Chapter 18

Examples using Iteration

Some sophisticated algorithms can be implemented as Java programs now that we are armed with `if` and `while` statements. This chapter provides a number of examples that show off the power of conditional execution and iteration.

18.1 Example 1: Drawing a Tree

Suppose such a tree must be drawn, but its height is provided by the user. `StarTree` (18.1) provides the necessary functionality.

```
import java.util.Scanner;

public class StarTree {
    public static void main(String[] args) {
        Scanner kbd = new Scanner(System.in);
        int height;    //  Height of tree
        System.out.print("Enter height of tree: ");
        height = kbd.nextInt();  //  Get height from user
        int row = 0;               //  First row, from the top, to draw
        while ( row < height ) {   //  Draw one row for every unit of height
            //  Print leading spaces
            int count = 0;
            while ( count < height - row ) {
                System.out.print(" ");
                count++;
            }
            //  Print out stars, twice the current row plus one:
            //    -  number of stars on left side of tree = current row value
            //    -  exactly one star in the center of tree
            //    -  number of stars on right side of tree = current row value
            count = 0;
            while ( count < 2*row + 1 ) {
                System.out.print("*");
```



```
                count++;
            }
            //  Move cursor down to next line
            System.out.println();
            //  Change to the next row
            row++;
        }
    }
}
```

Listing 18.1: `StarTree`—Draw a tree of asterisks given a user supplied height

When `StarTree` is run and the user enters `7`, the output is:

```
Enter height of tree: 7
       *
      ***
     *****
    *******
   *********
  ***********
 *************
```

`StarTree` uses two `while` loops nested within a `while` loop. The outer `while` loop is responsible for drawing one row of the tree each time its body is executed:

- As long as the user enters a value greater than zero, the body of the outer `while` loop will be executed; if the user enters zero or less, the program terminates and does nothing.
- The last statement in the body of the outer `while`,

  ```
  row++;
  ```

  ensures that the variable `row` increases by one each time through the loop; therefore, it eventually will equal `height` (since it initially had to be less than `height` to enter the loop), and the loop will terminate. There is no possibility of an infinite loop here.
- The body consists of more than one statement; therefore, it must be enclosed within curly braces. Whenever a group of statements is enclosed within curly braces a *block* is formed. Any variable declared within a block is local to that block. Its scope is from its point of declaration to the end of the block. For example, the variables `height` and `row` are declared in the block that is `main()`'s body; thus, they are local to `main()`. The variable `count` is declared within the block that is the body of the `while` statement; therefore, `count` is local to the `while` statement. An attempt to use `count` *after* the `while` statement outside its body would be an error.

The two inner loops play distinct roles:

- The first inner loop prints spaces. The number of spaces printed is equal to the height of the tree the first time through the outer loop and decreases each iteration. This is the correct behavior since each succeeding row moving down contains fewer leading spaces but more asterisks.

- The second inner loop prints the row of asterisks that make up the tree. The first time through the outer loop, row is zero, so no left side asterisks are printed, one central asterisk is printed (the top of the tree), and no right side asterisks are printed. Each time through the loop the number of left-hand and right-hand stars to print both increase by one and the same central asterisk is printed; therefore, the tree grows one wider on each side each line moving down. Observe how the `2*row + 1` value expresses the needed number of asterisks perfectly.

18.2 Example 2: Prime Number Determination

A *prime number* is an integer greater than one whose only factors (also called divisors) are one and itself. For example, 29 is a prime number (only 1 and 29 divide into it with no remainder), but 28 is not (2 is a factor of 28). Prime numbers were once merely an intellectual curiosity of mathematicians, but now they play an important role in cryptography.

