Appendix A

Program development plan[†]

If you are spending a lot of time debugging, it is probably because you do not have an effective **program development plan**.

A typical, bad program development plan goes something like this:

1. Write an entire method.
2. Write several more methods.
3. Try to compile the program.
4. Spend an hour finding syntax errors.
5. Spend an hour finding run time errors.
6. Spend three hours finding semantic errors.

The problem, of course, is the first two steps. If you write more than one method, or even an entire method, before you start the debugging process, you are likely to write more code than you can debug.

If you find yourself in this situation, the *only* solution is to remove code until you have a working program again, and then gradually build the program back up. Beginning programmers are often unwilling to do this, because their carefully crafted code is precious to them. To debug effectively, you have to be ruthless!

Here is a better program development plan:

1. Start with a working program that does something visible, like printing something.
2. Add a small number of lines of code at a time, and test the program after every change.

[†]This appendix appears in Allen B. Downey, *How to Think Like a Computer Scientist: Java Version*, 2002.

3. Repeat until the program does what it is supposed to do.

After every change, the program should produce some visible effect that demonstrates the new code. This approach to programming can save a lot of time. Because you only add a few lines of code at a time, it is easy to find syntax errors. Also, because each version of the program produces a visible result, you are constantly testing your mental model of how the program works. If your mental model is erroneous, you will be confronted with the conflict (and have a chance to correct it) before you have written a lot of erroneous code.

One problem with this approach is that it is often difficult to figure out a path from the starting place to a complete and correct program.

I will demonstrate by developing a method called `isIn` that takes a String and a Vector, and that returns a boolean: `true` if the String appears in the list and `false` otherwise.

1. The first step is to write the shortest possible method that will compile, run, and do something visible:

```
public static boolean isIn (String word, Vector v) {
    System.out.println ("isIn");
    return false;
}
```

   Of course, to test the method we have to invoke it. In `main`, or somewhere else in a working program, we need to create a simple test case.

   We'll start with a case where the String appears in the vector (so we expect the result to be `true`).

```
public static void main (String[] args) {
    Vector v = new Vector ();
    v.add ("banana");
    boolean test = isIn ("banana", v);
    System.out.println (test);
}
```

   If everything goes according to plan, this code will compile, run, and print the word `isIn` and the value `false`. Of course, the answer isn't correct, but at this point we know that the method is getting invoked and returning a value.

   In my programming career, I have wasted way too much time debugging a method, only to discover that it was never getting invoked. If I had used this development plan, it never would have happened.

2. The next step is to check the parameters the method receives.

```
public static boolean isIn (String word, Vector v) {
    System.out.println ("isIn looking for " + word);
    System.out.println ("in the vector " + v);
    return false;
}
```

The first print statement allows us to confirm that `isIn` is looking for the right word. The second statement prints a list of the elements in the vector.

To make things more interesting, we might add a few more elements to the vector:

```
public static void main (String[] args) {
    Vector v = new Vector ();
    v.add ("apple");
    v.add ("banana");
    v.add ("grapefruit");
    boolean test = isIn ("banana", v);
    System.out.println (test);
}
```

Now the output looks like this:

```
isIn looking for banana
in the vector [apple, banana, grapefruit]
```

Printing the parameters might seem silly, since we know what they are supposed to be. The point is to confirm that they are what we think they are.

3. To traverse the vector, we can take advantage of the code from Section 17.10. In general, it is a great idea to reuse code fragments rather than writing them from scratch.

```
public static boolean isIn (String word, Vector v) {
    System.out.println ("isIn looking for " + word);
    System.out.println ("in the vector " + v);

    for (int i=0; i<v.size(); i++) {
        System.out.println (v.get(i));
    }
    return false;
}
```

Now when we run the program it prints the elements of the vector one at a time. If all goes well, we can confirm that the loop examines all the elements of the vector.

4. So far we haven't given much thought to what this method is going to do. At this point we probably need to figure out an algorithm. The simplest algorithm is a linear search, which traverses the vector and compares each element to the target word.

   Happily, we have already written the code that traverses the vector. As usual, we'll proceed by adding just a few lines at a time:

```
public static boolean isIn (String word, Vector v) {
    System.out.println ("isIn looking for " + word);
    System.out.println ("in the vector " + v);

    for (int i=0; i<v.size(); i++) {
        System.out.println (v.get(i));
        String s = (String) v.get(i);
        if (word.equals (s)) {
            System.out.println ("found it");
        }
    }
    return false;
}
```

   As always, we use the `equals` method to compare Strings, not the `==` operator!

