**DEPARTMENT OF PHYSICS AND COMPUTER SCIENCE**

**SCHOOL OF TECHNOLOGY**

**NJALA UNIVERSITY**

**NJALA CAMPUS**



**PROGRAMMING IN JAVA**

**FOR**

**SECOND YEAR HD STUDENTS**

**BECAREFUL WHAT YOU WISH FOR**

**It is said you must crawl before you can walk, and walk before you can ride a bicycle.**

**INTRODUCTION TO JAVA**

The Java™ programming language, developed at Sun Microsystems under the guidance of Net luminaries James Gosling and Bill Joy, is designed to be a machine-independent programming language that is both safe enough to traverse networks and powerful enough to replace native executable code

Java is a portable language and programs written in Java can run on many different computers. For programming in general, *portability is an elusive* *goal.* Java is a fully object-oriented language

**Overview of Java**

Java originated at Sun Microsystems as a project for intelligent consumer-electronic devices.

When the World Wide Web exploded in popularity in 1993, Sun people saw the immediate potential

of using Java to create Web pages with so-called dynamic content.

• Java is now used to create Web pages with dynamic and interactive content, to develop large-scale

enterprise applications, to enhance the functionality of Web servers, to provide applications for

consumer devices and so on.

• Java programs consist of pieces called classes (You can think of a class as a code segment. But you have to tell Java where code segments start and end) Classes consist of pieces called methods that perform

tasks and return information when they complete their tasks.

• Most Java programmers use rich collections of existing classes in Java class libraries.

Java has become the language of choice for implementing Internet-based and Intranet based

applications and software for devices that communicate over a network. Do not be

surprised when your new stereo and other devices in your home will be networked together

by Java technology! Also, do not be surprised when your wireless devices, like cell phones,

pagers and personal digital assistants (PDAs) communicate over the so-called Wireless

Internet via the kind of Java-based networking protocols

**Importance of Java**

1. **Mature and Keeps Evolving**

Java is currently one of the mature and stable programming languages. But Oracle Corporation frequently updates the programming language with the support of a vibrant community. Each new version of Java comes with several new features and enhanced performance. For instance, the latest version of Java supports both functional and concurrent programming. Also, it comes with several new features like a new date/time API and improved Nashorn JVM JavaScript engine, while supporting parallel processing and lambda expressions. These new features make Java compete with other JVM-based languages more efficiently.

**2) Platform Independent**

Nowadays programmers have to write applications by targeting many devices and platforms. So they look for a programming language that enables them to write the application cod once and deploys the application code across multiple platforms without putting extra effort. The programmers can simply compile the Java code once into bytecode, and deploy the bytecode across many platforms without compiling the code again. The bytecode makes it easier for programmers to deploy the application code on any platform that supports Java. Also, they can easily port the application from one platform to another without compiling the code repeatedly.

**3) Supports Common Programming Paradigms**

The syntax rules of Java are based on the syntax of C and C++. Hence, the beginners find it easier to learn and use Java within a shorter amount of time. At the same time, Java is a concurrent, class-based and object-oriented programming language. As Java supports common object-oriented programming (OOP) concepts like inheritance, polymorphism, abstraction and encapsulation, it becomes easier for programmers to make the applications modular, extensible and scalable. Also, the developers can take advantage of certain Java libraries to implement the object-oriented design principles more efficiently.

**4) Google Recommends for Android App Development**

While building mobile apps, no developers can ignore Android – the mobile operating system with the largest installation base. The programmers have the option to write Android apps in C, C++ or Java. But Google recommends mobile app developers to write Android apps only in Java. The developers can further boost the application’s performance and compatibility of the Android apps easily by writing it in Java. The developers also have the option to use a variety of tools and libraries to write robust Android apps in Java within a shorter amount of time.

**5) Rich Set of APIs**

Java beats other programming languages in the category of rich application programming interfaces (APIs). The programmers have the option to use a variety of Java APIs to accomplish common development tasks without writing additional code. Some of these APIs are shared by large enterprises, while others are uploaded by members of the community. Based on their requirements, the developers have option to use APIs for database connection, input/output, networking, utilities, security and XML parsing. They can further combine these APIs with various open source Java libraries to boost the application’s functionality and performance without putting extra time and effort.

**6) Loads of Frameworks, Libraries, IDEs and Development Tools**

The developers also have the option to avail a variety of frameworks, libraries, IDEs and development tools for Java. They can take advantage of robust IDEs like Netbeans and Eclipse to write readable and quality Java code. They can even avail the advanced debugging capabilities and code completion features provided by the IDEs to make the Java core maintainable. Likewise, the developers also have the option to use a wide variety of open source libraries shared by large companies and individual developers. The programmers can further make modern web applications rapidly by using robust Java web frameworks like Spring, Play, Spark and Dropwizard. These frameworks, libraries, IDEs and development tools make it easier for programmers to write a variety of applications in Java without putting extra time and effort.

**7) Robust Security Features**

The security features provided by Java make it easier for programmers to develop large and enterprise applications. The Java Virtual Machine (JVM) evaluates the intermediate bytecode to prevent the application from performing unsafe operations. The developers can further avail the advanced security management features of Java to prevent the untrusted bytecode from accessing specific features and APIs by running them in a sandboxed environment. At the same time, the developers can also avail the robust security APIs provided by Java platform, along with performing user authentication and using secure communication protocols,

**8) Simplify Development of Real-Time Software**

The popularity and adaption rate of real-time software has been increasing consistently. Unlike conventional software applications, the real-time applications are required to deliver information or results within the shortest amount of time. At present, real-time software applications are being used by mobile devices, automobiles, medical devices and factories. Oracle has included several features in Java SE to facilitate the development of real-time software. The APIs provided by Java SE make it easier for programmers to implement the real-time applications smoothly and effectively. Many programmers will use Java SE in future for developing real-time software that is compatible with various devices and platforms, and can be integrated seamlessly with third-party components and applications.

**9) Facilitates Embedded Computing**

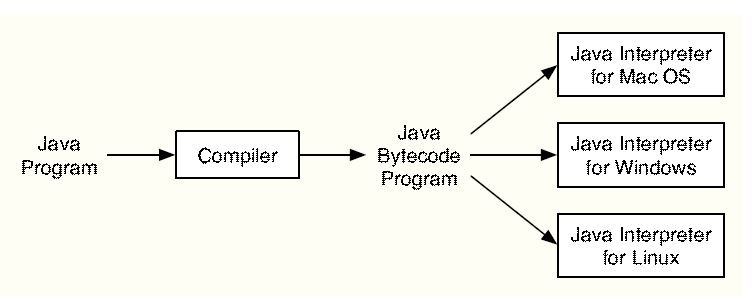
Java as originally designed for embedded programming. But many developers prefer C and Python over Java while developing embedded applications for systems with constrained memory. They now have the option to take advantage of the Java ME Platform to standard runtime environment for building a variety of embedded application. In addition to including built-in network protocols, robust security, and a flexible user interface, the micro edition of the Java Platform also supports a wide variety of mobile and embedded devices. The applications developed using Java ME can run on smartphones, tablets, sensors, gateways, printers and personal digital assistants (PDAs). These features will encourage more developers to use Java ME in future for embedded computing.

**10) Supports Internet of Things**

Java is one of the programming languages that currently support Internet of Things. The Project Jigsaw aims to make the programming language run on a wider variety of small and portable devices. However, the project still aims to maintain Java’s scalability, networking, maintainability, security, performance and other features while making it run on smaller and newer devices. That is why; Project Jigsaw has already proposed a design and standard module system for the Java SE Platform to accomplish its objectives. Once the proposal is implemented, programmers will use Java for writing a wide variety of Internet of Things applications.