The task is to write a program that displays all the prime numbers up to a value entered by the user. `PrintPrimes` (18.2) provides one solution.

```
import java.util.Scanner;

public class PrintPrimes {
    static boolean isPrime(int n) {
        for ( int trialFactor = 2;  trialFactor <= Math.sqrt(n);
              trialFactor++ ) {
            if ( n % trialFactor == 0 ) {  //  Is trialFactor a factor?
                return false;  //  Yes, return right away
            }
        }
        return true;  //  Tried them all, must be prime
    }
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int maxValue;
        System.out.print("Display primes up to what value? ");
        maxValue = scan.nextInt();
        int value = 2;  //  Smallest prime number
        while ( value <= maxValue ) {
            //  See if value is prime
            if (isPrime(value)) {
                System.out.print(value + " ");   //  Display the prime number
            }
            value++;  //  Try the next potential prime number
        }
        System.out.println();    //  Move cursor down to next line
    }
}
```

Listing 18.2: `PrintPrimes`—Prime number generator

18.2, with an input of 90, produces:

```
Display primes up to what value? 90
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89
```

The logic of `PrintPrimes` is a little more complex than that of `StarTree` (18.1). The user provides a value for `maxValue`. The main loop (outer `while`) iterates over all the values from two to `maxValue`.

- Two new variables, local to the body of the outer loop, are introduced: `trialFactor` and `isPrime`. `isPrime` is initialized to true, meaning `value` is assumed to be prime unless our tests prove otherwise. `trialFactor` takes on all the values from two to `value - 1` in the inner loop:

```
int trialFactor = 2;
while ( trialFactor < value - 1 ) {
    if ( value % trialFactor == 0 ) {  //  Is trialFactor a factor?
        isPrime = false;
        break;  //  No need to continue, we found a factor
    }
    trialFactor++;     //  Try next factor
}
```

  If any of the values of `trialFactor` is determined to actually be a factor of `value`, then `isPrime` is set to false, and the loop is prematurely exited. If the loop continues to completion, `isPrime` will never be set to false, which means no factors were found and `value` is indeed prime.

- The `if` statement after the inner loop:

```
if ( isPrime ) {
    System.out.print(value + " ");   //  Display the prime number
}
```

  simply checks the status of `isPrime`. If `isPrime` is true, then `value` must be prime, so `value` (along with an extra space so numbers printed later will not run together) will be printed.

Some important questions can be asked.

1. **If the user enters a 2, will it be printed?** `maxValue` = `value` = 2, so the condition of the outer loop

   ```
   value <= maxValue
   ```

   is true, since $2 \leq 2$. `isPrime` is set to true, but the condition of the inner loop

   ```
   trialFactor < value - 1
   ```

   is not true ($2 \not< 2 - 1$). Thus, the inner loop is skipped, `isPrime` is not changed from true, and 2 is printed.

2. **Is the inner loop guaranteed to always terminate?** In order to enter the body of the inner loop, `trialFactor < value - 1`. `value` does not change anywhere in the loop. `trialFactor` is not modified anywhere in the `if` statement, and it is incremented immediately after the `if` statement. Therefore, eventually `trialFactor` will equal `value - 1`, and the loop will terminate.

3. **Is the outer loop guaranteed to always terminate?** In order to enter the body of the outer loop, `value` $\leq$ `maxValue`. `maxValue` does not change anywhere in the loop. `value` is increased in the last statement within the body of the outer loop, and since the inner loop is guaranteed to terminate as shown in the previous answer, eventually `value` will exceed `maxValue` and the loop will end.

The logic of the inner `while` can be rearranged slightly to avoid the `break` statement. The current version is:

```
boolean isPrime = true;  //  Assume no factors unless we find one
while ( trialFactor < value - 1 ) {
    if ( value % trialFactor == 0 ) {  //  Is trialFactor a factor?
        isPrime = false;
        break;  //  No need to continue, we found a factor
    }
    trialFactor++;     //  Try next factor
}
```

It can be rewritten as:

```
boolean isPrime = true;  //  Assume no factors unless we find one
while ( isPrime && trialFactor < value - 1 ) {
    isPrime = !(value % trialFactor == 0);
    trialFactor++;     //  Try next factor
}
```

This version without the `break` introduces a slightly more complicated condition for the `while` but removes the `if` statement within it body. Profiling reveals that neither loop structure offers a performance advantage over the other.

