Ian Kenny

September 29, 2016

Software Workshop

Lecture 1

Welcome to Software Workshop

The aims of the module are

- To present the fundamental concepts of imperative and object-oriented programming.
- For you to develop the skills needed to design, develop and document programs.
- For you to gain working knowledge of the Java programming language.

The overall aim is for all students to have an understanding of some key programming ideas after term 1. Some students will enter the module having done lots of programming and may find that they know some or much of the material. Some students may have done no programming before at all. The aim of this module, therefore, is to get everyone to *at least* the same specific point.

Staff

Semester 1: Ian Kenny, room 232, i.s.kenny@cs.bham.ac.uk

Semester 2: Martin Escardo, room 212, m.escardo@cs.bham.ac.uk

Lead TA: Abdessalam Elhabbash, a.elhabbash@cs.bham.ac.uk

Module structure

Three lectures per week

- Monday, 1300, Vaughan Jeffreys, Education Building
- Thursday, 1500, Lecture Theatre, Arts Main Building
- Friday, 1300, Aston Webb Main Lecture Theatre

Labs are spread across the week. Refer to your timetable.

THERE ARE NO LABS ON THIS MODULE IN WEEK 1!

Assessment

The module has 100% continuous assessment.

The module is divided into two halves: term 1 and term 2. The part of the module we are discussing now is term 1.

Term 1 is worth 20% of the marks, term 2 is worth 80% of the marks.

Assessment is through weekly programming exercises that you must submit. Some of the weekly exercises will be *non-assessed*. These exercises will be more advanced than the regular exercises and will hopefully give you some things to think about.

There will also be weekly quizzes that are compulsory but do not carry any marks.

Content

The indicative content of the module is

- Fundamentals of imperative programming, Java introduction.
- Data types, variables, operators, selection, iteration, arrays.
- Object-oriented programming in Java: classes, methods, constructors, using objects, access modifiers.
- Inheritance and composition, inner classes.
- Polymorphism and method overriding.
- Abstract classes.
- Interface classes.
- Java collections, generics.
- Exceptions.
- File handling.
- GUI building.
- Design patterns.

This is not an exhaustive list. A lot of other topics will be covered within the above categories.

Module 'philosophy'

Programming is learned by doing. There are three lectures per week and the purpose of these is to introduce you to the things you should know. You cannot learn programming by simply attending lectures.

In any given week, not all of the material on the lecture slides may be covered. The lecture slides will be available to you, of course, so that you can review your understanding of what was discussed, and you should definitely do this. You can also read about the topics that may not have been covered. But, rest assured, all important stuff will definitely be covered.

The lab exercises exist so that you practice programming. You should do the exercises, obviously, but should also play around with them and try things yourself. Write programs that you have had the idea for and see if you can get them to work.

In this lecture

- Introduction to Java.
- A 'Hello, World' program.
- Primitive data types.
- Variable declaration and initialisation.
- Arithmetic operators.
- Unary operators.
- Relational and equality operators.
- Operator precedence.
- Conditional operators.
- Strings.
- Basic input/output.
- Control flow: selection.
- Control flow: iteration.
- Blocks.

Introduction to Java

One of the motivations for the creation of the Java was to enable the same program to run on multiple platforms. When Java was created, this was an unusual feature. In the past, programs had to be written for specific platforms. If it was desired that the same program run on multiple platforms, it had to be re-factored or completely rewritten for each platform. This is obviously time consuming and wasteful.

The goal of platform-independence was achieved by the creation of what is called the *Java Virtual Machine* (JVM). The JVM runs the Java program. The program does not run directly on the system's OS and hardware. The JVM provides a layer of insulation from the specifics of the machine meaning that the Java program can ignore the specifics of the machine. This allows the same program to be run on multiple platforms because a Java program is insulated from the platform.

Introduction to Java

The 'code' that the programmer writes is called *source code*. Neither the computer system nor the JVM understand Java source code. For the Java source code to be understandable by the JVM it must be *compiled* using the Java compiler `javac`.

The Java compiler produces Java *bytecode*. The bytecode is loaded and executed by the JVM. The program that runs Java programs (the JVM) is called simply `java` when using it from the command line, as we will be doing.

Introduction to Java

Java is an *object-oriented programming language*. This means that Java programs involve *objects* interacting with each other. In a real world video game program, for example, a type of object might be `Ghoul` or `MagicCoin`. Objects *encapsulate* the data relevant to the type of object and *operations* on that data. This will make more sense as we move along.

Creating a Java program

To create a Java program, the software developer writes Java *source code* in a text file.

They must write a structure called a `class`. The class contains the program (although, as will be seen, it is not essential that a class embodies a full program).

For the class to represent a program, it must contain a *method* called `main`.

Before it can *run* the program must be *compiled* using `javac`, the Java compiler. This converts the source code into *bytecode* that can be *interpreted* by the Java *runtime system*.

If the class is successfully compiled (i.e. it has no errors), it can be run by the Java runtime system. This is done by running the `java` program.

There are many terms on this slide that may not make sense yet. They will, in time.

A 'Hello, World' program in Java

```
public class HelloWorld {
    public static void main (String[] args) {
        System.out.println("Hello, World.");
    }
}
```

Listing 1 : HelloWorld.java

From a terminal, the following commands compile and run the program.

```
$ javac HelloWorld.java
$ java HelloWorld
```

Listing 2 : Terminal commands

The output will be:

```
Hello, World.
```

Data types

At its most basic, a program consists of data and operations on that data.

Data comes in various *types*. In Java there are two broad types of data types: *primitive types* and *reference types*. In the first part of this course we will mostly consider only primitive types.

Java is a *statically typed* language which means you must *declare* the type of an item of data when you create it. You must also give the data item a name.

Primitive data types in Java have a *range*: the range of values the type can hold.

Most of the primitive data types hold numerical data although there is a type for holding Boolean values (true and false) and a type for holding Unicode characters.

Primitive data types in Java

This table shows the primitive data types available in Java, what each type holds, the size of each type in bits and the range of each type.