**The Java Virtual Machine**

Java is platform independent. This means that it will run on just about any operating system. So whether your computer runs Windows, Linux, Mac OS, it's all the same to Java! The reason it can run on any operating system is because of the Java Virtual Machine. **The Virtual Machine is a programme that processes all your code correctly.** So you need to install this programme (Virtual Machine) before you can run any Java code Java is owned by a company called Sun Microsystems, so you need to head over to Sun's website to get the Java Virtual Machine, also known as the Java Runtime Environment (JRE).



**The Java Software Development Kit**

In order to write code and test it out, you need something called a Software Development kit.

**How to execute java program**

Java programs normally go through five phases to be executed

* Edit
* Compile
* Load
* verify
* execute.

**Phase 1** consists of editing a file. This is accomplished with an *editor program* (normally

known as an *editor*). The programmer types a Java program, using the editor, and

makes corrections, if necessary. When the programmer specifies that the file in the editor

should be saved, the program is stored on a secondary storage device, such as a disk. Java

program file names end with the ***.java*** *extension*.

**Phase 2** the programmer gives the command ***javac*** to *compile* the program. The Java compiler translates the Java program into *byte-codes*—the language understood by the Java interpreter.

To compile a program called **Welcome.java**, type **javac Welcome.java** at the command window of your system If the program compiles correctly, the compiler produces a file called **Welcome.class**. This is the file containing the bytecodes that will be interpreted during the execution phase.

**Phase 3** is called *loading.* The program must first be placed in memory before it can be

executed. This is done by the *class loader,* which takes the **.class** file (or files) containing

the bytecodes and transfers it to memory

**Phase 4**. Verification

This ensures that the bytecodes for classes that are loaded are valid and that they do not violate Java’s security restrictions. Java enforces strong security, because Java programs should

not be able to cause damage to your files and your system (as computer viruses might).

Note that bytecode verification also occurs in applications that download classes from a

network.

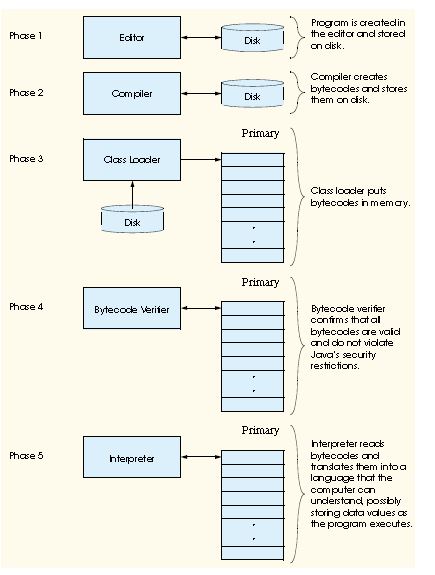
**Phase 5**. Execution

The computer, under the control of its CPU, interprets the program

one bytecode at a time, thus performing the actions specified by the program.

Programs might not work on the first try. Each of the preceding phases can fail because

of various errors.



• Java program file names end with the **.java** extension.

• The Java compiler (**javac**) translates a Java program into bytecodes—the language understood

by the Java interpreter. If a program compiles correctly, the compiler produces a file with the

**.class** extension. This is the file containing the bytecodes that are interpreted during the execution phase.

• A Java program must first be placed in memory before it can execute. This is done by the class

loader, which takes the **.class** file (or files) containing the bytecodes and transfers it to memory.

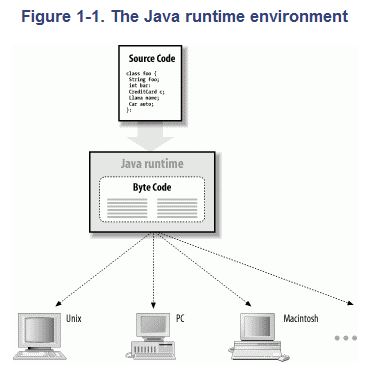
The **.class** file can be loaded from a disk on your system or over a network.

**The Process**

This can be simply explained as you write the actual code for your programs in a text editor. The code is called source code, and is saved with the file extension .java. A programme called Javac is then used to turn the source code into Java Byte Code. This is known as compiling. After Javac has finished compiling the Java Byte Code, it creates a new file with the extension .class. (At least, it does if no errors are detected.) Once the class file has been created, it can be run on the Java Virtual Machine.

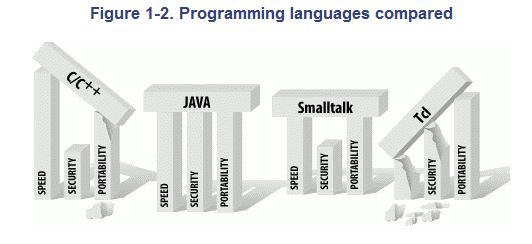
So:

* Create source code with the extension .java
* Use Javac to create (compile) a file ending in .class
* Run the compiled class



**Java Compared with Other Languages**

Java is a new language, but it draws on many years of programming experience with other languages in its choice of features. Much can be said in comparing and contrasting Java with other languages. There are at least three pillars necessary to support a universal language for network programming today: portability, speed, and security. Figure 1-2 shows how Java compares to other languages.



### Java as a General Application Language

### Java was introduced to the world through the web browser and the Java applet API. However, Java is more than just a tool for building multimedia applications. Java is a powerful, general-purpose programming language that just happens to be safe and architecture-independent. Standalone Java applications are not subject to the restrictions placed on applets; they can perform the same jobs as do programs written in languages such as C and C++.

Any software that implements the Java runtime system can run Java applications. Applications written in Java can be large or small, standalone or component-like, as in other languages. Java applets are different from other Java applications only in that they expect to be managed by a larger application.

# **The Structure of Java Code**

public class **FirstProject {**

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

**}**

**}**

You can think of a class as a code segment. But you have to tell Java where code segments start and end. You do this with curly brackets. The start of a code segment is done with a left curly bracket { and is ended with a right curly bracket }. Anything inside of the left and right curly brackets belong to that code segment.

What's inside of the left and right curly brackets for the class is another code segment. This one:

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

**}**

The word "main" is the important part here. Whenever a Java programme starts, it looks for a method called main. (A method is just a chunk of code. You'll learn more about these later.) It then executes any code within the curly brackets for main. You'll get error messages if you don't have a main method in your Java programmes. But as its name suggest, it is the main entry point for your programmes. The blue parts before the word "main" can be ignored for now

If you're curious, however, public means that the method can be seen outside of this class; static means that you don't have to create a new object; and void means it doesn't return a value - it just gets on with it. The parts between the round brackets of main are something called command line arguments. Don't worry if you don't understand any of that, though.)

The important point to remember is that we have a class called FirstProject. This class contains a method called main. The two have their own sets of curly brackets. But the main chunk of code belongs to the class FirstProject.

**Editing and Compiling in Java**

**Editing**

Use any text editor of your choosing, such as windows Notepad, JNotePad, or Hesky Data Pad.

When writing code in Java, use 2 spaces instead of a tab to indent, because the number of spaces that a tab represents varies with the editor or viewer of your code. For consistency and readability, avoid tabs in source code



When saving JAVA files from windows Notepad, put the filename in quotes to prevent Windows from adding **.txt** to the filename.