The performance can be enhanced, however, with a modification to the algorithm. The algorithm can be made more efficient. For example, if a number n is prime, `PrintPrimes` can only certify that n is prime after testing all the integers in the range $2 \ldots n-1$. For large n, this could take some time. A simple optimization is based on the fact that if n has no factors in the range $2 \ldots \sqrt{n}$, then n must be prime. Fortunately, a standard class named `Math` provides a square root method. To use this square root method, replace the condition of the inner loop of `PrintPrimes` with

```
//  Was: while ( trialFactor < value - 1 ) {
//  New version:
while ( trialFactor <= Math.sqrt(value) ) {
```

The `Math.sqrt()` method computes the square root of the `double` argument passed to it. The result returned is a `double` value. Automatic widening converts the `int` value to a `double`.

Does this optimization make any difference? Since computers are so fast, perhaps this minor change makes little or no difference in the speed of the program. A technique known as *profiling* can answer this question. The simplest form of profiling involves timing a section of code to see how long it takes to execute. The original code is modified to permit this timing. The act of modifying source code to derive information beyond what the original code is intended to provide is called *instrumenting* the code. Code can be instrumented for reasons other than performance testing.

`InstrumentedPrimes` (18.3) is the instrumented version of `PrintPrimes` (18.2).

```
import java.util.Scanner;
```



```
public class InstrumentedPrimes {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int maxValue;
        System.out.print("Display primes up to what value? ");
        maxValue = scan.nextInt();
        int value = 2;  //  Smallest prime number
        long startTime = System.currentTimeMillis();
        while ( value <= maxValue ) {
            //  See if value is prime
            int trialFactor = 2;
            boolean isPrime = true;  //  Assume no factors unless we find one
//            while ( trialFactor <= Math.sqrt(value) ) {
            while ( trialFactor < value - 1 ) {
                if ( value % trialFactor == 0 ) {  //  Is trialFactor a factor?
                    isPrime = false;
                    break;  //  No need to continue, we found a factor
                }
                trialFactor++;     //  Try next factor
            }
//            if ( isPrime ) {
//                System.out.print(value + " ");   //  Display the prime number
//            }
            value++;  //  Try the next potential prime number
        }
//        System.out.println();     //  Move cursor down to next line
        System.out.println("Elapsed time: " +
                           (System.currentTimeMillis() - startTime) +
                           " msec");

    }
}
```

Listing 18.3: `InstrumentedPrimes`—Instrumented prime number generator

The original version was modified as follows:

- The new inner `while` appears that uses `Math.sqrt()` is commented out, so the original condition is still checked. This configuration is used to time the original version. To test the new version, *un*comment the new version and comment out the original version.
- The method `System.currentTimeMillis()` gets information from the operating system's clock to determine the number of milliseconds since midnight January 1, 1970. Calling it twice and comparing the two results indicates the elapsed time in milliseconds between the two calls.
- The original output statements (the `print()` and `println()` calls) are commented out. We wish to compare the raw speed of the two versions, and I/O tends to slow things down considerably. If we try large values (like 10,000) the slowdown from displaying all the primes would somewhat obscure the speed difference between the two programs. We're testing raw processor speed, not the speed of the I/O devices.

- The new print statement displays the difference in times (elapsed time) after the outer loop has terminated.

Three runs of the instrumented original version on one computer system produce[1]:

```
#>java PrintPrimes
Display primes up to what value? 100000
Elapsed time: 49360 msec
#>java PrintPrimes
Display primes up to what value? 100000
Elapsed time: 49359 msec
#>java PrintPrimes
Display primes up to what value? 100000
Elapsed time: 49360 msec
```

Three runs of the instrumented square root version produce:

```
#>java PrintPrimes
Display primes up to what value? 100000
Elapsed time: 790 msec
#>java PrintPrimes
Display primes up to what value? 100000
Elapsed time: 799 msec
#>java PrintPrimes
Display primes up to what value? 100000
Elapsed time: 792 msec
```

All the primes up to 100,000 are found during each run. The difference is dramatic. The original version averages a little over 49 seconds to find all the prime numbers; the optimized version takes less than one second! The change was minor, but the effect was major. This illustrates the value of a quality algorithm. A good algorithm is both correct *and* efficient.

