INTRODUCTION TO COMPUTER PROGRAMMING

Richard Pierse

Class 6:
Programming in *Java*

History of the *Java* Language

The *Java* language was created in 1995 by Sun Microsystems. It was conceived as a language to develop platform independent applications that can run on all types of consumer electronic devices, from computers to remote controls. The syntax was based on the earlier C and C++ languages but *Java* is simpler and more compact and avoids some of the problems in those languages. The original name for the language was *Oak* but this was already in use as the name of another language so a new name had to be invented. The official story is that the inspiration for the name came on a trip to a local coffee shop, but others have speculated that it is an acronym of the names of three members of the development team: **J**ames Gosling, **A**rthur **V**an Hoff and **A**ndy Bechtolsheim.

The development of *Java* coincided with the explosion of the World Wide Web and it soon became clear that *Java* was the ideal tool for developing small programs, known as ***applets***, that can be sent across the Internet to be run locally by a web browser on the user's computer.

Java has rapidly become one of the most widely used computer languages, for writing both *applets* and stand-alone programs.

The *Java* Compiler

The *Java* compiler operates rather differently from other compilers. Instead of directly producing machine code for a specific machine, it produces *virtual* machine code called ***bytecode*** that is machine-independent. The bytecode can then be executed on any machine by a ***Java Virtual Machine*** *(JVM)*, which translates it into actual machine code and runs it. Web browsers such as *Internet Explorer* and *Netscape* are *JVMs*.

The *Java* Language

The *Java* language has many similarities to *BASIC* but also some differences. In the following, we highlight the differences but should bear in mind the many common features. Java is a case-sensitive language. Required keywords are indicated in **bold** typeface.

Command Terminators

Commands in *Java* are normally terminated by a semicolon. This allows several commands to appear on one line as in:

```
r = 3.0; pi = 3.141592654; area = pi*r*r;
```

The only exception is with certain commands like the **`if`** command that are followed by a block of statements delimited by braces `{}`.

```
if(i < 10) {j = k*i; m = n*i; }
```

Here the braces work like the `then...end if` construct in *BASIC* and no semicolon is needed after the final `}`.

Comments

There are two different kinds of comments allowed in *Java* source code. The double slash `//` indicates that the rest of the line is to be treated as a comment, just like the *BASIC* quote `'` character. In addition, longer comments extending over several lines can be delimited by the pair of characters `/*` and `*/`.

Multiple Assignment Statements

Java has many constructs designed to allow you to program in a more concise way. A simple example is multiple assignments of the form:

```
x = a; y = a; z = a; w = a;
```

Java lets you write this as the single statement

```
x = y = z = w = a;
```

Declaring Variables

The *Java* language uses *static typing*. What this means is that *Java* requires that all variables in your program are *declared* before they are used. There are various types of variable in *Java*:

Type	*Description*
int	integer values: range $\pm 2.1 \times 10^9$ e.g. `73`, `-44`
long	long integer values: range $\pm 9.2 \times 10^{18}$ e.g. `73`, `-44`
float	floating-point values (up to 6 decimal places)
double	floating-point values (up to 14 decimal places)
boolean	logical values taking the value **true** or **false**
char	single character value e.g.: `'A'`, `'z'`
String	multiple character string e.g. `"I am a string"`

Variables are declared as in the following example:

```
int i, j, k;      // declare three integers
float x, y, z;    // declare three reals
boolean ok, cancel;
char a, b, c;     /* three characters */
String str1, str2;
```

These variables are declared but not initialised. Alternatively, these variables could have been initialised as in:

```
int i=1, j=7, k=-54;
float x=0.459, y=3.14159, z=-3.3333;
boolean ok=true, cancel=false;
char a='A', b='B', c='C';
String str1="I think", str2="I am";
```

Arrays

Arrays of variables of any type can also be defined. They are declared using a **new** command. For example:

```
int list[] = new int[100];
```

declares an array of `100` integer elements. The array elements are numbered from `0` so the statement `list[0]=1;` sets the first element to `1` and `list[99]=-3` sets the last element to `-3`.

Operators

Java has the same standard mathematical operators as *BASIC* but also has some extra ones of its own. Possibly the most useful are the incremental operators `++` and `--` , which increment or decrement a value. For example, the statements

```
++j;      or    j++;
```

are both equivalent to the statement `j=j+1;`

The difference between the two forms is to do with the precedence of the two operators. In the first *prefix* form, the `++` operator has higher precedence than any other operator and so will be evaluated before any thing else. In the second *postfix* form the operator has lower precedence and so will be evaluated after everything else. Thus, for example, `k = ++j*2;` is equivalent to

```
j = j + 1; k = j * 2;
```

whereas, `k = 2*j++;` is equivalent to

```
k = 2 * j; j = j + 1;
```

If `j` is initially equal to `3`, then in the first case `k` equals `8` and in the second case `k` equals `6`.

