CS61B Note 2: Fall 2019

Kevin Moy

Abstract

Now that we've established the fundamentals we need to know with the Java language, we can immediately dive into how to program some algorithms and solve different problems. More specifically, we'll outline a process to do so.

1 Program Construction: Prime Number Algorithm

In lecture, we saw the construction of a Java program that printed all the prime numbers up to a user-specified limit. Let's break down the steps needed to construct such a program. I've devised a simple 3-step PSI process: planning, skeleton, and implementation. PSI will be very useful for getting started on the upcoming projects in this class, and on programming projects in general.

1.1 Step 1: Planning

The first step is to lay out as much as possible about our program first. This includes things such as inputs and outputs, mathematical formulas and processes, edge cases, stupid errors, etc.

A good way to start is to think about how we'd do this on paper. For example, how would we get the primes up to 20 without computer help?

First, even if it seems obvious, write out exact definitions. This will be convenient when the coding implementation process begins.

Definition 1. *Prime Number An integer greater than 1 that has no divisors smaller than itself other than 1.*

I'd think to look at the numbers **2** through 20 (numbers *greater* than 1) and see which ones are prime.

Another useful fact you might find is that we only need to actually check numbers up to the **square root** of our upper limit: $\forall(N, k > 0)k \leq \sqrt{N} \Leftrightarrow \frac{N}{k} \geq \sqrt{N}$

Finally, let's write out our inputs, our outputs, and how we want the program to be called. Our input is our upper limit, which we specify as a command line argument:

```
java PrimePrinter 20
```

Remember to run a Java program, we need to compile and run it by the class name. So `PrimePrinter` seems like a pretty meaningful name. Convention for naming classes is not camelCase, but just capitals.

We want our output to just print the numbers separated by a single space. The standard `System.out.println` automatically line breaks, so we can use `System.out.print` instead and add the space manually.

1.2 Step 2: Skeleton

Now that we have a good idea of the process our algorithm takes, we aim to go from general to specific. We start by drawing out a **code skeleton**. Essentially, this is just outlining the various functions we need to carry out, as an abstract, unimplemented process, for now. You'll see this given to you for most of the projects this semester (except Gitlet, which you start pretty much from scratch).

First, a function to print primes. This is the function that we call from main. Remember you can be super abstract here - it's ok to write note-taking code at this stage.

```
public static void main(String[] args) {
   printPrimes(first command line argument);
}

private static void printPrimes(int limit) {
   //For all integers between 2 and limit:
   // Check if prime
   // print it if it is, then space.
}

private static boolean isPrime(int x) {
   return //True if x is prime. How?????
}
```

Now we just have to think about how to check if a number is prime. We covered this in planning- we can just check integers up to the square root of the number in question, and see if they divide evenly!

```
private static boolean isPrime(int x) {
   if (x <= 1) return False
   else //check all numbers up to the square root of x.
      //If any of those numbers mod x return 0, immediately return false.
   //If none of them do, return true.
}
```

Remember that last note that command line arguments were stored in the args[] **String** array. Since our limit is a String, it has to be **typecasted** to an integer so it can be used for integer stuff like for loops. Java is far more strict on correct typing than Python. If you ever get an `incompatible types` error, check that you've casted correctly!

Also notice that since these are pure functions and are called without any object creation, we have to make them static.

1.3 Step 3: Implementation

Now that everything is down conceptually, this last stage is probably the easiest- just convert your abstractions and notes into functional Java code.

```java
import java.lang.Math;
public class PrimePrinter{
   public static void main(String[] args) {
      printPrimes(Integer.parseInt(args[0]));
   }

   private static void printPrimes(int limit) {
      for (int num = 2; num <= limit; i++) {
         if (isPrime(num)) {
            System.out.print(num + " ");
         }
      }
   }
   private static boolean isPrime(int x) {
      if (x <= 1) {
         return false;
      } else {
         for (int i = 2; i <= Math.sqrt(x); i++) {
            if (x % i == 0) {
               return true;
            }
         }
         return false;
      }
   }
}
```

NOTE: The code above still isn't perfect! Next note, we'll figure out how what ways we can ”improve” it.

Overall, modularly written code is far easier to work with than code written all at once. It will also allow you to catch and fix bugs far more easily, since you know what parts of the program do what! It's all about organization and role management leading to increasing quality.

2 Testing with JUnit

Writing your own test cases not only makes up some project points in this class, it's also a good practice in general. It forces you to find your own bugs and edge cases to ensure your program works as well as possible.

To write such cases, we can utilize Java's `JUnit` library. This is a special library that allows us to have a second `main` method, one used only for testing. Simply use `import org.junit.Test` in your testing class to import it.

In `JUnit`, we make a separate class for test cases (this is usually called `UnitTest` in this class's projects). Each test cases is represented by a function with the `@Test` annotation. When you do so, you should see a green arrow that looks like ▶ pop up next to your function. Click it to run it. If the test case function runs without any kinds of errors, the green arrow will be replaced with indicating a passed test case.

In order to check if outputs are correct, a very useful library is the `Assert` library, which you can import with `import static org.junit.Assert.*`. This will give us many useful "checker" functions, which allows us to "assert" that actual program outputs are the expected (correct) outputs. If they aren't, a nice `AssertionError` will be raised.

An example test case for our `printPrimes` program above would look like this:

```java
import org.junit.Test;
import static org.junit.Assert.*;
public class UnitTest {
   /** Test our isPrime function on a few numbers. **/
   @Test
   public void testIsPrime() {
      assertTrue(isPrime(11));
      assertTrue(isPrime(29));
      assertFalse(isPrime(100));
      assertFalse(isPrime(169));
   }
}
```

Let's end the note here. The next note covers the second half of the lecture, which will analyze different ways to approach and improve the `printPrimes` program.

3 Summary

In this note, we covered:

- The PSI process for developing and writing quality programs.

- How to utilize the `JUnit` library to write test cases for your program.