Chapter 1: Introducing Java

1. What is Java?

Java is a programming language offering many features that make it attractive for mathematical illustration. First, it is a high-level language providing a wealth of constructs, yet at the same time it is relatively simple to use so that, after mastering a few concepts, you may begin writing useful code quickly. It was designed with graphics in mind and now supports powerful graphical tools similiar to those found in PostScript. Also important is that Java code runs and, at least in theory, behaves in the same way on every platform. Combined with its compatibility with networking and the Internet, this makes Java a powerful tool for publishing illustrations.

The first thing to know is that Java is an *object-oriented language*. This provides the programmer with a natural way to divide programs into smaller, reusable pieces, called objects, thus making it easier to manage, debug and expand the code. If you have no experience with object-oriented languages, this will likely require the greatest adjustment in your thinking. At the same time, it is one of the nicest features of Java: you will likely find that it provides a natural, elegant way to construct programs and ultimately it will allow you to develop code more efficiently.

Java's syntax is similar to the programming language C, and those with experience with C generally find it relatively easy to read Java code. In fact, C++ is something of a middle ground between C and Java as it adds object-oriented features to C. However, Java is different from these languages in that it adopts the object-oriented model more fully; loosely said, everything is an object in Java.

This first chapter is meant to be an introduction to programming in Java. In particular, you'll see how to create and run Java programs and build a few programs of your own. The next chapter will explain object-oriented programming. Like most programming languages, Java is best learned by actively writing programs and these notes aim to get you to that point quickly. I prefer at the beginning not to explain every detail that you will see in the examples. Instead, we will borrow some existing code, modify it by focusing on smaller pieces and only later go back to consider the big picture. This is likely the same way in which most people learn $\TeX$.

2. Running Java programs

To develop Java programs and run them, you will first need to obtain a version of the Java Developer's Kit freely distributed by Sun and available at

```
http://java.sun.com/
```

Following the link to Downloads, obtain the Java 2, Standard Edition (J2SE) applicable to your operating system. (At this time, Sun provides the Java 2 Platform for Windows, Linux and Solaris. If you are running another operating system, you will have to browse around to see if a Developer's Kit is available.) It is important that you obtain the Software Developer's Kit (SDK), as this kit provides the tools necessary to compile and run Java code.

Exercise 1: If you do not already have it, download and install the J2SDK following the installation instructions. This is usually a painless process.

Now let's look at our first Java program.

```
public class WelcomeToJava {
    public static void main(String[] args) {
        System.out.println("Welcome to Java");
    }
}
```

This may look mysterious now, but we will gradually come to understand what everything means. Our present goal is simply to run this program.

To begin, you should open a text editor, enter the code and save it as an unformatted text file. While the Java compiler, which we will meet in a moment, is relatively insensitive to spacing, it is a good idea to follow the format of one instruction per line. The indentation is a matter of style, but it is quite useful to follow this pattern to help you proofread your code. Many text editors, such as Emacs, will automatically indent your code.

Once you have entered the code, save it in a file called `WelcomeToJava.java`. For reasons we will understand later, the Java compiler is picky about the name of the file. It is just as well since this naming convention will help you keep track of large amounts of code.

Now you are ready to compile your program. The Java compiler is included with the SDK and may be invoked using the command `javac` assuming that this executable file is located in your path. To do this, call up a terminal window and choose the current directory to be the one in which you saved the file. Enter

```
$ javac WelcomeToJava.java
```

Assuming everything works, the compiler will not report back to you. Instead, you should see a new file `WelcomeToJava.class` in your directory. The compiler has taken the human-readable file `WelcomeToJava.java` and converted it into executable Java byte code, which may in turn be read and executed by any "Java Virtual Machine," one of which is included with the SDK distribution installed on your system. To do this, enter

```
$ java WelcomeToJava
```

If the program works properly, it should print out "Welcome To Java" on a new line. The most likely problem you could encounter results from the environment variable CLASSPATH being set incorrectly. This environment variable tells Java where to find your class file. Therefore, if Java reports `Can't find class WelcomeToJava`, you should make sure the CLASSPATH environment variable is set to the current directory. If you are using a terminal window running the bash shell, you can do this as:

```
export CLASSPATH=.
```

In a Windows command prompt window, you can use:

```
set CLASSPATH=.
```

Exercise 2: Compile and run the program WelcomeToJava on your system.

This is a very simple program. The first line:

```
public class WelcomeToJava {
```

declares that we are defining a Java *class* called `WelcomeToJava`. For now, you only need to know that

every piece of Java code you write will be contained within some class.

This is a feature of the object-oriented nature of Java about which we will learn more later.

Brackets { and } are used to delimit blocks of code. As this example shows, blocks of code can be nested inside one another. For instance, all of the code between { on the first line and the closing } on the last line comprises the definition of the class `WelcomeToJava`.

The second line begins the definition of a *method* called `main`.

```
    public static void main(String[] args) {
```

In other languages, methods are sometimes called subroutines or procedures. When Java byte code is executed, the virtual machine looks inside the `.class` file for a method called `main` to execute.

