

2. WRITING JAVA CODE

Starting to write Java code	1
The CodePad in BlueJ	1
Variables	2
Operators	4
Expressions	5
Statements	6
Output	7
Wrap up	7
Appendix: Java operator precedence	8

Starting to write Java code

In the previous lecture you saw that a class is created by writing Java source code. You will soon see how to create a complete Java class, but before that you need to learn a little bit about how to write Java code.

A programming language is like a natural language that we speak or write in that it has components (in a natural language these would be the words, sentences, paragraphs, and so on) and rules, or **syntax**. When you speak you need to follow the rules or the person you are speaking to will not understand you. When you write a program, you also need to follow the rules for the programming language, otherwise the computer will not be able to understand.

When you listen to someone speak you can often work out or guess what they mean even if they don't follow the rules of language very closely. Computers can't do that with instructions, though, so programming language syntax is very strict.

As you saw previously, Java code is actually translated into something that the computer can execute. This process is called compiling, and also does the job of checking that your syntax is correct.

In this lecture you will see some short fragments of Java code that will demonstrate the basic components and rules of the language. Once you are familiar with these you will be ready to write classes, and then programs that make use of classes.

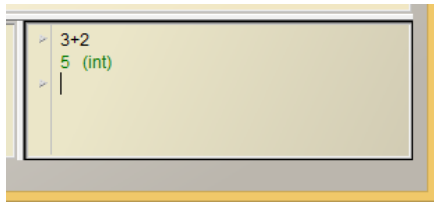
The CodePad in BlueJ

You can experiment with short fragments of Java code using the **CodePad** feature in BlueJ. You need to have a project open in BlueJ – you can open the *emptyproject* project that is available for this lecture. This project has no classes, but it allows you use the CodePad, and you need it open to try out all the examples in this lecture.

Select Show CodePad from the View menu, and the CodePad will appear as a separate area to the right of the object bench. You can type something here and immediately see its effect. To try this out, type the following and press Enter.

3 + 2

This simply causes two numbers to be added and the result should be displayed.



```
> 3+2
5 (int)
> |
```

These lectures notes and some later ones contain some practice exercises using the CodePad. In each lecture, the first exercise will tell you what project to open in BlueJ and the other exercises will assume that the project is still open. The exercises in this lecture assume that the project *emptyproject* is still open.

Variables

In the previous lecture you saw some examples of information (or data) that is stored and used in a program. Objects have properties, for example, and information is passed to methods as parameters. You will see many other uses for information in programs throughout this module.

In general, programs store information using **variables**. Variables are stored in the computer's memory, and can be retrieved from that memory when needed by the program.

Variable names

A variable is an item of information which has a **value** and is named by an **identifier**. It is called a variable because its value may change as the program runs. By giving it an identifier you can always **refer** to it by name, so that its value can be retrieved from memory.

A variable name in Java must be a legal identifier – a series of characters that begins with a letter. Example: **myVariable**. The name cannot contain spaces. The name must not be a Java keyword (e.g. *class*), or *true*, *false* or *null*.

By convention variable names start with a lower case character. If the name consists of more than one word, each following word begins with an uppercase letter.

Types

A variable has a **type** that determines what kind of values it can hold. As you saw in the previous lecture there can be many different types of data in a program, such as numbers (integers, floating point numbers), characters, and so on.

It is important for the type of a variable to be defined, as the computer needs to store its value. As the value may change, the computer needs to reserve enough space in memory to store any valid value of that type. If a variable is an integer, then the space reserved must be able to hold the minimum and maximum possible values for an integer and any value in between, for example.

In Java, some types of information are stored as **primitive types**. This means that a single piece of information is stored as the value for a variable which has a primitive type. Java has a key word to identify each type. Primitive data types include:

keyword	description
byte	Byte-length integer
short	Short integer
int	Integer
long	Long integer
float	Single-precision floating point
double	Double-precision floating point
char	A single Unicode character
boolean	A Boolean value (true or false)

There are different sizes of integer types. A byte uses a single byte of memory and can only store values within the range -128 to 127. The other integer types use more bytes of memory to store each value and can store larger numbers. Float and double types store floating point numbers, and differ in the number of decimal places of precision they can store.

Another useful data type for variables is **String**. A string stores a collection of characters, so it is not a primitive type.

Java also allows variables which have object types. Object type variables don't store single values. Instead, they hold references to objects. We will look at object types later in the module.

Declaring a variable

To give a variable a type and a name, and make sure space is reserved for it in memory, you write a variable declaration, e.g.

```
int myVariable;
```

Note that a variable declaration is a **statement** in Java that needs to end in a semi-colon. We will look at statements in more detail shortly.