Using Java Command Line Compiling Choose a folder in which to work (in this example, a folder named “java” on a drive mapped to the “C:”

drive). Enter these four commands to activate the compiler (assuming that **javac.exe** is located in

C:\Program Files\Java\jdk1.6.0\_10\bin):

Type cd\java in the command prompt

**Compiling**

***To compile***, use a command like the following to create a CLASS file:

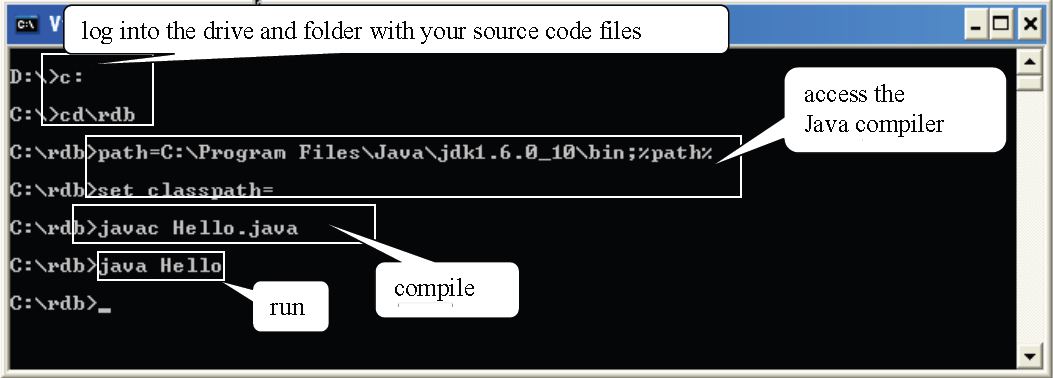
**javac Hello.java**

**Running**

After successfully compiling the JAVA file, you can run it as many time as you want.

***To run***, use a command like the following:

**java Hello**



In the above example, the file Hello.class was created on the “rdb” folder of the c: drive as a result of compiling. It’s run in the last command, without reference to the .class part of the filename Java requires that the filename extension be left out.

# **Java Comments**

When you create a New Project in NetBeans, you'll notice that some text is greyed out, with lots of slashes and asterisks:



The greyed-out areas are comments. When the programme runs, comments are ignored. So you can type whatever you want inside of your comments. But it's usual to have comments that explain what is you're trying to do. You can have a single line comment by typing two slashes, followed by your comment:

**//**This is a single line comment

If you want to have more than one line, you can either do this:

**//**This is a comment spreading **//**over two lines or more Or you can do this:

**/\***This is a comment spreading  
over two lines or more **\*/**

In the comment above, note how it starts with /\*. To end the comment, we have \*/ instead.

There's also something called a Javadoc comment. You can see two of these in the coding image on the previous page. A Javadoc comment starts with a single forward slash and two asterisks (/\*\*) and ends with an asterisk and one slash ( \*/ ). Each line of the comment starts with one asterisk:

**/\*\*  
\***This is a Javadoc comment  **\*/**

Javadoc comments are used to document code. The documented code can then be turned into an HTML page that will be helpful to others. You can see what these look like by clicking Run from the menu at the top of NetBeans. From the Run menu, select Generate Javadoc. There's not much to see, however, as you haven't written any code yet!

**Java Package**

A **Package** can be defined as a grouping of related types (classes, interfaces, enumerations and annotations ) providing access protection and namespace management.

**The six foundation Java packages are:**

* java.lang: Contains classes for primitive types, strings, math functions, threads, and

Exception

* java.util: Contains classes such as vectors, hash tables, date etc.
* java.io: Stream classes for I/O
* java.awt: Classes for implementing GUI – windows, buttons, menus etc.
* java.net: Classes for networking
* java.applet: Classes for creating and implementing applets

**Basic Aspects of Programming**

There are two basic aspects of programming:

* Data: to work with data, you need to understand variables and types;
* Instructions: to work with instructions, you need to understand control structures and subroutines.

**Java Variables and Primitive Type**

A variable is just a memory location (or several locations treated as a unit) that has been given a name so that it can be easily referred to and used in a program.

Programme work by manipulating data placed in memory. The data can be numbers, text, objects, pointers to other memory areas, and more besides. The data is given a name, so that it can be re-called whenever it is need. The name, and its value, is known as a Variable. We'll start with number values.

To store a number in java, you have lots of options. Whole numbers such as 8, 10, 12, etc., are stored using the int variable. (The int stands for integer.) Floating point numbers like 8.4, 10.5, 12.8, etc., are stored using the double variable. You do the storing with an equals sign ( = ). Let's look at some examples

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

int **first\_number;**

**System.out.println("**My First Project**");**

**}**

So to tell Java that you want to store a whole number, you first type the word int, followed by a space. You then need to come up with a name for your integer variable. You can call them almost anything you like, **but there are a few rules:**

* Variable names can't start with a number. So first\_number is OK, but not 1st\_number. You can have numbers elsewhere in the variable name, just not at the start.
* Variable names can't be the same as Java keywords. There are quite a lot of these, and they will turn blue in NetBeans, like **int** above.
* You can't have spaces in your variable names. The variable declaration **int first number** will get you an error. We've used the underscore character, but it's common practise to have the first word start with a lowercase letter and the second or subsequent words in uppercase: firstNumber, myFirstNumber
* Variable names are case sensitive. So firstNumber and FirstNumber are different variable names.

**Types**

A variable in Java is designed to hold only one particular type of data; it can legally hold that type of data and no other. The compiler will consider it to be a syntax error if you try to violate this rule. We say that Java is a strongly typed language because it enforces this rule.

There are eight so-called primitive types built into Java. The primitive types are named: byte, short, int, long, float, double, char, and boolean. The first four types hold integers (whole numbers such as 17, -38477, and 0). The four integer types are distinguished by the ranges of integers they can hold. The float and double types hold real numbers (such as 3.6 and -145.99). Again, the two real types are distinguished by their range and accuracy. A variable of type char holds a single character from the Unicode character set. And a variable of type boolean holds one of the two logical values true or false.

**Data Types**

|  |  |
| --- | --- |
| The following is a list of Java’s primitive data types: Data Type | Description |
| int | Integer – 32bit ranging from -2,147,483,648 to 2,147,483,648 |
| byte | 8-bit integer ranging from -128 to 127 |
| short | 16-bit integer ranging from -32,768 to 32,768 |
| long | 64-bit integer from -9,223,372,036,854,775,808 to -9,223,372,036,854,775,808 |
| float | Single-precision floating point, 32-bit |
| double | Double-precision floating point, 64-bit |
| char | Character , 16-bit unsigned ranging from 0 to 65,536 (Unicode) |
| Boolean | Can be true or false only |

The ‘String’ type has not been left out by mistake. It is not a primitive data type, but strings (a sequence of characters) in Java are treated as Objects.

To store something in the variable called **first\_number**, you type an equals sign and then the value you want to store:

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

int **first\_number;**

first\_number = 10;

**System.out.println("**My First Project**");**

**}**

So this tells java that we want to store a value of 10 in the integer variable that we've called first\_number. If you prefer, you can do all this on one line:

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

int first\_number = 10;

**System.out.println("**My First Project**");**

**}**

To see all this in action, change the println method slightly to this:

System.out.println( **"First number = " + first\_number** );

What we now have between the round brackets of println is some direct text enclosed in double quotes:

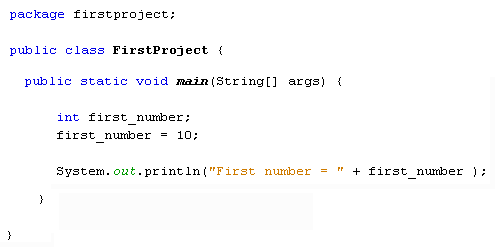
**("First number = "**

We also have a plus sign, followed by the name of our variable:

**+ first\_number );**

The plus sign means "join together". So we're joining together the direct text and the name of our variable. The joining together is known as concatenation.

