the entire array.

Having a carefully written loop invariant is useful because it can guide you in writing your loops. In this case, if we knew before we had written the loop that this would be our loop invariant and that i would be incremented through the array with each iteration, we would know exactly what we had to do within each iteration. We'd know that before each iteration $a[0 \ldots i-1]$ would be sorted with a maximum value no larger than m. So we'd just have to check that $a[i]$ doesn't violate the order and then update m and increment i.

EXERCISE: Now it's your turn. Pretend that `Math.pow` doesn't exist and you have to write an exponentiation function on integers. Pick a suitable loop invariant and implement the function. Check that your loop invariant is initially true, that each iteration preserves the invariant, and that when the loop terminates the invariant will give you the answer you want.

EXERCISE: Look at my first implementation (given in class). Use the same steps to identify and correct the bug.

EXERCISE: What's the asymptotic complexity of the algorithm we implemented?

EXERCISE: Let's improve the algorithm so that our function runs much faster. We can use the equivalence $a^b = (a^2)^{b/2}$ when b is even. Write a recursive version first so you understand the idea. When b is odd, use the same idea as before.

EXERCISE: Find an upper bound on the number of recursive calls made. What's the asymptotic complexity of this new algorithm?

EXERCISE: Now let's write the iterative version of this algorithm. This is tricky. Hint: Let the invariant be $a^b = ra_2^i$, and incrementally *reduce* i to 0. Then the result is just r.

Resources

Steve McConnell's classic book *Code Complete* is a fantastic resource for learning how to construct better code in any language. It is extremely readable, and I highly recommend it.

David Gries' book *The Science of Programming* or his handout *Correctness of Programs* (`http://www.cs.cornell.edu/Courses/cs211/2001fa/handouts/correctness.pdf`) contain more than you'll ever want to know about invariants.

Parts of this handout are based on the Java assertion documentation at `http://java.sun.com/j2se/1.4/docs/guide/lang/assert.html`.