Finally, inside the `main` method, we see the third line

```
        System.out.println("Welcome to Java");
```

which has the effect of printing the words "Welcome to Java" onto the terminal.

At this point, there is no need to concern yourself with words such as `public` and `static`.

3. A few basic constructions

This section presents several examples that illustrate standard programming constructions. The next chapter is devoted to a general description of object-oriented programming.

Example 1: Use of variables

```
public class Squaring {
    public static void main(String[] arguments) {
        int n = Integer.parseInt(arguments[0]);
        int n2 = n * n;
        System.out.println(n + " squared is " + n2);
    }
}
```

Save this in a file called `Squaring.java` and compile it. To run it, you will want to give the program an argument on the command line like this:

```
$ java Squaring 10
```

There are a few things I would like to point out here. The first line of the `main` method defines a variable `int n`. This means that the name of the variable is `n` and it holds an `int` data type. As opposed to other languages like Perl or PostScript, the type of every variable in Java must be explicitly declared.

Java provides several primitive data types; the ones you will probably use most frequently, at least at the beginning, are

`boolean`, whose value is either `true` or `false`,
`int`, a 32 bit signed integer,
`float`, a 32 bit floating point number, and
`double`, a 64 bit floating point number.

Once a variable is declared, it is accessible from anywhere within the smallest block of code that contains it.

Second, you will notice the phrase `Integer.parseInt(arguments[0])`. This has the effect of taking the first command-line argument, called `arguments[0]` in the example, in text form and converting it to an `int`.

Java supports common mathematical operations such as

`+`	for addition,
`-`	for subtraction,
`*`	for multiplication,
`/`	for division, and
`%`	for modulo.

Finally, pieces of text may be stored as `String`s in Java; for example, we may give an instruction such as

```
String s = "Some piece of text.";
```

We will see in the next chapter that `String`s are somewhat different than the data types `int` and `double` that we introduced above as they are examples of objects. From the example above, you can see that `String`s can be concatenated using the `+` operator and that `int`s will be converted to `String`s in an appropriate context.

Exercise 3: What happens if you give an argument that cannot be parsed to an `int`? For example,

```
$ java Squaring abcdefg
```

Exercise 4: Modify the Squaring program to allow for inputs that are `double`. That is, your program should respond to

```
$ java Squaring 10.1
```

with something like

```
10.1 squared is 102.00999999999999
```

To parse the argument into a `double`, use `Double.parseDouble(arguments[0])`.

Example 2: Controlling the flow of execution

The following program accepts a non-negative integer n for its input and prints out $n!$.

```
public class Factorial {
    public static void main(String[] arguments) {
        int n = Integer.parseInt(arguments[0]);
        if (n < 0) {
            System.out.println("Please enter a non-negative integer");
            return;
        }
        int factorial = 1;
        for (int i = n; i >= 1; i = i - 1) {
            factorial = factorial * i;
        }
        System.out.println(n + "! = " + factorial);
    }
}
```

Here we see two new features. First, we test the input n to make sure it is non-negative. The expression `n < 0` evaluates to give a `boolean`. If the value of this boolean is `true`, then the following block of code is executed. In this case, we print an error message and use the instruction `return` to terminate the `main` method.

After this, we see a construction typical of high-level programming languages: the `for` loop. The syntax requires three expressions: the first, `int i = n`, describes any initialization performed the first time the block of code is executed, the second, `i >= 1`, gives a boolean expression that must be `true` for the block to be evaluated, and the third, `i = i - 1`, is evaluated at the *end* of the block of code.

It is common for a variable to be modified by, say, adding or multiplying it by another quantity. To make this easier, Java allows us to write

`i++` for `i = i + 1`

and

`x /= 10` for `x = x / 10`

Exercise 5: Write a program to approximate e^2 by n terms of the Taylor series:

$$e^2 \approx \sum_{k=0}^{n} \frac{2^k}{k!}$$

Be careful how you generate the terms in the sum. For instance, it is not a good idea to compute 2^k and $k!$ since these numbers both grow quickly. Instead, consider how one term is obtained from the preceeding term in the series.

Java also supports a `while` loop construction as demonstrated by this implementation of the Euclidean Algorithm.

```
public class EuclideanAlgorithm {
    public static void main(String[] args) {
        int a = Integer.parseInt(args[0]);
        int b = Integer.parseInt(args[1]);
        if (a < 0 || b < 0) {
            System.out.println("Enter non-negative integers");
            return;
        }
        int m = a;  int n = b;
        while (n > 0) {
            int r = m % n;
            m = n;
            n = r;
        }
        System.out.println("gcd( " + a + ", " + b + ") = " + m);
    }
}
```

Here are the logical operators supported by Java:

`||`	for logical "or" (as illustrated above),
`&&`	for logical "and", and
`!`	for negation.