Your coding window should now look like this (note how each line of code ends with a semicolon):



Let's try some simple addition. Add two more **int** variables to your code, one to store a second number, and one to store an answer:

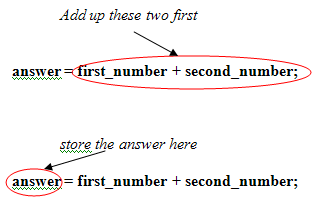
int **first\_number, second\_number, answer;**

Notice how we have three variable names on the same line. You can do this in Java, if the variables are of the same type (the **int** type, for us). Each variable name is then separated by a comma.

We can then store something in the new variables:

**first\_number = 10;  
second\_number = 20;  
answer = first\_number + second\_number;**

For the answer variable, we want to add the first number to the second number. Addition is done with the plus ( + ) symbol. Java will then add up the values in first\_number and second\_number. When it's finished it will store the total in the variable on the left of the equals sign. So instead of assigning 10 or 20 to the variable name, it will add up and then do the assigning. In case that's not clear, here's what happens:



The above is equivalent to this:

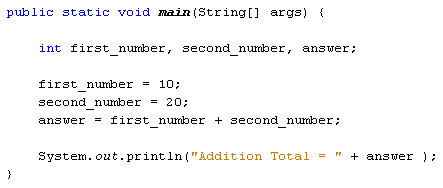
**answer = 10 + 20;**

But Java already knows what is inside of the two variables, first\_number and second\_number, so you can just use the names.

Now change your println method slightly to this:

**System.out.println("Addition Total = " + answer );**

Again, we are combining direct text surrounded by double quotes with a variable name. Your coding window should look like this:



So our programme has done the following:

* Stored one number
* Stored a second number
* Added these two numbers together
* Stored the result of the addition in a third variable
* Printed out the result

You can also use numbers directly. Change the answer line to this:

**answer = first\_number + second\_number + 12;**

Run your programme again. What printed out? Was it what you expected?

You can store quite large numbers in the **int** variable type. The maximum value is 2147483647. If you want a minus number the lowest value you can have is -2147483648. If you want larger or smaller numbers you can use another number variable type: **double**. You'll meet them in the next part of the course.

# **The Double Variable**

The double variable can hold very large (or small) numbers. The maximum and minimum values are 17 followed by 307 zeros.

The double variable is also used to hold floating point values. A floating point value is one like 8.7, 12.5, 10.1. In other words, it has a "point something" at the end. If you try to store a floating point value in an int variable, NetBeans will underline the faulty code. If you try to run the programme, the compiler will throw up an error message.

Let's get some practise using doubles.

Change the **int** from your previous code to double. So change this:

int **first\_number, second\_number, answer;**

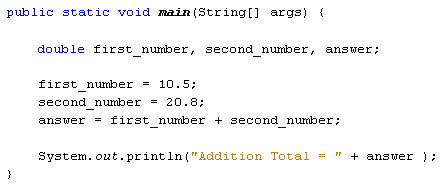
to this:

double **first\_number, second\_number, answer;**

Now change the values being stored:

**first\_number = 10.5;  
second\_number = 20.8;**

Leave the rest of the programme as it is. Your coding window should look like this:



Try changing the values stored in first\_number and second\_number. Use any values you like. Run your programme and see what happens.

# **Short and Float Variables**

Two more variable types you can use are **short** and **float**. The **short** variable type is used to store smaller number, and its range is between minus 32,768 and plus 32,767. Instead of using **int** in our code on the previous pages, we could have used short instead. You should only use short if you're sure that the values that you want to hold don't go above 32, 767 or below -32,768.

The double value we used can store really big numbers of the floating point variety. Instead of using double, **float** can be used. When storing a value in a float variable, you need the letter "f" at the end. Like this:

float **first\_number, second\_number, answer;**

**first\_number = 10.5f;  
second\_number = 20.8f;**

So the letter "f" goes after the number but before the semicolon at the end of the line. To see the difference between float and double, see below.

**Simple Arithmetic**

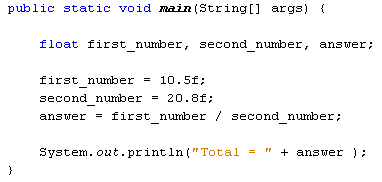
With the variables you've been using, you can use the following symbols to do calculations:

+ (the plus sign)  
- (the minus sign)  
\* (the multiplication sign is the asterisk sign)  
/ (the divide sign is the forward slash)

Try this exercise:

Delete the plus symbol that is used to add first\_number and second\_number. Replace it with the symbols above, first the minus sign, then the multiplication sign, and then the divide. The answer to the final one, the divide, should give you a really big number.

The number you should get for divide is 0.5048076923076923. This is because you used the double variable type. However, change the double to float. Then add the letter "f" to the end of the numbers. So your code should look like this:



When you run the above code, the answer is now 0.5048077. Java has taken the first 6 numbers after the point and then rounded up the rest. So the double variable type can hold more numbers than float. (Double is a 64 bit number and float is only 32 bit.)

**Operator Precedence**

You can, of course, calculate using more than two numbers in Java. But you need to take care of what exactly is being calculated. Take the following as an example:

**first\_number = 100;  
second\_number = 75;  
third\_number = 25;**

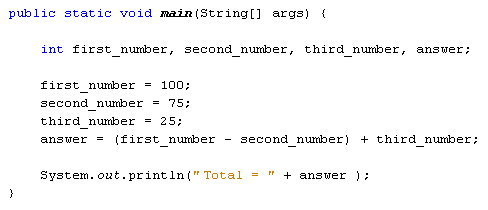
**answer = first\_number - second\_number + third\_number;**

If you did the calculation left to right it would be 100 - 75, which is 25. Then add the third number, which is 25. The total would be 50. However, what if you didn't mean that? What if you wanted to add the second and third numbers together, and then deduct the total from the first number? So 75 + 25, which is 100. Then deduct that from the first number, which is 100. The total would now be 0.

To ensure that Java is doing what you want, you can use round brackets. So the first calculation would be:

**answer = (first\_number - second\_number) + third\_number;**

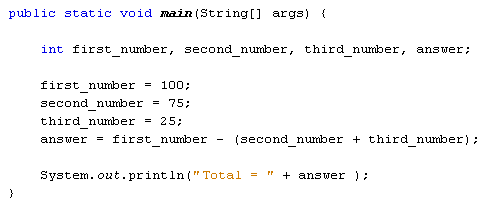
Here's the coding window so that you can try it out:



The second calculation is this:

**answer = first\_number - (second\_number + third\_number);**

And here's the code window:



Now let's try some multiplication and addition.

Change your math symbols (called Operators) to plus and multiply:

**answer = first\_number + second\_number \* third\_number;**

Delete all your round brackets, and then run your programme.

With no brackets, you'd think Java would calculate from left to right. So you'd think it would add the first number to the second number to get 175. Then you'd think it would multiply by the third number, which is 25. So the answer would be 4375. Run the programme, though. The answer that you actually get is 1975! So what's going on?