Type	Contains	Size (bits)	Range
boolean	true or false	1	N/A
char	Unicode character	16	\u0000 to \uffff
byte	signed integer	8	-128 to 127
short	signed integer	16	-32768 to 32767
int	signed integer	32	-2,147,483,648 to 2,147,483,647
long	signed integer	64	-9,223,372,036,854,775,808 to 9 ,223,372,036,854,775,807
float	IEEE754 floating point	32	1.40129846432481707e-45 to 3.40282346638528860e+38
double	IEE754 floating point	64	4.94065645841246544e-324 to 1.79769313486231570e+308

Table 1 : Java data types

Variables

To use these data types in programs the software developer creates *variables* and *constants*. We will consider variables for now.

The process of creating a variable is called *declaring* the variable. A variable declaration specifies the *type* of the variable, its *name* and, optionally (but see the comment below), its *initial value*.

Although the setting of the initial value is optional, it is considered *good practice* to always do so. The Java compiler is strict about the use of variables that have not been *initialised*.

The format of a declaration is as follows.
{data type} {name} [= <initial value>]{;}

{ } = compulsory;[ ] = optional;<> = a value.

Variable declarations

The program below demonstrates the declaration of some variables. The code on each line, up to and including the semicolon, is a *statement* in Java.

Note that the Java compiler will not allow use of these variables yet because they have not been initialised (see previous slide).

```
public class VariablesDemo1 {
    public static void main (String[] args) {

        boolean b;
        char c;
        int i;
        float fp;
        double x;
    }
}
```

Listing 3 : VariablesDemo1.java

Variable initialisation

In Java, to use variables we must initialise them. Variables are given a value using the *assignment operator* '='.

```
public class VariablesDemo2 {
    public static void main (String[] args) {

        boolean b = true;
        char c = 'a';
        int i = -40003;
        float fp = 0.655f;
        double x = 0.56566522;
    }
}
```

Listing 4 : VariablesDemo2.java

Variable initialisation

Each of the variables in the previous listing has been *assigned* an initial value from the range shown in Table 1.

(Note, however, that the value assigned to `fp` has the character '`f`' appended to it. This is because a *floating point literal*, i.e. a floating point number entered directly in the source code, is of type `double`. Assigning a `double` to a `float` could cause a loss of precision and the compiler flags this as an error. The '`f`' appended to the value tells the compiler that we want a float literal not a double literal.)

Using variables

Once a variable has been declared (and initialised) it can be used for whichever purpose it was created for. Once a variable has been declared, its type is no longer specified when it is used. Variables can also be assigned new values.

Listing 5 on the next slide demonstrates these concepts.

Demo programs

Some of the demo programs do not have all of the results of the operations identified in the comments. Where this is the case, I suggest you attempt to work out what the result is before looking at the next slide to see the answer. Some of them may be answered in the lectures.

Using variables: demo

```
public class VariablesDemo3 {
    public static void main (String[] args) {

        // declare two integers and initialise them
        int i1 = 0;
        int i2 = 5;

        // change the value of i1 (no type needed now)
        i1 = 7;

        // set i2 to be the same value as i1
        i2 = i1;

        // change the value of i1
        i1 = 10;

        // print the values of i1 and i2 to the screen
        System.out.println("i1 = " + i1 + "; i2 = " + i2 +
            ".");
    }
}
```

Listing 5 : VariablesDemo3.java

Using variables: demo

The following terminal commands show the compilation, execution and output of the class `VariablesDemo3`.

```
$ javac VariablesDemo3.java
$ java VariablesDemo3
i1 = 10; i2 = 7.
```

Listing 6 : Terminal commands

Arithmetic operators

The programs developed so far have no practical use. In order to do useful things, programs will generally need to manipulate variables using *operators* (amongst other things). We have seen one operator so far: the assignment operator '='.

In order to do mathematical operations on variables, we need to use the *arithmetic operators*.These are shown below. Note that they are all binary operators requiring two operands.

Operator	Function
$+$	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo (remainder)

Table 2 : The arithmetic operators

Order of precedence

The order of precedence of the arithmetic operators follows their mathematical counterparts. The table below shows the order of precedence. Rows higher up in the table are of higher precedence. Within the rows the precedence is the same and expressions involving those operators will be evaluated in left-to-right order.

Type	Precedence
Multiplicative	* / %
Additive	+ -

Table 3 : Order of precedence

Arithmetic operators

In the following, we first consider the application of the arithmetical operators to integers.

Addition

```
public class ArithmeticDemo1 {
    public static void main (String[] args) {

        // i1 is set to 7
        int i1 = 2 + 5;

        // i1 is now set to 10
        i1 = i1 + 3;

        // shorthand for the above (i1 now = 13)
        i1 += 3;

        // i2 is set to 33
        int i2 = i1 + i1 + 7;

        // i2 is ???
        i2 = i2 + (i2 + i1) + 6;

        System.out.println("i2 = " + i2 + ".");
    }
}
```

Listing 7 : ArithmeticDemo1.java

Addition

```
$ javac ArithhmeticDemo1.java
$ java ArithhmeticDemo1
i2 = 85.
```

Listing 8 : Terminal commands

Subtraction

```
public class ArithmeticDemo2 {
    public static void main (String[] args) {

        // i1 is set to 6
        int i1 = 10 - 4;

        // i1 is now set to 3
        i1 = i1 - 3;

        // shorthand for the above (i1 now = 0)
        i1 -= 3;

        // i1 is set to 26
        i1 = 30 + 3 - 7;

        // i2 is set to 12
        int i2 = i1 - 10 - 4;

        // i2 is ???
        i2 = i2 - 11 + i1 - 6;

        System.out.println("i2 = " + i2 + ".");
    }
}
```

Listing 9 : ArithmeticDemo2.java

Subtraction

```
$ javac ArithhmeticDemo2.java
$ java ArithhmeticDemo2
i2 = 21.
```

Listing 10 : Terminal commands

Multiplication

```
public class ArithmeticDemo3 {
    public static void main (String[] args) {

        // i1 is set to 12
        int i1 = 2 * 6;

        // i1 is set to 24
        i1 = i1 * 2;

        // i1 is set to 48
        i1 *= 2;

        // i2 is set to ???
        int i2 = i1 * 3 * -2;

        System.out.println("i2 = " + i2 + ".");
    }
}
```

Listing 11 : ArithmeticDemo3.java

Multiplication

```
$ javac ArithhmeticDemo3.java
$ java ArithhmeticDemo3
i2 = -288.
```

Listing 12 : Terminal commands

Division

When using division with integers the question arises: what happens when the (mathematical) result is a decimal fraction? For example, `10/3`.