   Again, I added a print statement so that when the new code executes it produces a visible effect.

5. At this point we are pretty close to working code. The next change is to return from the method if we find what we are looking for:

```
public static boolean isIn (String word, Vector v) {
    System.out.println ("isIn looking for " + word);
    System.out.println ("in the vector " + v);

    for (int i=0; i<v.size(); i++) {
        System.out.println (v.get(i));
        String s = (String) v.get(i);
        if (word.equals (s)) {
            System.out.println ("found it");
            return true;
        }
    }
    return false;
}
```

   If we find the target word, we return `true`. If we get all the way through the loop without finding it, then the correct return value is `false`.

If we run the program at this point, we should get

```
isIn looking for banana
in the vector [apple, banana, grapefruit]
apple
banana
found it
true
```

6. The next step is to make sure that the other test cases work correctly. First, we should confirm that the method returns `false` if the word in not in the vector.

   Then we should check some of the typical troublemakers, like an empty vector (one with size 0) and a vector with a single element. Also, we might try giving the method an empty String.

   As always, this kind of testing can help find bugs if there are any, but it can't tell you if the method is correct.

7. The penultimate step is to remove or comment out the print statements.

```
public static boolean isIn (String word, Vector v) {
    for (int i=0; i<v.size(); i++) {
        System.out.println (v.get(i));
        String s = (String) v.get(i);
        if (word.equals (s)) {
            return true;
        }
    }
    return false;
}
```

   Commenting out the print statements is a good idea if you think you might have to revisit this method later. But if this is the final version of the method, and you are convinced that it is correct, you should remove them.

   Removing the comments allows you to see the code most clearly, which can help you spot any remaining problems.

   If there is anything about the code that is not obvious, you should add comments to explain it. Resist the temptation to translate the code line by line. For example, no one needs this:

```
        // if word equals s, return true
        if (word.equals (s)) {
            return true;
        }
```

You should use comments to explain non-obvious code, to warn about conditions that could cause errors, and to document any assumptions that are built into the code. Also, before each method, it is a good idea to write an abstract description of what the method does.

8. The final step is to examine the code and see if you can convince yourself that it is correct.

   At this point we know that the method is syntactically correct, because it compiles.

   To check for run time errors, you should find every statement that can cause an error and figure out what conditions cause the error.

   The statements in this method that can produce a run time error are:

`v.size()`	if `v` is `null`.
`word.equals (s)`	if `word` is `null`.
`(String) v.get(i)`	if `v` is `null` or `i` is out of bounds, or the ith element of `v` is not a String.

   Since we get `v` and `word` as parameters, there is no way to avoid the first two conditions. The best we can do is check for them.

```
public static boolean isIn (String word, Vector v) {
    if (v == null || word == null) return false;
    for (int i=0; i<v.size(); i++) {
        System.out.println (v.get(i));
        String s = (String) v.get(i);
        if (word.equals (s)) {
            return true;
        }
    }
    return false;
}
```

   In general, it is a good idea for methods to make sure their parameters are legal.

   The structure of the `for` loop ensures that `i` is always between 0 and `v.size()-1`. But there is no way to ensure that the elements of `v` are Strings. On the other hand, we can check them as we go along. The `instanceof` operator checks whether an object belongs to a class.

```
    Object obj = v.get(i);
    if (obj instanceof String) {
        String s = (String) v.get(i);
    }
```

This code gets an object from the vector and checks whether it is a String. If it is, it performs the typecast and assigns the String to `s`.

As an exercise, modify `isIn` so that if it finds an element in the vector that is not a String, it skips to the next element.

If we handle all the problem conditions, we can prove that this method will not cause a run time error.

We haven't proven yet that the method is semantically correct, but by proceeding incrementally, we have avoided many possible errors. For example, we already know that the method is getting parameters correctly and that the loop traverses the entire vector. We also know that it is comparing Strings successfully, and returning `true` if it finds the target word. Finally, we know that if the loop exists, the target word cannot be in the vector.

Short of a formal proof, that is probably the best we can do.