The reason Java got the "wrong" answer was because of Operator Precedence. Java treats some mathematical symbols as more important than others. It sees multiplication as having a priority over addition, so it does this first. It then does the addition. So Java is doing this:

**answer = first\_number + (second\_number \* third\_number);**

With the round brackets in place, you can see that second number is being multiplied by third number. The total is then added to the first number. So 75 multiplied by 25 is 1875. Add 100 and you get 1975.

If you want it the other way round, don't forget to "tell" Java by using round brackets:

**answer = (first\_number + second\_number) \* third\_number;**

Division is similar to multiplication: Java does the dividing first, then the addition or subtraction. Change your answer line to the following:

**answer = first\_number + second\_number / third\_number;**

The answer you get is 103. Now add some round brackets:

**answer = (first\_number + second\_number) / third\_number;**

The answer this time will be 7. So without the round brackets, Java does the dividing first, and then adds 100 to the total - it doesn't work from left to right.

**Here's a list on Operator Precedence**

* Multiply and Divide - Treated equally, but have priority over Addition and Subtraction
* Add and Subtract - Treated equally but have a lower priority than multiplication and division

So if you think Java is giving you the wrong answer, remember that Operator Precedence is important, and add some round brackets. (BODMAS)

In the next part, we'll take a look at how to store text values using Java.

# **String Variables**

As well as storing number values, variables can hold text. You can store just one character, or lots of characters. To store just one character, the char variable is used. Usually, though, you'll want to store more than one character. To do so, you need the string variable type.

To set up a string variable, you type the word **String** followed by a name for your variable. Note that there's an uppercase "S" for String. Again, a semicolon ends the line:

String **first\_name**;

Assign a value to your new string variable by typing an equals sign. After the equals sign the text you want to store goes between two sets of double quotes:

**first\_name = "Francis";**

If you prefer, you can have all that on one line:

String **first\_name = "Francis";**

Set up a second string variable to hold a surname/family name:

String **family\_name = "Manna";**

To print both names, add the following println( ):

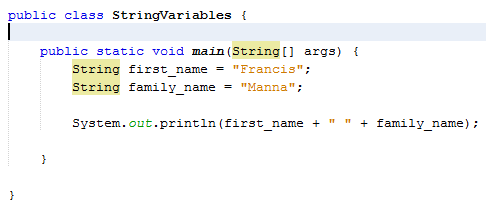
**System.out.println(** first\_name + " " + family\_name **);**

In between the round brackets of **println**, we have this:

**first\_name + " " + family\_name**

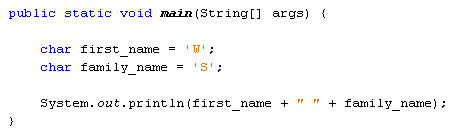
We're saying print out whatever is in the variable called **first\_name**. We then have a plus symbol, followed by a space. The space is enclosed in double quotes. This is so that Java will recognise that we want to print out a space character. After the space, we have another plus symbol, followed by the **family\_name** variable.

Although this looks a bit messy, we're only printing out a first name, a space, then the family name. Your code window should look like this:



**Char**

If you are storing just a single character, then the variable you need is **char** (lowercase "c"). To store the character, you use single quotes instead of double quotes. Here's our programme again, but this time with the char variable:



If you try to surround a char variable with double quotes, you will have errors. You can, however, have a String variable with just a single character. But you need double quotes. So this is OK:

**String first\_name = "W";**

But this is not:

**String first\_name = 'W';**

The second version has single quotes, while the first has double quotes.

There are lot more to strings, and you'll meet them again later. For now, let's move on and get some input from a user.

# **Accepting Input from a User**

One of the strengths of Java is the huge libraries of code available to you. This is code that has been written to do specific jobs. All you need to do is to reference which library you want to use, and then call a method into action. One really useful class that handles input from a user is called the **Scanner** class. The Scanner class can be found in the **java.util** library. To use the Scanner class, you need to reference it in your code. This is done with the keyword **import**.

The six input methods for the primitive numerical data types are listed below

|  |  |
| --- | --- |
| Table 5.1 Methods to input six numerical data types Method | Example |
| nextByte( ) | byte b = scanner.nextByte( ); |
| nextDouble( ) | double d = scanner.nextDouble( ); |
| nextFloat( ) | float f = scanner.nextFloat( ); |
| nextInt( ) | int i = scanner.nextInt( ); |
| nextLong( ) | long l = scanner.nextLong( ); |
| nextShort( ) | short s = scanner.nextShort( ); |

import **java.util.Scanner;**

The import statement needs to go just above the Class statement:

**import java.util.Scanner;**

**public class StringVariables {**

**}**

This tells java that you want to use a particular class in a particular library - the Scanner class, which is located in the java.util library.

The next thing you need to do is to create an object from the Scanner class. (A class is just a bunch of code. It doesn't do anything until you create a new object from it.)

To create a new Scanner object the code is this:

**Scanner user\_input = new Scanner(** System.in **);**

So instead of setting up an **int** variable or a **String** variable, we're setting up a **Scanner** variable. We've called ours **user\_input**. After an equals sign, we have the keyword **new**. This is used to create new objects from a class. The object we're creating is from the Scanner class. In between round brackets we have to tell java that this will be System Input (System.in).

To get the user input, you can call into action one of the many methods available to your new Scanner object. One of these methods is called **next**. This gets the next string of text that a user types on the keyboard:

**String first\_name;  
first\_name = user\_input.next( );**

So after our user\_input object we type a dot. You'll then see a popup list of available methods. Double click **next** and then type a semicolon to end the line. We can also print some text to prompt the user:

**String first\_name;  
System.out.print("Enter your first name: ");  
first\_name = user\_input.next( );**

Notice that we've used **print** rather than **println** like last time. The difference between the two is that println will move the cursor to a new line after the output, but print stays on the same line.

We'll add a prompt for a family name, as well:

**String family\_name;  
System.out.print("Enter your family name: ");  
family\_name = user\_input.next( );**

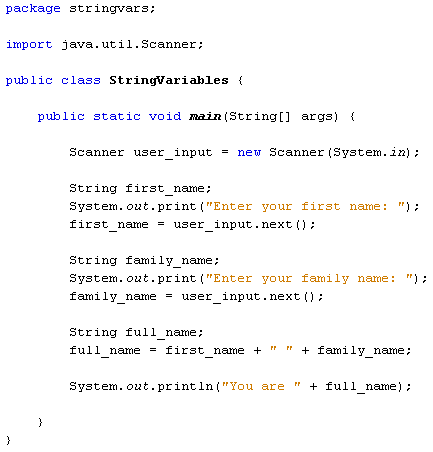
This is the same code, except that java will now store whatever the user types into our family\_name variable instead of our first\_name variable.

To print out the input, we can add the following:

**String full\_name;  
full\_name = first\_name + " " + family\_name;**  
  
**System.out.println("You are " + full\_name);**

We've set up another String variable, **full\_name**. We're storing whatever is in the two variables **first\_name** and **family\_name**. In between the two, we've added a space. The final line prints it all out in the Output window.

So adapt your code so that it matches that in the next image:



Run your programme until your Output window displays

Java is now pausing until you enter something on your keyboard. It won't progress until you hit the enter key. So left click after "Enter your first name:" and you'll see your cursor flashing away. Type a first name, and then hit the enter key on your keyboard.