The answer is that the result is simply *truncated*. The fractional part of the result is removed. No rounding occurs.

Division

```java
public class ArithmeticDemo4 {
    public static void main (String[] args) {

        // i1 is set to 25
        int i1 = 50/2;

        // i1 is set to 12 (note: integer division)
        i1 = i1 / 2;

        // i1 is set to 6
        i1 /= 2;

        // i2 is set to ???
        int i2 = 60/i1/2;

        System.out.println("i2 = " + i2 + ".");

        // i3 is set to ???
        int i3 = 12/0;

        System.out.println("i3 = " + i3 + ".");

    }
}
```

Listing 13 : ArithmeticDemo4.java

Division

```
$ javac ArithhmeticDemo4.java
$ java ArithhmeticDemo4
i2 = 5.
Exception in thread "main" java.lang.ArithmeticException:
    / by zero
at ArithmeticDemo4.main(ArithmeticDemo4.java:18)
```

Listing 14 : Terminal commands

You cannot divide by zero. The result is 'undefined'.

Modulo

```
public class ArithmeticDemo5 {
    public static void main (String[] args) {

        // i1 is set to 6
        int i1 = 20 / 3;

        // i2 is set to 2
        int i2 = 20 % 3;

        // i1 is set to 12
        i1 = i1 * i2;

        // i1 is set to 0
        i1 = i1 % 2;

        // i2 is set to ???
        i2 = 30 + 10 % 3;

        System.out.println("i2 = " + i2 + ".");
    }
}
```

Listing 15 : ArithmeticDemo5.java

Modulo

```
$ javac ArithhmeticDemo5.java
$ java ArithhmeticDemo5
i2 = 31.
```

Listing 16 : Terminal commands

Precedence

```java
public class ArithmeticDemo6 {
    public static void main (String[] args) {

        // i1 is set to ???
        int i1 = 10 + 5 * 2 / 4 - 1;

        System.out.println("i1 = " + i1 + ".");

        // i2 is set to ???
        int i2 = 100 - 5 * 3 - 1 + 11 / 3;

        System.out.println("i2 = " + i2 + ".");
    }
}
```

Listing 17 : ArithmeticDemo6.java

Precedence

```
$ javac ArithhmeticDemo6.java
$ java ArithhmeticDemo6
i1 = 11.
i2 = 87.
```

Listing 18 : Terminal commands

Arithmetic with floating point types

We must be careful when using the arithmetical operators with the floating point types: `float` and `double`. This is because a computer system can store only a finite number of values of the floating point types, not all possible values. The `float` type is stored in 32 bits and the `double` type is stored in 64 bits. The finite number of bits used to represent these types means that some values cannot be represented. The format used for these types is referred to as the IEEE754 format.

With the arithmetic operators we may simply see an 'incorrect' result. When we discuss conditional statements that use the floating point types we will see other problems (that nevertheless stem from the same issue).

Arithmetic with floating point types

What will the output of this program be?

```java
public class ArithmeticDemo7 {
    public static void main (String[] args) {

        float f1 = 10 / 3;

        System.out.println("f1 = " + f1 + ".");
    }
}
```

Listing 19 : ArithmeticDemo7.java

Arithmetic with floating point types

The output of the program will be

```
f1 = 3.0.
```

This is because the the two operands of the division operator are given as integer literals (10 and 3). In order for them to be treated as floating point numbers, one of them must be explicitly denoted a floating point number. As shown on the next slide.

Arithmetic with floating point types

```
public class ArithmeticDemo8 {
    public static void main (String[] args) {

        float f1 = 10.0f / 3;

        System.out.println("f1 = " + f1 + ".");
    }
}
```

Listing 20 : ArithmeticDemo8.java

For which the output is

```
f1 = 3.3333333.
```

Note that this issue has nothing to do with the IEEE754 representation of floating point values which simply restricts the available numbers that can be represented. This issue is due to the use of integer literals in the expression.

(Don't forget that a floating point literal is of type `double` in Java, hence the 'f' appended to the value).

Arithmetic with floating point types

What will the output of this program be?

```
public class ArithmeticDemo9 {
    public static void main (String[] args) {

        float f1 = 100.0f * 3.33333f;

        System.out.println("f1 = " + f1 + ".");
    }
}
```

Listing 21 : ArithmeticDemo9.java

Arithmetic with floating point types

Of course, the expected output would be
`f1 = 333.333.`

However, here is the answer from one run of this program
`f1 = 333.33298.`

This result arises from the fact that the IEEE754 representation of floating point values means that not all values can be represented.

This is a difference of 333.333 - 333.33298 $=$ 0.00002. This may or may not be a significant difference, depending on the application.

Arithmetic with floating point types

In summary, floating point arithmetic works largely as expected but there are some possible pitfalls to look out for.

Arithmetic operators

Arithmetic operators make no sense with the boolean type (true + true = ???) but they can be used with char since that is an integral type. This can be a useful operation but may be somewhat 'opaque'.What would be the output of this program?

```
public class ArithmeticDemo10 {
    public static void main (String[] args) {

        char c = 'a';

        c += 25;

        System.out.println("c = " + c + ".");

        c /= 3;

        System.out.println("c = " + c + ".");

        c *= 2;

        System.out.println("c = " + c + ".");

    }
}
```

Listing 22 : ArithmeticDemo10.java

Arithmetic operators

```
$ javac ArithhmeticDemo10.java
$ java ArithhmeticDemo10
c = z.
c = (.
c = P.
```

Listing 23 : Terminal commands

Unary operators

There are a number of unary operators available in Java. They are shown in the table below.