18.3 Example 3: Digital Timer Display

We can put the `System.currentTimeMillis()` method to good use in implementing a digital timer. `DigitalTimer` (18.4) implements a digital timer using loops, conditional statements and the `System.currentTimeMillis()` method.

```
public class DigitalTimer {
    private static long hours = 0;
    private static long minutes = 0;
    private static long seconds = 0;
    public static void main(String[] args) {
```

[1]Note: if you run this on a multi-user system with a number of people logged on and doing work, the results may differ more widely between each run of the same version of the program. The elapsed time computed from the `System.currentTimeMillis()` calls indicates the real-time elapsed time, not the time dedicated exclusively to your Java process.


```
        //  Some conversions from milliseconds
        final long SECONDS = 1000,  //  1000 msec = 1 sec
                   MINUTES = 60 * SECONDS,  //  60 seconds = 1 minute
                   HOURS = 60 * MINUTES,  //  60 minutes = 1 hour
                   _24_HOURS = 24 * HOURS;  //  24 hours = 24 hours
        //  Record starting time
        long start = System.currentTimeMillis();
        //  Elapsed time
        long elapsed = System.currentTimeMillis() - start,
             previousElapsed = elapsed;

        //  Counts up to 24 hours, then stops
        while ( elapsed < _24_HOURS ) {
            //  Update the display only every second
            if ( elapsed - previousElapsed >= 1000 ) {
                //  Remember when we last updated the display
                previousElapsed = elapsed;
                //  Compute hours
                hours = elapsed/HOURS;
                //  Remove the hours from elapsed
                elapsed %= HOURS;
                //  Compute minutes
                minutes = elapsed/MINUTES;
                //  Remove the minutes from elapsed
                elapsed %= MINUTES;
                //  Compute seconds
                seconds = elapsed/SECONDS;
                //  Display results  with leading zeroes
                System.out.printf("%02d:%02d:%02d\n",
                                  hours, minutes, seconds);
            }
            //  Update time: number of milliseconds since starting
            elapsed = System.currentTimeMillis() - start;
        }
    }
}
```

Listing 18.4: `DigitalTimer`—a digital timer program that accurately keeps time

The operating system has an accurate real-world clock (as accurate as the hardware clock that the OS uses, at least). We can continuously get the current time to measure the time difference from when the timer began. The absolute number of milliseconds since the timer was started then can be displayed in an hours:minutes:seconds format.

In `DigitalTimer`:

- The timer begins at 00:00:00 and counts up to 23:59:59. When it reaches its maximum value it stops and the program terminates.

- Three variables keep track of the hours, minutes, and seconds that the timer displays. The variables cannot be displayed directly as is; otherwise, the display 12:2:4 will result when 12:02:04 is desired. We use the `%02d` control code in `System.out.printf()` to print two-digit numbers with leading zeroes, if necessary.
- The key variable is `elapsed`, which keeps track of milliseconds since starting the timer. It is updated at each time through the loop.
- Four constants are defined to simplify the calculations in the code. Each represents the number of milliseconds in the time interval specified by its name. Observe how a previously defined constant can be used in the definition of a new constant.
- Each time through its loop `DigitalTimer` checks the total elapsed time from the start.
- The body of the loop can be executed many times each second. If the display were updated each time through the loop, most of the time the new display would be unchanged.
- The appropriate hours, minutes, and seconds are derived from the milliseconds elapsed since the start. We start with the highest time unit, hours, and work toward the lowest unit, seconds. Observe that the process is regular:
  - divide `elapsed` by the time unit to get the number of those time units expressed in the milliseconds stored in `elapsed` and
  - use the modulus operator with the time unit to get the remainder of milliseconds left over after removing the milliseconds represented by that time unit—this remainder is used to compute the lower time units.

18.4 A Mutable List

Recall `List` (16.7), an abstract class that defines the interface of list objects. In our original implementation in § 16.2 our lists were immutable, meaning an existing list could not be changed. Appending to an existing list produced a copy of the list with the new element tacked onto the end. Immutable objects—like Java's `String` objects—offer some inherent advantages over mutable objects. In general it is easier to reason about the correctness of programs when objects are immutable. Since an immutable object's state cannot change, when the object's value is determined at one point in the program's execution it is guaranteed to have that same value at any point later during the program's execution.