Example 3: Arrays

Our next program uses arrays to print out n rows of Pascal's triangle with the binomial coefficients computed using the recursion:

$$\binom{r}{c} = \binom{r-1}{c-1} + \binom{r-1}{c}.$$

There is some formatting performed to make the rows line up.

```
public class PascalTriangle {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        if (n < 0) {
            System.out.println("Enter a non-negative integer");
            return;
        }
        int[] row = new int[1];
        row[0] = 1;                           // initialize first row
        for (int r = 0; r <= n; r++) {
            for (int c = 0; c <= r; c++) {   // print out row r
                double x = row[c];
                int digits = 0;               // add spaces
                while (x >= 1) {              // so that columns
                    x /= 10;                  // line up
                    digits++;
                }
                for (int space = digits;  space < 6; space++)
                    System.out.print(" ");
                System.out.print(row[c]);
            }
            System.out.println();
            int[] nextRow = new int[r+2];    // construct next row
            for (int c = 0; c <= r+1; c++) {
                int aboveLeft = 0;
                if (c > 0) aboveLeft = row[c-1];
                int above = 0;
                if (c < r+1) above = row[c];
                nextRow[c] = aboveLeft + above;
            }
            row = nextRow;
        }
    }
}
```

One new feature here is the presence of arrays, in this case, arrays of `int`s. Variables that hold arrays must be declared as such and the way to do this is with a modifier such as `int[]`. Also, the size of the array needs to be specified; this happens with a phrase, such as `new int[3]`, which creates an array of three `int`s, individual elements of which may be addressed by an index between 0 and 2. Generally speaking, array indices begin with 0 and end at one less than the size of the array. Elements of an array are accessed by following the array name with the index in square brackets [].

This explains something we have seen earlier: the parameter passed to the main method is declared to be `String[]`, an array of `String`s that are entered on the command line. We have been retrieving the first one using `args[0]`.

Also present in this example are comments. When the Java compiler sees `//`, it ignores the remainder of the line in the input file. Comments spanning several lines may be enclosed between `/*` and `*/`.

Finally, notice that the command `System.out.print` prints the text without adding a carriage return.

Exercise 6: Create a program that prints out all the prime numbers smaller than n using the sieve of Erastothenes. Begin by creating an array of n `boolean` values and initializing them to be `true`. Then set the values indexed by multiplies of primes to `false`.

Example 4: Methods

All the code we have written so far has been inside the definition of a method called `main`. It is, of course, possible to define other methods to avoid repetition of pieces of our code or simply to improve its readability. The follow program will print out a given number of rows of Pascal's Triangle. However, the binomial coefficients are now computed as

$$\binom{r}{c} = \frac{r!}{c!(r-c)!}$$

and a new method is introduced to compute the factorials.

```
public class PascalTriangleWithMethods {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        if (n < 0) {
            System.out.println("Enter a non-negative integer");
            return;
        }
        for (int r = 0; r <= n; r++) {
            for (int c = 0; c <= r; c++) {
                int bc = factorial(r)/factorial(c)/factorial(r-c);
                for (int space = digits(bc);  space < 6; space++)
                    System.out.print(" ");
                System.out.print(bc);
            }
            System.out.println();
        }
    }
    public static int factorial(int k) {
        int factorial = 1;
        for (int i = k; i >= 1; i = i - 1) {
            factorial = factorial * i;
        }
        return factorial;
    }
    public static int digits(int k) {   // determine the number
        double x = k;                   // of digits in k
        int digits = 0;
        while (x >= 1) {
            x /= 10;
            digits++;
        }
        return digits;
    }
}
```

There are two things to pay attention to here: the definitions of the methods and their usage within the method `main`. The definition of a method begins with a line like:

```
public static int factorial(int k)
```

The information contained in this declaration is called the *signature* of the method. The meaning of `public static` will become clear in the next chapter so let's not worry about it now. The word `factorial` gives the name of the method by which we refer to it and the modifier `int` describes the type of data that is returned from the method. Finally, `int k` describes the types of any parameters passed to the method and gives them a name by which they may be referred within the definition of the method.

It is possible to have a method that accepts no parameters just as it is possible that a method does not return anything. In the second case, the return type `void` is given. In fact, we have seen such a method: `main` does not return any data and so has return type `void`.

Once the methods are defined, we are free to refer to them as, say, `factorial(r)`.

Exercise 7: Consider the data types used in this program and the maximum values that can be represented by them. Why does this program give incorrect results for $\binom{r}{c}$ when $r \geq 13$?

Exercise 8: Remember that the Taylor series

$$\arctan x = \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k+1}}{2k+1}$$

is valid for $|x| < 1$. Moreover, for a given value of x, this gives an alternating series. Using a `while` loop to add terms until the error is sufficiently small, create a method

```
public static double arctan(double x, double tolerance)
```

that uses the Taylor series to compute $\arctan x$ with an error less than `tolerance`. Again, be careful when creating the individual terms of the series.

Also, remembering that

$$\frac{\pi}{4} = \arctan\frac{1}{2} + \arctan\frac{1}{3},$$

create a program using your method to compute π with an error less than 10^{-n} where n is input into the program. If $n = 20$, does your program give the correct result? Explain why or why not.

An alternate way to define the `factorial` method is recursively; that is, the method is computed by calling itself again:

```
public int factorial(int n) {
    if (n == 1) return 1;
    return n * factorial(n - 1);
}
```