Operator	Function
+	Indicates a positive value (the default)
-	Negates a numerical value
++	Increments a value by 1
- -	Decrements a value by 1
!	Negates a boolean value

Table 4 : The unary operators

There are *prefix* and *postfix* versions of the ++ and − − operators (to be explained shortly).

Operator precedence review

As stated earlier, the arithmetic operators follow the usual order of precedence followed by their mathematical counterparts.

The unary operators just introduced, however, have a higher precedence than the arithmetic operators, with the postfix operators introduced having the highest precedence.

This means that in expressions that combine these operators brackets may need to be inserted in order to force the order of evaluation that is desired.

Operator precedence review

The table below shows the order of precedence of the operators we have covered so far.

Type	Precedence
Postfix unary	*expr*++ *expr*- -
Unary	++*expr* - -*expr* +*expr* -*expr* !*expr*
Multiplicative	* / %
Additive	+ -

Table 5 : Order of precedence

Unary operators

The next program demonstrates the use of the unary + and unary − operators.

The unary + simply marks an integral value as positive. It has no other effects (apart from type promotion which we haven't got to yet and is best achieved in other ways).

The unary − operator negates its operand, which can be a useful operation.

Unary operators: demo

```java
public class UnaryDemo1 {
    public static void main (String[] args) {

        // the default sign is positive
        int i1 = 10;

        // this means the same as the previous assignment
        int i2 = +10;

        int i3 = -1;

        // has no effect so i4 = i3
        int i4 = +i3;

        // i5 is set to 5
        int i5 = 5;

        // i5 is set to -5
        i5 = -i5;

        // i6 is set to -5
        int i6 = -(-i5);

        System.out.println("i1 = " + i1 + ".");
        System.out.println("i2 = " + i2 + ".");
        System.out.println("i3 = " + i3 + ".");
        System.out.println("i4 = " + i4 + ".");
        System.out.println("i5 = " + i5 + ".");
        System.out.println("i6 = " + i6 + ".");
    }
}
```

Listing 24 : UnaryDemo1.java

Unary operators: demo

```
$ javac UnaryDemo1.java
$ java UnaryDemo1
i1 = 10.
i2 = 10.
i3 = -1.
i4 = -1.
i5 = -5.
i6 = -5.
```

Listing 25 : Terminal commands

Unary operators

The next program demonstrates the use of the unary increment and decrement operators. They are straightforward but both have a prefix and a postfix version.

The different meanings of the prefix and postfix versions will become clear when looking at the next program but, in essence, when the operators are used as part of an assignment the postfix version evaluates to its *existing* value, which is then used in the assignment operation, and **then** increments or decrements its operand, whilst the prefix version **first** increments or decrements its operand and then this new value is used in the assignment.

Unary operators: demo

```java
public class UnaryDemo2 {
    public static void main (String[] args) {

        int i1 = 2;

        // i1 now = 3
        i1++;

        int i2 = 5;

        // i2 now = 6
        ++i2;

        // i3 is set to 3, i1 is set to 4
        int i3 = i1++;

        // i4 is set to 4, i3 is set to 4
        int i4 = ++i3;

        // i5 is set to ???
        int i5 = ++i4 * 2 + 3;

        // i6 is set to ???
        int i6 = i1++ + +i2 - ++i4;

        System.out.println("i1 = " + i1 + ".");
        System.out.println("i2 = " + i2 + ".");
        System.out.println("i3 = " + i3 + ".");
        System.out.println("i4 = " + i4 + ".");
        System.out.println("i5 = " + i5 + ".");
        System.out.println("i6 = " + i6 + ".");
    }
}
```

Listing 26 : UnaryDemo2.java

Unary operators: demo

```
$ javac UnaryDemo2.java
$ java UnaryDemo2
i1 = 5.
i2 = 6.
i3 = 4.
i4 = 6.
i5 = 13.
i6 = 4.
```

Listing 27 : Terminal commands

Unary operators

The final unary operator we are covering is the negation operator. This negates boolean expressions, which means if its operand is `true` then it evaluates to `false` and vice versa.

Unary operators: demo

```java
public class UnaryDemo3 {
    public static void main (String[] args) {

        boolean b1 = false;
        boolean b2 = true;

        // b1 is now true
        b1 = !b1;

        // b1 is now false
        b1 = !b1;

        // b2 is false
        b2 = b1;

        // b2 is true
        b2 = !b1;

        // b2 is ???
        b2 = !(!(!b1));

        System.out.println("b1 = " + b1 + ".");
        System.out.println("b2 = " + b2 + ".");
    }
}
```

Listing 28 : UnaryDemo3.java

Unary operators: demo

```
$ javac UnaryDemo3.java
$ java UnaryDemo3
b1 = false.
b2 = true.
```

Listing 29 : Terminal commands

Boolean expressions

Boolean expressions evaluate to `true` or `false`. They are used extensively in control, selection and iteration, which we will be coming to shortly.

Firstly, we will look at another set of operators: the *relational and equality operators*. These are used in boolean expressions.

Relational and equality operators

There are a number of relational and equality operators available in Java. They are shown in the table below.

Operator	Function
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equals
!=	Not equal to

Table 6 : The relational and equality operators

Note that the equality operator is ==. This is somewhat unfortunate and leads to confusion and bugs (particularly in more 'relaxed' languages like C).