After you hit the enter key, java will take whatever was typed and store it in the variable name to the left of the equals sign. For us, this was the variable called first\_name.

The programme then moves on to the next line of code:

we used the Scanner class to get input from a user. Whatever was typed was stored in variables. The result was then printed to the Output window.

# **Conditional Logic**

The programming you're doing now is sequential programming, meaning the code is executed from top to bottom. It's very linear, in that each and every line of code will be read, starting with the first line of code you write and ending at the last line.

But you don't always want your programmes to work like that. Often, you want code to be executed only if certain conditions are met. For example, you might want one message to display if a user is below the age of 18 and a different message if he or she is 18 or older. You want to control the flow of the programme for yourself. You can do this with conditional logic.

Conditional logic is mainly about the IF word: IF user is less than 18 then display this message; IF user is 18 or older then display that message. Fortunately, it's very easy to use conditional logic in Java. Let's start with IF Statements.

### IF Statements

Executing code when one thing happens rather than something else is so common in programming that that the IF Statement has been developed. The structure of the IF Statement in Java is this:

**if (** Statement **) {**

**}**

You start with the word IF (in lowercase) and a pair of round brackets. You then use a pair of curly brackets to section off a chunk of code. This chunk of code is code that you only want to execute IF your condition is met. The condition itself goes between round brackets:

**if (** user < 18 **) {**

**}**

This condition says "IF user is less than 18". But instead of saying "is less than" we use the shorthand notation of the left-pointing angle bracket ( < ). IF the user is less than 18 then we want something to happen, to display a message, for example:

i**f (** user < 18 **) {**

**//**DISPLAY MESSAGE

**}**

If the user is not less than 18 then the code between the curly brackets will be skipped, and the programme continues on its way, downwards towards the last line of code. Whatever you type between the curly brackets will only be executed IF the condition is met, and this condition goes between the round brackets.

Before we try this out, another shorthand notation is this symbol >. The right-pointing angle bracket means "greater than". Our IF Statement above can be amended slightly to check for users who are greater than 18:

**if (** user > 18 **) {**

**//**DISPLAY MESSAGE

**}**

The only thing new in this code is the > symbol. The condition now checks for users who are greater than 18.

But the condition doesn't check for people who are exactly 18, just those greater than 18. If you want to check for those who are 18 or over, you can say "greater than or equal to". The symbols for this are the greater than sign ( > ) followed by an equals sign ( = ):

**if (** user >= 18 **) {**

**//**DISPLAY MESSAGE

**}**

You can also check for "less than or equal to" in a similar way:

**if (** user <= 18 **) {**

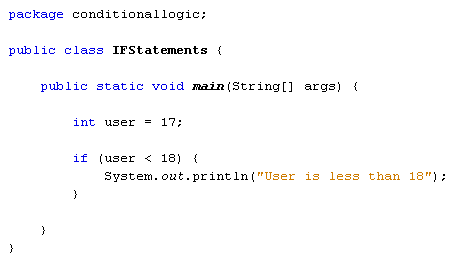
**//**DISPLAY MESSAGE

**}**

The above code contains a less than symbol ( < ) followed by the equals sign.

Let's try all this out in a simple programme.

Start a new project You can call your class names anything you like. Enter the following code (the Class is called **IFStatements**):



We've set up an integer variable, and assigned a value of 17 to it. The IF statement checks for "less than 18". So the message between the curly brackets should be printed out. Run your programme and check it out.

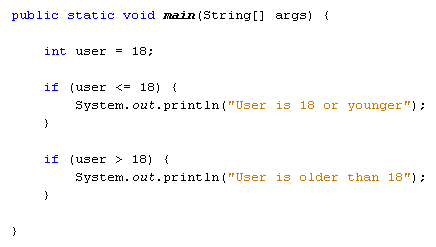
Now change the value for the user variable from 17 to 18. Run your programme again. You should see that nothing go printed on the screen if the program runs successfully

So the programme runs OK, with no error messages. It's just that nothing gets printed out. The reason is that the message code is between the curly brackets of the IF Statement. And the IF Statement is checking for values less than 18. IF the condition is not met, Java ignores the curly brackets altogether and moves on.

**Exercise**  
Replace your "less than" symbol with the "less than or equal to" symbols. Change your message to suit, something like "user is less than or equal to 18". Run your programme again. Do you see the message?

**Exercise**  
Change the user value to 20. Run your programme again. Do you still see the message?

You can have more than one IF Statement in your code. Try the following code:



This time, we have two IF Statements. The first tests for values less than or equal to 18. The second tests for values greater than 18. When the code is run with a value of 18 or less for the user variable, Changing the value of the user variable to 20

So only one of the IF Statements will Output a print line. And it all depends on what the value of the user variable is.

# **IF … ELSE**

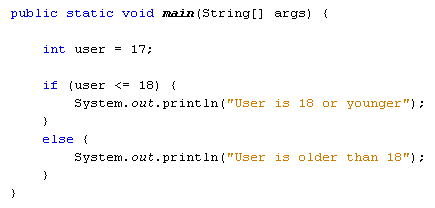
Instead of using two IF Statements, you can use an IF … ELSE Statement instead. Here's the structure of an IF … ELSE statement:

**if (** condition\_to\_test **) {**

**}  
else {**

**}**

The first line starts with if, followed by the condition you want to test for. This goes between two round brackets. Again, curly brackets are used to section off the different choices. The second choice goes after the word else and between its own curly brackets. Here's our code again that checks a user's age:



So there are only two choices here: either the user is 18 or younger, or the user is older than that. Adapt your code to match that in the image above and try it out. You should find that the first message prints out. Now change the value of the user variable to 20 and run the code again. The message between the ELSE curly brackets should display in the Output window.

### IF … ELSE IF

You can test for more than two choices. For example, what if we wanted to test for more age ranges, say 19 to 39, and 40 and over? For more than two choices, the IF … ELSE IF statement can be used. The structure of an IF … ELSE IF is this:

**if (** condition\_one **) {**

**}  
else if (** condition\_two **) {**

**}  
else {**

**}**

The new part is this:

**else if (** condition\_two **) {**

**}**

So the first IF tests for condition number one (18 or under, for example). Next comes else if, followed by a pair of round brackets. The second condition goes between these new round brackets. Anything not caught by the first two conditions will be caught be the final else. Again, code is sectioned off using curly brackets, with each if, else if, or else having its own pair of curly brackets. Miss one out and you'll get error messages.

Before trying out some new code, you'll need to learn some more conditional operators. The ones you have used so far are these:

**> Greater Than  
< Less Than  
>= Greater Than or Equal To  
<= Less Than or Equal To**

Here's four more you can use:

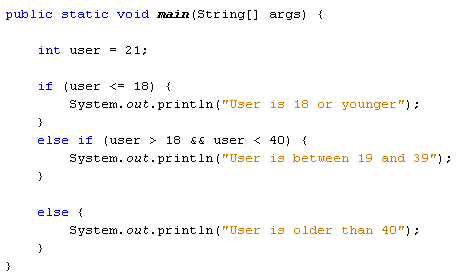
**&& AND  
|| OR  
== HAS A VALUE OF  
! NOT**

The first one is two ampersand symbols, and is used to test for more than one condition at the same time. We can use it to test for two age ranges:

**else if ( user > 18 && user < 40 )**

Here, we want to check if the user is older than 18 but younger than 40. Remember, we're trying to check what is inside of the user variable. The first condition is "Greater than 18" ( user > 18 ). The second condition is "Less than 40" ( user < 40). In between the two we have our AND operator ( &&). So the whole line says "else if user is greater than 18 AND user is less than 40."