Initialising a variable

Variables can be **initialised** with an assignment statement when they are declared, e.g.

```
int myVariable = 10;
```

Other types of variable

Here are some examples of how to declare and initialise other types of variable. Note the way the value is written in each case. For example, `float` values can be written as a decimal number followed by the letter `f`, while `double` values can be just the number. `char` values are enclosed in single quotes, `String` values in double quotes. `boolean` variables can only have the values `true` and `false`.

```
float myFloat = 10.1f;
double myDouble = 10.1;
boolean myBoolean = true;
char myChar = 'j';
String myString = "java";
```

CodePad – try the following in the CodePad

- Declare and initialise an `int` variable
- Type the name of the variable – you should see the value it holds
- Declare and initialise examples of each of the other variable types listed above
- For each one type the name of the variable and check that it holds the value you initialised it with

```

int myVariable = 10;
myVariable
10 (int)
float myFloat = 10.1f;
myFloat
10.1 (float)
double myDouble = 10.1;
myDouble
10.1 (double)

```

Operators

A variable is not much use unless you do something with it, so now we want to start to write code that does more than simply assign values. Firstly, you need to learn about operators. An **operator** performs a function on one, two or three **operands**. An operand can be, for example, a variable name or a literal value. Java operators include the following:

Arithmetic operators

Operator	Example	Description
+	<code>x + y</code>	Adds <code>x</code> and <code>y</code> , also concatenates strings
-	<code>x - y</code>	Subtracts <code>y</code> from <code>x</code>
*	<code>x * y</code>	Multiplies <code>x</code> by <code>y</code>
/	<code>x / y</code>	Divides <code>x</code> by <code>y</code>
%	<code>x % y</code>	Gives remainder on dividing <code>x</code> by <code>y</code>
Shortcut operators		
++	<code>x++</code>	Increments <code>x</code> by 1; gives value of <code>x</code> before increment
++	<code>++x</code>	Increments <code>x</code> by 1; gives value of <code>x</code> after increment
--	<code>x--</code>	Decrements <code>x</code> by 1; gives value of <code>x</code> before decrement
--	<code>--x</code>	Decrements <code>x</code> by 1; gives value of <code>x</code> after decrement

Note that if an integer and a floating-point number are used as operands to a single arithmetic operator, the result is floating point.

Relational operators

Operator	Example	Description
>	<code>x > y</code>	Returns true if <code>x</code> is greater than <code>y</code>
>=	<code>x >= y</code>	Returns true if <code>x</code> is greater than or equal to <code>y</code>
<	<code>x < y</code>	Returns true if <code>x</code> is less than <code>y</code>
<=	<code>x <= y</code>	Returns true if <code>x</code> is less than or equal to <code>y</code>
==	<code>x == y</code>	Returns true if <code>x</code> is equal to <code>y</code>
!=	<code>x != y</code>	Returns true if <code>x</code> is not equal to <code>y</code>

Conditional operators

Operator	Example	Description
&&	a && b	Returns true if a and b are true
	a b	Returns true if a or b is true
!	!a	Returns true if a is false

Assigning using operators

You can **assign** a new value to a variable once it has been declared. The new value replaces the currently value if the variable has been initialised. The basic assignment operator is =, e.g.

`x = 3` assigns a value to a variable, replacing its current value

`x = y` assigns the value of variable `y` to variable `x`

The = sign has a special meaning in Java programming which is different to its meaning in maths. It **assigns the value on its right hand side to the variable named on its left** hand side. It does **not** simply mean that the two sides are equal.

In Java there are also some shortcut assignment operators. For example

`x = x + 3`

which adds 3 to the value of the variable `x`, can be written as

`x += 3`

Similarly, we can have `x -= 1`, `x *= 10`, `x /= 2`, `x %= 3`

If you simply want to increment or decrement a variable by 1 you can use one of the shortcut arithmetic operators, for example

`x++`

Unlike the other arithmetic operators, `++` and `--` can assign a new value to a variable without the use of an assignment operator.

Expressions

An **expression** is a series of items, which can include literal values, variables, operators and method calls that **evaluates to a single value**. You sometimes need to use brackets to ensure that operators are applied in the order you want where this differs from the default **operator precedence** – for example `*` and `/` are applied before `+` and `-` by default in an expression (for reference, there is a complete table of operator precedence at the end of this document).