Relational and equality operators

```java
public class RelationalDemo1 {
    public static void main (String[] args) {

        int i1 = 3;
        int i2 = 5;
        int i3 = -2;

        // b1 = true
        boolean b1 = i1 < i2;
        // b2 = false
        boolean b2 = i3 < i3;
        // b3 = true
        boolean b3 = i3 <= i3;
        // b4 = true
        boolean b4 = i1 != i3;
        // b5 = ???
        boolean b5 = i2 == i3 == b4;
        // b6  = ???
        boolean b6 = true == false;
        // b7 = ???
        boolean b7 = !(b3 == (i3 == i2) == (i1 != 7));

        System.out.println("b1 = " + b1 + ".");
        System.out.println("b2 = " + b2 + ".");
        System.out.println("b3 = " + b3 + ".");
        System.out.println("b4 = " + b4 + ".");
        System.out.println("b5 = " + b5 + ".");
        System.out.println("b6 = " + b6 + ".");
        System.out.println("b7 = " + b7 + ".");
    }
}
```

Listing 30 : RelationalDemo1.java

Relational and equality operators

```
$ javac RelationalDemo1.java
$ java RelationalDemo1
b1 = true.
b2 = false.
b3 = true.
b4 = true.
b5 = false.
b6 = false.
b7 = true.
```

Listing 31 : Terminal commands

Floating point values revisited

Earlier we saw that we will not always get the mathematically correct result when using arithmetical operators with floating point values due to the finite nature of the IEEE754 representation. This can also lead to unexpected results when using floating point results in boolean expressions, as the next slide shows. Programmers must always be mindful of the limitations of the IEEE754 representation of floating point values.

Relational and equality operators

```
public class FloatComparisonDemo1 {
    public static void main (String[] args) {

        float f = 0.333333f;

        f = f / 10;

        boolean b = f == 0.0333333f;

        System.out.println("b = " + b);

        System.out.println("f = " + f);
    }
}
```

Listing 32 : FloatComparisonDemo1.java

Relational and equality operators

We would expect the output to show that b is `true` but it doesn't.

```
$ javac FloatComparisonDemo1.java
$ java FloatComparisonDemo1
b = false
f = 0.033333298
```

Listing 33 : Terminal commands

Operator precedence revisited

We can now update the table showing order of precedence with the relational and equality operators

Type	Precedence
Postfix unary	*expr*++ *expr*- -
Unary	++*expr* - -*expr* +*expr* -*expr* !*expr*
Multiplicative	* / %
Additive	+ -
Relational	< > <= >=
Equality	== !=

Table 7 : Order of precedence

Conditional operators

The Conditional operators are binary operators (requiring two operands) that enable the combination of boolean expressions into more complex, compound boolean expressions. The two conditional operators are AND and OR but you cannot use those terms in Java. The table below shows the operators and their syntax in Java.

Operator	Java syntax
AND	&&
OR	||

Table 8 : The conditional operators

An expression connected with AND is `true` if all of the operands are `true`.

An expression connected with OR is `true` if any of the operands are `true`.

Conditional operators: demo

```java
public class ConditionalDemo1 {
    public static void main (String[] args) {

        // b1 is true
        boolean b1 = true && true;

        // b2 is false
        boolean b2 = true && false;

        // b3 is false
        boolean b3 = false && true;

        // b4 is false
        boolean b4 = false && false;

        //b5 is true
        boolean b5 = true || true;

        //b6 is true
        boolean b6 = true || false;

        //b7 is true
        boolean b7 = false || true;

        //b8 is false
        boolean b8 = false || false;
    }
}
```

Listing 34 : ConditionalDemo1.java

Conditional operators: demo

```java
public class ConditionalDemo2 {
    public static void main (String[] args) {

        int i1 = 3;
        int i2 = 6;
        int i3 = 9;

        // b1 = false
        boolean b1 = i1 < i2 && i3 < i2;

        // b2 = true
        boolean b2 = i1 != 3 || i2 != 5;

        // b3 = false
        boolean b3 = i2 == i3 || (b2 && b1);

        // b4 is ???
        boolean b4 = i1 > i2 || i2 > i3 || i1 == i3 || i3
            >= 8;

        // b5 is ???
        boolean b5 = i2 < i3 && (i3 == i2 || i1 <= 3) &&
            b4 != false;

        System.out.println("b1 = " + b1 + ".");
        System.out.println("b2 = " + b2 + ".");
        System.out.println("b3 = " + b3 + ".");
        System.out.println("b4 = " + b4 + ".");
        System.out.println("b5 = " + b5 + ".");
    }
}
```

Listing 35 : ConditionalDemo2.java

Conditional operators: demo

```
$ javac ConditionalDemo2.java
$ java ConditionalDemo2
b1 = false.
b2 = true.
b3 = false.
b4 = true.
b5 = true.
```

Listing 36 : Terminal commands

Operator precedence revisited

We can now update the table showing order of precedence with the conditional operators.

Type	Precedence
Postfix unary	*expr*++ *expr*- -
Unary	++*expr* - -*expr* +*expr* -*expr* !*expr*
Multiplicative	* / %
Additive	+ -
Relational	< > <= >=
Equality	== !=
AND	&&
OR	||

Table 9 : Order of precedence

Terminology review

Java programs are built from *expressions*, *statements* and *blocks*.

An expression is a string of identifiers and operators that evaluates to a single value. The value itself could be 'complex'. In a previous program we saw the following expression

```
i1++ + 2
```

This group of identifiers and operators will evaluate to a single value (that depends on the value of the variable `i1`).

Statements were mentioned earlier. They terminate with a semicolon and constitute a complete 'sentence' in the Java programming language. Statements are built from expressions but also from other constructs such as conditional statements (which we will be looking at shortly). In all of the programs we have seen so far, each line terminating in a semicolon contains a statement.

When we look at conditional and other types of statement, we will consider blocks. Blocks are larger groupings of statements.

Strings, Input/Output

In order to create programs that actually *do* something we will now look at some Java reference types that enable the use of strings (sequences of characters) and basic input and output.

If you are new to Java, some of these concepts will make more sense when we cover *classes* in detail.

Strings

Most of the programs we have seen so far have used strings but 'implicitly': the `String` type wasn't explicitly mentioned. All of the screen output statements in the programs seen so far use `String`.

A `String` is a sequence of characters and can be used to hold words, names, numbers (as strings) and, indeed, any sequence of characters. Strings are very convenient in Java (compared to a language like C) and are easy to use. The `String` class has many *methods* so that we can do useful things with strings. Methods will be discussed in detail when we look at classes.

As usual, we will now look at some code.