We'll get to the other three conditional operators in a moment. But here's some new code to try out:



Run your programme and test it out. You should be able to guess what it will print out before running it. Because we have a value of 21 for the user variable the message between the curly brackets of else if will display in the Output window.

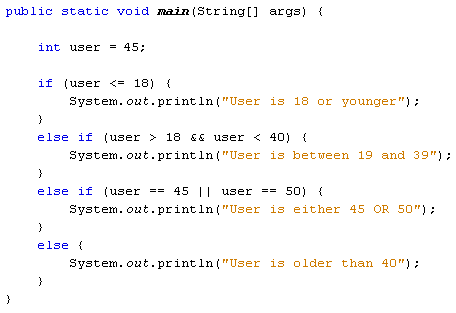
**Exercise**  
Change the value of the user variable from 21 to 45. The message for the else section of the code should now display.

You can add as many else if parts as you want. Suppose we wanted to check if the user was either 45 or 50. We can use two of the new conditional operators above. We can check if the user variable "has a value of 45" OR "has a value of 50":

**else if (user == 45 || user == 50)**

To test if the user variable has a value of something you use two equal signs, with no space between them. Java will test for that value and no other values. Because want to test for the user being 50 as well, we can have another condition in the same round brackets: user == 50. This just says "test if the user variable has a value of 50". In between the two conditions, we have the OR operator. This is two pipe characters, which is just to the left of the letter "z" on a UK keyboard. Again, there's no space between the two. The whole of the line above says "Else if the user has a value of 45 OR the user has a value of 50".

Here's our code with the new else if part:



Try it out for yourself. Change the value of the user variable to 45 and run your code. Then change it to 50 and run the code again. In both cases the new message should display.

The various conditional operators can be tricky to use. But you're just testing a variable for a particular condition. It's simply a question of picking the right conditional operator or operators for the job.

### Nested IF Statements

You can nest IF Statements. (This also applies to IF ... ELSE and IF ... ELSE IF statements.) Nesting an IF Statement just means putting one IF Statement inside of another. For example, suppose you want to find out if somebody is younger than 18, but older than 16. You want to display a different message for the over 16s. You start with the first IF Statement:

**if (** user < 19 **) {  
System.out.println( "18 or younger");  
}**

To check for over 16, you can place a second IF Statement inside of the one you already have. The format is the same:

**if (** user < 19 **) {  
if (** user > 16 && user < 19 **) {  
System.out.println( "You are 17 or 18");  
}  
}**

So the first IF Statement catches the user variable if it's less than 19. The second IF Statement narrows the user variable down even further, for ages over 16 and under 19. To print different messages, you can have an IF ... ELSE statement instead of the IF Statement above:

**if (** user < 19 **) {**

**if (** user > 16 && user < 19 **) {  
System.out.println( "You are 17 or 18");  
}  
else {  
System.out.println( "16 or younger");  
}**

**}**

Notice where all the curly brackets are in the code: get one wrong and your programme won't run.

Nested IF Statements can be tricky, but all you're trying to do is to narrow down the choices.

In the next section, you'll about the Boolean variable type.

# **Boolean Values**

A Boolean value is one with two choices: true or false, yes or no, 1 or 0. In Java, there is a variable type for Boolean values:

boolean **user = true;**

So instead of typing int or double or string, you just type boolean (with a lower case "b"). After the name of you variable, you can assign a value of either true or false. Notice that the assignment operator is a single equals sign ( = ). If you want to check if a variable "has a value of" something, you need two equal signs ( = =).

Try this simple code:

**boolean user = true;**

**if ( user == true) {  
System.out.println("it's true");  
}  
else {  
System.out.println("it's false");  
}**

So the first IF Statement checks if the user variable has a value of true. The else part checks if it is false. You don't need to say "else if ( user = = false)". After all, if something is not true then it's false. So you can just use else: there's only two choices with boolean values.

The only other conditional operator on our lists is the NOT operator. You can use this with boolean values. Have a look at the following code:

**boolean user = true;**

**if ( !user ) {  
System.out.println("it's flase");  
}  
else {  
System.out.println("it's true");  
}**

It's almost the same as the other boolean code, except for this line:

**if ( !user ) {**

This time, we have our NOT operator before the user variable. The NOT operator is a single exclamation mark ( ! ) and it goes before the variable you're tying to test. It's testing for negation, which means that it's testing for the opposite of what the value actually is. Because the user variable is set to true then !user will test for false values. If user was set to false then !user would test for true values. Think of it like this: if something is NOT true then what is it? Or if it's NOT false then what?

In the next part, we'll look at Java Switch Statements.

# **Switch Statements in Java**

Another way to control the flow of your programmes is with something called a switch statement. A switch statement gives you the option to test for a range of values for your variables. They can be used instead of long, complex **if … else if** statements. The structure of the switch statement is this:

**switch (** variable\_to\_test **) {  
case value:   
code\_here;  
break;  
case value:   
code\_here;  
break;  
default:  
values\_not\_caught\_above;**

**}**

So you start with the word **switch**, followed by a pair of round brackets. The variable you want to check goes between the round brackets of switch. You then have a pair of curly brackets. The other parts of the switch statement all go between the two curly brackets. For every value that you want to check, you need the word **case**. You then have the value you want to check for:

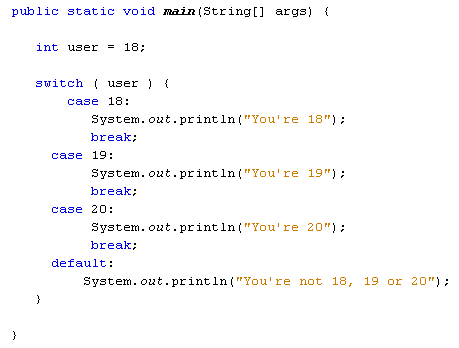
**case** value:

After case value comes a colon. You then put what you want to happen if the value matches. This is your code that you want executed. The keyword **break** is needed to break out of each case of the switch statement.

The default value at the end is optional. It can be included if there are other values that can be held in your variable but that you haven't checked for elsewhere in the switch statement.

If all of that is confusing, here's some code to try out. You can either start a new project for this, or just comment out the code you have.

But here's the code:



The first thing the code does is to set a value to test for. Again, we've set up an integer variable and called it **user**. We've set the value to 18. The switch statement will check the user variable and see what's in it. It will then go through each of the case statements in turn. When it finds one that matches, it will stop and execute the code for that case. It will then break out of the switch statement.

Try the programme out. Enter various values for the user variable and see what happens.

Sadly, you can't test for a range of values after case, just the one value. So you can't do this:

**case (user <= 18):**

But you can do this:

**case 1: case 2: case 3: case 4:**

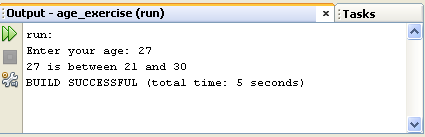
So the above line tests for a range of values, from 1 to 4. But you have to "spell out" each value. (Notice where all the case and colons are.)

To end this section on conditional logic, try these exercises.