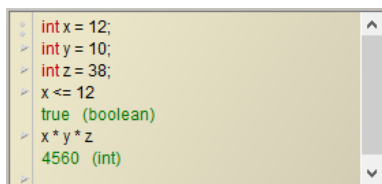
For example:

`x <= 12` evaluates to Boolean `true` if `x` is less than or equal to 12

<code>x * y * z</code>	evaluates to result of multiplication
<code>(y > 3) && (y < 10)</code>	evaluates to Boolean <code>true</code> if <code>y</code> is between 4 and 9
<code>x + 3 * y</code>	evaluates to 42 (<code>3 * y</code> is evaluated first)
<code>(x + 3) * y</code>	evaluates to 150 (<code>x+3</code> is evaluated first)
<code>++x</code>	increments <code>x</code> and evaluates to the new value

CodePad

- Clear the CodePad – you can do this by clicking the Compile button
- Declare and initialise `int` variables `x=12`, `y=10`, `z=38`
- Enter the expressions listed above and check that they have the values expected (note that you don't type a semicolon (;) when entering an expression). Do them in order – the last one changes the value of `x`!



```

int x = 12;
int y = 10;
int z = 38;
x <= 12
true (boolean)
x * y * z
4560 (int)

```

Statements

A *statement* forms a complete unit of execution that instructs the computer to perform an action such as assigning a variable. A statement is terminated with a semicolon (;). You have already used statements to declare and initialise variables, but there are many other actions that can be performed using statements.

For example:

<code>x = x + 10;</code>	assignment
<code>x--;</code>	decrementing
<code>double y = 12.345;</code>	declaration and assignment

CodePad – clear the CodePad and try the following:

- Declare and initialise `int` variable `x=0`
- Enter the statements listed above (note that you do type the ; when entering a statement) – entering a statement in CodePad causes that statement to be executed
- How can you see the effects of entering these statements?

Note that an **expression represents a value**, while a **statement actually does something**. A statement may contain expressions. For example:

`x + 10`

is an expression whose value is 10 more than the value of the variable `x`, while

`x = x + 10;`

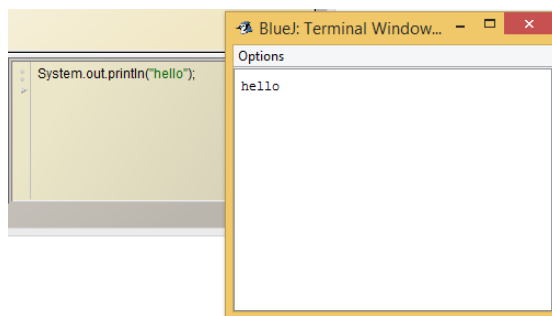
is a statement which assigns the value of that expression to the variable `x`. A statement does not have a value.

Output

It is often necessary for a program to print **output** for the user. In Java you can print like this:

```
System.out.println("hello");
```

`System.out.println` writes its output to a separate window called the Terminal Window. It works in any Java program, not just in the CodePad.



CodePad - Clear the CodePad and try the following:

- Declare and initialise int variable `x=3`
- Enter a statement to print the string "hello"
- Enter a statement to print the value of the variable `x`

Wrap up

You've been introduced in this lecture to the following concepts:

Variables, Operators, Expressions, Statements, Output

In the next lecture you will start to learn how to write the source code to create your own classes, making use of the Java language concepts in this lecture.

Appendix:Java operator precedence

Note that this is a complete listing of operator precedence, and includes some operators which you have not yet seen in the lectures.

Precedence	Operator	Type	Associativity
15	() [] .	Parentheses Array subscript Member selection	Left to Right
14	++ --	Unary post-increment Unary post-decrement	Right to left
13	++ -- + - ! ~ (type)	Unary pre-increment Unary pre-decrement Unary plus Unary minus Unary logical negation Unary bitwise complement Unary type cast	Right to left
12	* / %	Multiplication Division Modulus	Left to right
11	+ -	Addition Subtraction	Left to right
10	<< >> >>>	Bitwise left shift Bitwise right shift with sign extension Bitwise right shift with zero extension	Left to right
9	< <= > >= instanceof	Relational less than Relational less than or equal Relational greater than Relational greater than or equal Type comparison (objects only)	Left to right
8	== !=	Relational is equal to Relational is not equal to	Left to right
7	&	Bitwise AND	Left to right
6	^	Bitwise exclusive OR	Left to right
5		Bitwise inclusive OR	Left to right
4	&&	Logical AND	Left to right
3		Logical OR	Left to right
2	? :	Ternary conditional	Right to left
1	= += -= *= /= % =	Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment	Right to left