String: demo

```
public class StringDemo1 {
    public static void main (String[] args) {

        // a declaration. This String cannot be used yet.
        String s1;

        // now it can (initialised to the empty string)
        s1 = "";

        // this String can be used as it was initialised.
        String s2 = "abc";

        String s3 = "contains numbers 123";
        String s4 = "%^0))";

        String s5 = "Hacker";
        String s6 = "Dog";
        String s7 = "T.";

        // String concatenation
        String s8 = s5 + " " + s7 + " " + s6;

        String s9 = "4" + "6";

        // String output
        System.out.println("s1 = " + s1);
        System.out.println("s2 = " + s2);
        System.out.println("s3 = " + s3);
        System.out.println("s4 = " + s4);
        System.out.println("s5 = " + s5);
        System.out.println("s8 = " + s8);
        System.out.println("s9 = " + s9);

    }
}
```

Listing 37 : StringDemo1.java

String concatenation

Note that the '+' operator is *overloaded* for String. This means that '+' has been given a meaning for String in addition to its meaning for numerical values, and that meaning is quite different. With String, '+' means *concatenate*. If two strings are concatenated then a new string is formed that contains all of the characters from the first string followed by all of the characters from the second string.

String operations

There are other useful `String` operations. These include

- Finding the length of a `String` (the number of characters in the `String`).
- Extracting a substring from a `String`.
- Finding the starting position of a substring within a `String`, including the position of a particular character.
- Getting the character at a particular position in a `String`.
- Checking if two strings are equal.

We will now look at examples of these operations. Note that there are **many** methods in the `String` class. Have a look at the official Java API documentation to see them.

String length: demo

```
public class StringDemo2 {
    public static void main (String[] args) {

        String s = "David Bowie";
        int slen = s.length();
        System.out.println("The String is " + slen + "
            characters long.");
    }
}
```

Listing 38 : StringDemo2.java

```
$ javac StringDemo2.java
$ java StringDemo2
The String is 11 characters long.
```

Listing 39 : Terminal commands

Substring: demo

```java
public class StringDemo3 {
    public static void main (String[] args) {

        String s = "David Bowie";
        String fname = s.substring(0, 5);
        System.out.println("First name is: " + fname);
    }
}
```

Listing 40 : StringDemo3.java

```
$ javac StringDemo3.java
$ java StringDemo3
First name is: David
```

Listing 41 : Terminal commands

The first *parameter* of this version of `substring()` is the start character position of the required substring, and the second parameter is the final character position $+ 1$.

Substring: demo

```
public class StringDemo4 {
    public static void main (String[] args) {

        String s = "David Bowie";
        String lname = s.substring(6);
        System.out.println("Last name is: " + lname);
    }
}
```

Listing 42 : StringDemo4.java

```
$ javac StringDemo4.java
$ java StringDemo4
Last name is: Bowie
```

Listing 43 : Terminal commands

This version of `substring()` takes only one parameter: the start character position. The end character position will be the end of the `String`.

indexOf(): demo

```
public class StringDemo5 {
    public static void main (String[] args) {

        String s = "David Bowie";
        int spacePos = s.indexOf(' ');
        System.out.println("The space character is in
            position: " + spacePos);
    }
}
```

Listing 44 : StringDemo5.java

```
$ javac StringDemo5.java
$ java StringDemo5
The space character is in position: 5
```

Listing 45 : Terminal commands

`indexOf()` finds the position of the first instance of the character passed as an argument (the space character in this example). There are many related methods to this one. See the official Java API documentation.

indexOf(): demo

```
public class StringDemo6 {
    public static void main (String[] args) {

        String s = "David Bowie";
        int spacePos = s.indexOf(' ');
        String fname = s.substring(0, spacePos+1);
        System.out.println("First name is " + fname);
    }
}
```

Listing 46 : StringDemo6.java

```
$ javac StringDemo6.java
$ java StringDemo6
First name is David
```

Listing 47 : Terminal commands

Note that `indexOf()` allows us to create a more robust version of the first name finder. This version supports variable first name lengths!

charAt(): demo

```
public class StringDemo7 {
    public static void main (String[] args) {

        String s = "David Bowie";

        char c1 = s.charAt(0); // D
        char c2 = s.charAt(4); // d
    }
}
```

Listing 48 : StringDemo7.java

`charAt()` returns the character at the given position, as a `char` not as a `String`.

equals(): demo

What will the output be?

```
public class StringDemo8 {
    public static void main (String[] args) {

        String s1 = "David Bowie";
        String s2 = "Iggy Pop";
        String s3 = "David Bowie";

        boolean b1 = s1.equals(s2);
        boolean b2 = s1.equals(s3);

        System.out.println("b1 is " + b1);
        System.out.println("b2 is " + b2);
    }
}
```

Listing 49 : StringDemo8.java

equals(): demo

What will the output be?

```java
public class StringDemo9 {
    public static void main (String[] args) {

        String s1 = new String("David Bowie");
        String s2 = new String("David Bowie");

        boolean b1 = s1.equals(s2);
        boolean b2 = s1 == s2;

        System.out.println("b1 is " + b1);
        System.out.println("b2 is " + b2);
    }
}
```

Listing 50 : StringDemo9.java

Be careful with == and strings since it compares *references* not values. Always use equals().

Screen output

So that programs can communicate with the outside world, it is useful to be able to send messages from programs to output devices, for example, the computer screen. This is useful for communicating the program's actions and for debugging and for other purposes. Java makes printing to the screen (and sending messages via other channels) straightforward.

Most of the programs we have looked at so far have contained a line such as

```
System.out.println("b1 = " + b1 + ".");
```

`System` is a built-in Java class that provides some 'global' methods and fields that are useful to any program. One of these fields is a `PrintStream` object that is bound to the *console* - the standard output device (it can be bound to a different device, depending on the system).

Screen output

println() is a method in the PrintStream class that writes a line of text. In the case of System.out it will send this line of text to the console. The ln appended to the word print means that the stream will also send a newline character to the output device, meaning that the 'cursor' will move to the next line after printing (like pressing the 'return' key on a computer keyboard).

println() is overloaded for all of the primitive types, which means it can print those directly without them needing to be converted to String first. println can also handle mixed types.