Immutability does have its price, however. Object creation is a relatively expensive operation compared to say, assigning a number to a numeric variable or testing a condition. Making a copy of an object when we need a modified form of it is inefficient if we never intend to use the original version again. In the case of our earlier list implementation, suppose we have a list containing 10,000 elements, and we wish to add a new element to the end:

```
//  lst is a NonemptyList containing 10,000 elements.
lst = lst.append("Betty");
```

The right-hand side of the assignment creates a new list and copies all the elements from `lst` plus `"Betty"` to the new list object. The assignment operator directs the reference `lst` to refer to this newly created list. Appending our new element thus results in the creation of 10,001 objects! A more efficient approach would create one new object (`"Betty"`) and somehow link it into the existing list.

We can begin with the same abstract class, `List` (16.7), and create a mutable list object, `MutableList` (18.6):


```
public class ListNode {
    public Object element;
    public ListNode next;
    public ListNode(Object elem) {
        element = elem;
        next = null;
    }
}
```

Listing 18.5: `ListNode`—a primitive building block for mutable lists

```
public class MutableList extends List {

    private ListNode first;  //  First node in the list (null if empty)
    private ListNode last;   //  Last node in the list (null if empty)

    public MutableList() {
        first = last = null;  //  List initially empty
    }

    //  Counts the number of elements in the list
    public int length() {
        int len = 0;
        ListNode cursor = first;  //  Cursor steps through list;
                                  //      set it to the beginning
        while (cursor != null) {  //  While we have not reached the
                                  //      of the list . . .
            len++;                //  Count the current element
            cursor = cursor.next; //  Move to the next element
        }
        return len;
    }

    //  Adds an element to the end of the list without creating a copy
    //  of the list.
    public List append(Object newElement) {
        //  Make node for new element
        ListNode newNode = new ListNode(newElement);
        if (first == null) {  //  Empty list
            first = last = newNode;
        } else {
            last.next = newNode;  //  Add to current end of list
            last = newNode;       //  Update the end of list to the
        }                         //      to the new element
        return this;              //  Return current list to comply
    }                             //      with superclass requirements

    protected String toStringHelper() {
```



```
        String result = " ";        //  Empty, until determined
                                    //      otherwise
        ListNode cursor = first;    //  Consider first element, if it
                                    //      exists
        while (cursor != null) {    //  Continue until we reach the end
                                    //      of the list
            result += cursor.element + " ";
            //  Move to next element in the list
            cursor = cursor.next;
        }
        return result;
    }
}
```

Listing 18.6: `MutableList`—a class for building mutable list objects

In addition to mutability, our new list class uses iteration exclusively instead of recursion. Iteration is more efficent than recursion, because a method call is relatively expensive compared to a non-method call statement (such as an assignment or conditional check). Whenever a method is called, some information needs to be stored in memory:

- parameters, if applicable (separate copies for each recursive invocation)
- local variables, if applicable (separate copies for each recursive invocation)
- location of the call (return address), so execution can return to the correct position within the code when the method is finished with its invocation

Storing this information takes time and, of course, extra memory.

`MutableListTest` (18.7) tests the `append()` and `toString()` methods of our mutable list:

```
import junit.framework.TestCase;

public class MutableListTest extends TestCase {

    public void testAppendMethod() {
        List list = new MutableList();
        assertEquals("[ ]", list.toString());
        assertEquals(0, list.length());
        list = list.append(22);
        //System.out.println(list);
        assertEquals("[ 22 ]", list.toString());
        assertEquals(1, list.length());
        list = list.append("Fred");
        assertEquals("[ 22 Fred ]", list.toString());
        assertEquals(2, list.length());
        list = list.append(2.2);
        assertEquals("[ 22 Fred 2.2 ]", list.toString());
        assertEquals(3, list.length());
```



```
            list = list.append("Wilma");
            assertEquals("[ 22 Fred 2.2 Wilma ]", list.toString());
            assertEquals(4, list.length());
        }
}
```

Listing 18.7: `MutableListTest`—testing our mutable list

18.5 Summary

- Iteration is a powerful mechanism and can be used to solve many interesting problems.
- A block is any section of source code enclosed within curly braces.
- A variable declared within a block is local to that block.
- Complex iteration using nested loops mixed with conditional statements can be difficult to do correctly.
- Sometimes simple optimizations can speed up considerably the execution of loops.

18.6 Exercises

1. Add item here