Assignment Operators

Java also has special assignment operators to simplify common updating expressions such as

```
n = n + 3;    or    f = f * 0.5;
```

These expressions can be written in *Java* as

```
n += 3;       and    f *= 0.5;
```

using the special assignment operators `+=` and `*=`. Other *Java* assignment operators with obvious interpretation are `-=` and `/=`.

Relational Operators

Java uses the following six relational operators. Notice that two of these (== and `!=`) are different from their *BASIC* equivalents.

Operator	*Description*	*Example*
<	*Less than*	`sales < maxsales`
>	*Greater than*	`amount > 100.0`
==	*Equal to*	`age == 21`
>=	*Greater than or equal to*	`grade >= 70`
<=	*Less than or equal to*	`price <= 1.0`
!=	*Not equal to*	`year != 2004`

Conditionals

Java **`if`** statements take the form:

```
if(condition)
     { block of one or more Java statements }
else
     { block of one or more Java statements }
```

Here is an example of a nested **`if`** statement:

```
if(age < 18) {
     if(age > 12) {
          teenager = true;
          child = adult = false;
     }         // end of inner if clause
     else {
          child = true;
          teenager = adult = false;
     }         // end of inner else clause
}    // end of outer if clause
else {
     adult = true;
     child = teenager = false;
}    // end of outer else clause
```

`if` statements can be nested to any depth in *Java*. Notice that no semicolon is needed after the `}` character.

Loops

Like *BASIC*, *Java* supports both **`for`** and **`while`** loops.

The **`for`** loop takes the general form:

```
for(exp1; exp2; exp3)
    { block of one or more Java statements }
```

where

`exp1` is a *start expression* evaluated once at the loop start
`exp2` is a boolean *test expression* evaluated each iteration
`exp3` is an *increment expression* evaluated each iteration

This is best explained through a simple example:

```
for(i=1; i <= 10; ++i) { k *= 10;}
```

Here the *start expression* is `i=1`, which initialises the variable `i`. The *test expression* is `i <= 10`. The loop will continue to execute as long as this expression is true. The *increment expression* is `++i`, which increases the value of `i` each iteration by `1`.
Thus the loop will execute `10` times with `i` increasing from `1` to `10` in steps of `1`. In a second example

```
for(x=1.0; x >= 0.0; x-=0.1) { k *= 10;}
```

the loop is executed `11` times with the (real) variable `x` taking the values `1.0, 0.9, 0.8,...,0.0`.

Note that if the test condition is false at the start of the loop, then the loop will not execute at all.

The **`while`** loop takes the general form:

```
while(condition)
    { block of one or more Java statements }
```

The following example is equivalent to the previous **`for`** loop:

```
x=1.0; while(x >= 0.0) { k *= 10; x -= 0.1;}
```

Input and Output

Java is an *Object Oriented Programming* (*OOP*) language. We will postpone a detailed discussion of *OOP* until next week's class. However, for now we need to be aware that input and output in *Java* makes use of two special objects, called `System.in` and `System.out`.

To print a line to the standard output window, we use the `println` *method* of the object `System.out` (a *method* is like a function). For example

```
System.out.println("my string");
```

prints the string `"my string"`, while

```
System.out.println(i);
```

prints the variable `i`.

At the end of every *Java* program, there should be a call to `System.exit()`. This method takes an argument, generally `0`.

A First *Java* Program

Remember the first program you wrote back in class 1? Here is the equivalent *Java* code for that program:

```
int i;
for(i=10;i>=1;--i) {
    System.out.println(i);
}
System.out.println("Blast Off!");
System.exit(0);
```

Because everything in *Java* is an object, this code needs to be embedded in a function called `main` within a *class* declaration of the form:

```
public class MyProg {
    public static void main (String args[])
        { program code goes here }
}
```

The Complete Program

The complete program should look like this:

```
public class MyProg {
     public MyProg() {}
     public static void main (String args[]) {
          int i;
          for(i=10;i>=1;--i) {
               System.out.println(i);
          }
          System.out.println(”Blast Off!”);
          System.exit(0);
     }
}
```

The program can now be compiled and run using the freeware *Java* compiler, *JCreator LE*. The steps are:

- Open the *JCreator* compiler.
- To open a new file select *File* then *New* from the menu.
- Choose *Java file* as the file type and name the file `MyProg`.
- A blank window will open in which the program can be entered.
- Select *Compile File* from the *Build* menu to compile the file.
- Select *Execute File* from the *Build* menu to run the file.

The following alternative version of the program uses the *Java* Swing library to display output via Windows message boxes:

```
import javax.swing.*;
public class WinProg {
  public WinProg() {}
  public static void main (String args[]) {
     int i;
     for(i=10;i>=1;--i) {
        JOptionPane.showMessageDialog(null,"" + i);
     }
     JOptionPane.showMessageDialog(null,"Blast Off!");
     System.exit(0);
  }
}
```

The differences from the original version have been underlined. If you edit the original version to make this *Windows* version, make sure to name it `WinProg`.