In this line of code

```
System.out.println("b1 = " + b1 + ".");
```

"b1 = " is a String; b1 is a boolean variable; and "." is a String. All three have been concatenated with +.

Keyboard input

It is also useful to be able to submit information to a program from the keyboard. For this, will use another standard Java class called `Scanner`.

`Scanner` has methods to read particular types from the keyboard. It has a slightly more complex usage in that you must create a Scanner `object` to use it. Again, this will become clearer when we get to classes.

A `Scanner` object is created like this

```
Scanner scanner = new Scanner(System.in);
```

However, at the top of the file we need an `import statement` so that the compiler knows where to find the definition of `Scanner`

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

I/O: demo

```java
import java.util.Scanner;

public class InputDemo1 {
    public static void main (String[] args) {

        Scanner scanner = new Scanner(System.in);

        // read an int from the keyboard
        System.out.println("Please enter an integer.");
        int input = scanner.nextInt();
        System.out.println("You entered: " + input);

        // read a float from the keyboard
        System.out.println("Please enter a float.");
        float finput = scanner.nextFloat();
        System.out.println("You entered: " + finput);

        // read a String from the keyboard
        System.out.println("Please enter your name.");
        String name = scanner.nextLine();
        System.out.println("You entered: " + name);
    }
}
```

Listing 51 : InputDemo1.java

I/O: demo

```
$ javac InputDemo1.java
$ java InputDemo1
Please enter an integer.
123
You entered: 123
Please enter a float.
2.4
You entered: 2.4
Please enter your name.
You entered:
```

Listing 52 : Terminal commands

What can't really be seen on this static demo is that the program skips the request to 'Please enter your name'. This is because the `next*()` methods such as `nextFloat()` do **not** remove the newline character that was entered via the keyboard. This can cause problems when subsequent attempts to read the keyboard read the newline character that is still sitting in the keyboard buffer. This problem can be avoided with a slightly clunky hack that simply reads the buffer *again* to remove the newline character.

I/O: demo

```java
import java.util.Scanner;

public class InputDemo2 {
   public static void main (String[] args) {

      Scanner scanner = new Scanner(System.in);

      // read an int from the keyboard
      System.out.println("Please enter an integer.");
      int input = scanner.nextInt();
      System.out.println("You entered: " + input);

      // read a float from the keyboard
      System.out.println("Please enter a float.");
      float finput = scanner.nextFloat();
      System.out.println("You entered: " + finput);

      scanner.nextLine(); // removes the newline
          character from the buffer

      // read a String from the keyboard
      System.out.println("Please enter your name.");
      String name = scanner.nextLine();
      System.out.println("You entered: " + name);
   }
}
```

Listing 53 : InputDemo2.java

Control flow

The programs seen so far have simply started at the first line of code in the `main` method and then proceeded to execute each line of the code in sequence before ending after the last statement.

However, real programs almost always require statements that transfer the *flow* of the program, either to a different part of the program completely, to a point 'backwards' in the code or a point 'forwards'. Programs usually need to perform some actions only in the case that certain conditions are true or only in the case when they are false. Programs also need to repeat certain actions for either a fixed number of times, a variable number of times, until some condition is true or false, etc.

We will now look at *control statements* which allow our programs to behave in these ways.

The `if` statement

The `if` statement allows us to perform some actions if a condition is true. The condition can be a complex boolean expression of the type we have seen.

The format of the `if` statement is as follows

```
if (boolean-expression-is-true)
    then-consequence;
```

Like other statements, the `if` statement is terminated by a semicolon.

`if`: demo

```java
import java.util.Scanner;

public class ControlDemo1 {
    public static void main (String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.println("Please enter an integer
            greater than 100.");

        int input = scanner.nextInt();

        if (input > 100)
            System.out.println("Well done!");
    }
}
```

Listing 54 : ControlDemo1.java

if: demo

```
$ javac ControlDemo1.java
$ java ControlDemo1
Please enter a positive integer greater than 100.
12
ian@ian-laptop:~/Documents/tex/SWW2$ java ControlDemo1
Please enter a positive integer greater than 100.
120
Well done!
```

Listing 55 : Terminal commands

This listing shows two runs of the program. In run one, the integer entered is less than 100 hence no output is produced. The code in the `if` statement is skipped. In run two, an integer greater than 100 is entered therefore the code in the `if` statement is executed.

The `if` statement

What if we would like some output in the case the condition is `false`? In this case when the user does not enter an integer greater than 100.

if: demo

Does this look like it will produce the behaviour we seek?

```java
import java.util.Scanner;

public class ControlDemo2 {
    public static void main (String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.println("Please enter an integer
            greater than 100.");

        int input = scanner.nextInt();

        if (input > 100)
            System.out.println("Well done!");
        System.out.println("Bad luck!");
    }
}
```

Listing 56 : ControlDemo2.java

if: demo

```
$ javac ControlDemo2.java
$ java ControlDemo2
Please enter an integer greater than 100.
80
Bad luck!
ian@ian-laptop:~/Documents/tex/SWW2$ java ControlDemo2
Please enter an integer greater than 100.
130
Well done!
Bad luck!
```

Listing 57 : Terminal commands

This listing again shows two runs of the program. In run one the program appears to be doing what we want but in the second the flaw is revealed. The second output line is **always** produced.

The `if-else` statement

This problem is easily fixed by simply using an `if-else` statement instead.

The format of the `if-else` statement is as follows

```
if (boolean-expression-is-true)
    then-consequent;
else what-to-do-if-the-expression-is-false;
```

With the `if-else` statement, we can include a statement that will only be executed if the condition is `false`. We place it in the `else` clause as shown.

if-else: demo

This version will produce the behaviour we seek.

```
import java.util.Scanner;

public class ControlDemo3 {
    public static void main (String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.println("Please enter an integer
            greater than 100.");

        int input = scanner.nextInt();

        if (input > 100)
            System.out.println("Well done!");
        else System.out.println("Bad luck!");
    }
}
```

Listing 58 : ControlDemo3.java

`if`: demo (revisited)

What will this program do in both cases?

```
import java.util.Scanner;

public class ControlDemo4 {
    public static void main (String[] args) {

        boolean answerCorrect = false;

        Scanner scanner = new Scanner(System.in);

        System.out.println("Please enter an integer
            greater than 100.");

        int input = scanner.nextInt();

        if (input > 100)
            System.out.println("Well done!");
            answerCorrect = true;

        if (answerCorrect)
            System.out.println("You have reached level
                5.");
    }
}
```

Listing 59 : ControlDemo4.java

if: demo

```
$ javac ControlDemo4.java
$ java ControlDemo4
Please enter an integer greater than 100.
101
Well done!
You have reached level 5.
ian@ian-laptop:~/Documents/tex/SWW2$ java ControlDemo4
Please enter an integer greater than 100.
77
You have reached level 5.
```

Listing 60 : Terminal commands

This is an incorrect result. This is because the boolean value `answerCorrect` is set to true whatever the truth of the condition in the `if` statement because the `if` statement ends at the first semicolon. Only one 'line of code' can appear in an `if` statement unless a *block* is created.

Blocks

Blocks contain lines of code that are somehow related to each other. Blocks of code have already been used in the example programs so far. The definition of a class must be contained in a block and methods must be contained in a block, hence the block around the main method. Blocks are defined by a start brace and end brace { }. For example

```java
import java.util.Scanner;

public class ControlDemo4 {
    public static void main (String[] args) {

        boolean answerCorrect = false;

        Scanner scanner = new Scanner(System.in);

        System.out.println("Please enter an integer
            greater than 100.");

        int input = scanner.nextInt();

        if (input > 100)
            System.out.println("Well done!");
            answerCorrect = true;

        if (answerCorrect)
            System.out.println("You have reached level
                5.");
    }
}
```

Listing 61 : ControlDemo4.java

Blocks

`if` statements and if-else statements can use blocks in order to allow more than one line of code to be included in the statement.

The example program on the next slide fixes the problem with `ControlDemo4`.

Blocks

```
import java.util.Scanner;

public class ControlDemo5 {
    public static void main (String[] args) {

        boolean answerCorrect = false;

        Scanner scanner = new Scanner(System.in);

        System.out.println("Please enter an integer
            greater than 100.");

        int input = scanner.nextInt();

        if (input > 100) {
            System.out.println("Well done!");
            answerCorrect = true;
        }

        if (answerCorrect) {
            System.out.println("You have reached level
                5.");
        }
    }
}
```

Listing 62 : ControlDemo5.java

Blocks

In addition to the blocks defining the class and the `main` method, there are now blocks defining the extent of the two `if` statements. The first `if` statement will now work correctly since the two lines of code in the statement will only be executed if the condition is `true`.

Note that, strictly speaking, the block defined for the second `if` statement is unnecessary since it only contains one line of code. There is nothing wrong with adding a block here, however, and it might prevent an error if later the `if` statement is edited and more lines of code are added.

Iteration

We often need to execute a section of code a number of times rather than simply executing it or not executing it, as with the `if` constructs.

The number of *iterations* we perform of the code may be fixed or variable.

There are a number of iteration constructs in Java: `while`, `do-while` and `for`. We will now look at these.

`while`

The `while` loop requires a boolean expression as a condition, exactly as with the `if` constructs.

Iteration: demo

What will the output be?

```
public class IterationDemo1 {
    public static void main (String[] args) {

        int i = 0;

        while (i < 5) {
            System.out.println("The value of i is " + i);
            i++;
        }
    }
}
```

Listing 63 : IterationDemo1.java

Iteration: demo

We can iterate a variable number of times.

```java
import java.util.Scanner;
public class IterationDemo2 {
    public static void main (String[] args) {

        System.out.println("Please enter an integer
            value.");
        Scanner s = new Scanner(System.in);
        int numIterations = s.nextInt();

        int i = 0;

        while (i < numIterations) {
            System.out.println("The value of i is " + i);
            i++;
        }
    }
}
```

Listing 64 : IterationDemo2.java

`do while`

The `do while` loop is a more rarely used loop. Its function can be performed with a regular `while` loop. It is a more natural construct, however, when the loop must iterate *at least once*.

The program on the next slide demonstrates how `do while` can be used to get input at least once.

Try and figure out how this program works and what behaviour it will produce.

Iteration: demo

```
import java.util.Scanner;
public class IterationDemo3 {
    public static void main (String[] args) {

        Scanner s = new Scanner(System.in);
        int numIterations = 0;

        do {

            System.out.println("Please enter a number. -1
                to quit.");
            numIterations = s.nextInt();

            int i = 0;

            while (i < numIterations) {
                System.out.println("The value of i is " +
                    i);
                i++;
           }

        } while (numIterations >= 0);
    }
}
```

Listing 65 : IterationDemo3.java

`for`

The `for` loop is another 'convenience' construct in that its function can be performed with the `while` loop (although there is a newer version of the `for` loop that we will see later in the course and that is a useful construct.)

The advantage of the `for` loop is that it 'gathers together' the control parameters of the loop.

Iteration: demo

```
public class IterationDemo4 {
    public static void main (String[] args) {

        for (int i = 0; i < 5; i++) {
            System.out.println("The value of i is " + i);
        }
    }
}
```

Listing 66 : IterationDemo4.java

This works in the same way as the `while` loop that did the same thing. With the `for` loop, the 'header' of the loop contains the index variable (here, `i`), the condition, and the operation that moves the loop to the next iteration (here, `i++`).

Iteration: demo

```
public class IterationDemo5 {
    public static void main (String[] args) {

        int i = 0;

        for (; i < 5;) {
            System.out.println("The value of i is " + i);
            i++;
        }
    }
}
```

Listing 67 : IterationDemo5.java

Note that we can force the `for` loop to 'degenerate' to the `while` loop by moving the initialisation of the index variable, and the iteration operation, out of the loop header.

Watch out!

What will the output be?

```
public class IterationDemo6 {
    public static void main (String[] args) {

        int i = 0;

        while (i < 7) {
            System.out.println("The value of i is " + i);
        }
    }
}
```

Listing 68 : IterationDemo6.java

A **very** easy mistake to make.