**Exercise**  
Write a programme that accepts user input from the console. The programme should take a number and then test for the following age ranges: 0 to 10, 11 to 20, 21 to 30, 30 and over. Display a message in the Output window in the following format:

**user\_age + " is between 21 and 30"**

So if the user enters 27 as the age, the Output window should be this:



If the user is 30 or over, you can just display the following message:

**"Your are 30 or over"**

**Help for this exercise**

To get string values from the user, you did this:

**String age = user\_input.next( );**

But the **next**( ) method is used for strings. The age you are getting from the user has to be an integer, so you can't use next( ). There is, however, a similar method you can use: **nextInt**( ).

**Exercise**  
If you want to check if one String is the same as another, you can use a Method called **equals**.

**String user\_name = "Bill";**

**if (** user\_name.**equals**( "Bill" ) **) {  
//DO SOMETHING HERE  
}**

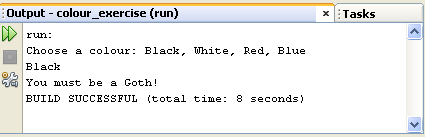
In the code above, we've set up a String variable and called it user\_name. We've then assigned a value of "Bill" to it. In between the round brackets of IF we have the variable name again, followed by a dot. After the dot comes the word "equals". In between another pair of round brackets you type the string you're trying to test for.

NOTE: When checking if one string is the same as another, they have to match exactly. So "Bill" is different from "bill". The first has an uppercase letter "B" and the second has a lowercase "b".

For this exercise, write a programme that asks a user to choose between four colours: black, white, red, or blue. Use IF … ELSE IF statements to display one of the following messages, depending on which colour was chosen:

**BLACK** "You must be a Goth!"  
**WHITE** "You are a very pure person"  
**RED** "You are fun and outgoing"  
**BLUE** "You're not a Chelsea fan, are you?"

When your programme ends, the Output window should look like something like this:



OK, let's move on and have a look at loops. We'll be upping the pace a bit in this next section, so hang on to your hats!

# **Loops in Java**

As we mentioned earlier, the programming you are doing now is sequential programming. This means that flow is downward, from top to bottom, with every line of code being executed, unless you tell Java otherwise.

You saw in the last section that one way to "tell" Java not to execute every line is by using IF Statement to section off areas of code.

Another way to interrupt the flow from top to bottom is by using loops. A programming loop is one that forces the programme to go back up again. If it is forced back up again you can execute lines of code repeatedly.

As an example, suppose you wanted to add up the numbers 1 to 10. You could do it quite easily in Java like this:

**int addition = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10;**

But you wouldn't really want to use that method if you needed to add up the numbers 1 to a 1000. Instead, you can use a loop to go over a line of code repeatedly until you've reached 1000. Then you can exit the loop and continue on your way.

### Java For Loops

We'll start with For Loops, one of the most common types of loops. The "For" part of "For Loop" seems to have lost its meaning. But you can think of it like this: "Loop **FOR** a set number of times." The structure of the For Loop is this:

**for ( start\_value; end\_value; increment\_number ) {**

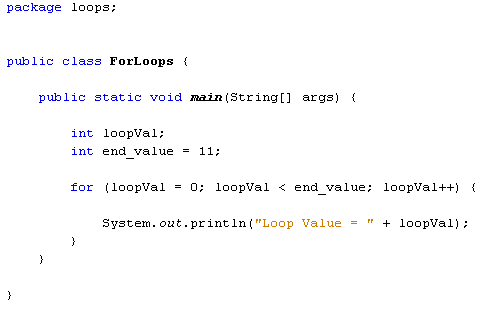
**//YOUR\_CODE\_HERE**

**}**

So after the word "for" (in lowercase) you have a pair of round brackets. Inside of the round brackets you need three things: the start value for the loop, the end value for the loop, and a way to get from one number to another. This is called the increment number, and is usually 1. But it doesn't have to be. You can go up in chunks of 10, if you want.

After the round brackets you need a pair of curly brackets. The curly brackets are used to section off the code that you want to execute repeatedly. An example might clear things up.

Start a new project for this. Call the Project and Class anything you like. (We've called our Project "loops", and the Class "ForLoops"). Now add the following code:



We start by setting up an integer variable, which we've called loopVal. The next line sets up another integer variable. This variable will be used for the end value of the loop, and is set to 11. What we're going to do is to loop round printing out the numbers from 0 to 10.

Inside the round brackets of the for loop, we have this:

**loopVal =0; loopVal < end\_value; loopVal++**

The first part tells Java at what number you want to start looping. Here, we're assigning a value of zero to the loopVal variable. This will be the first number in the loop. The second part uses some conditional logic:

**loopVal < end\_value**

This says "loopVal is less than end\_value". The for loop will then keep going round and round while the value inside the loopVal variable is less than the value in the variable called end\_value. As long as it's true that loopVal is less than end\_value, Java will keep looping over the code between the curly brackets.

The final part between the round brackets of the for loop is this:

**loopVal++**

What we're doing here is telling Java how to go from the starting value in loopVal to the next number in the sequence. We want to count from 0 to 10. The next number after 0 is 1. loopVal++ is a shorthand way of saying "add 1 to the value in the variable".

Instead of saying loopVal++ we could have wrote this:

**loopVal = loopVal + 1**

To the right of the equals sign we have loopVal + 1. Java will then add 1 to whatever is currently stored in the loopVal variable. Once it has added 1 to the value, it will store the result inside of the variable to the left of the equals sign. This is the loopVal variable again. The result is that 1 keeps getting added to loopVal. This is called incrementing the variable. It is so common that the shorthand notation variable++ was invented:

**int some\_number = 0;  
some\_number++;**

The value of some\_number will be 1 when the code above is executed. It is the short way of saying this:

**int some\_number = 0;  
some\_number = some\_number + 1;**

To recap then, our for loop is saying this:

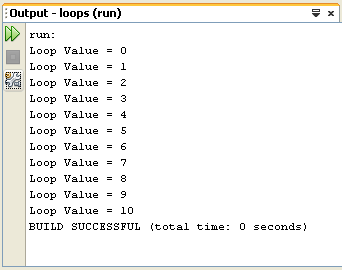
**Loop Start value**: 0  
**Keep Looping Whil**e: Start value is less than 11  
**How to advance to the end value**: Keep adding 1 to the start value

Inside of the curly brackets of the for loop we have this:

**System.out.println("Loop Value = " + loopVal);**

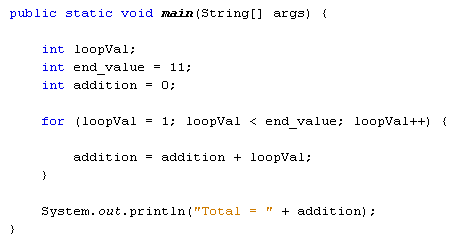
Whatever is currently inside of the loopVal variable will be printed out, along with some text.

Run your programme and you should see this in the Output window:



So we've trapped the programme in a loop, and forced it to go round and round. Each time round the loop, 1 gets added to the **loopVal** variable. The loop keeps going round and round while the value inside of **loopVal** is less than the value in **end\_value**. Whatever is inside of the loop's curly brackets is the code that will be executed over and over. And that is the whole point of the loop: to execute the curly bracket code over and over.

Here's some code that adds up the numbers 1 to 10. Try it out:



The answer you should get in the Output window is 55. The code itself is more or less the same as the previous for loop. We have the same two variables set up at the top, **loopVal** and **end\_value**. We also have a third integer variable, which we've called **addition**. This will hold the value of the 1 to 10 sum.

In between the round brackets of the for loop, it's almost the same as last time: we're looping while **loopVal** is less than **end\_value**; and we're adding 1 to the loopVal variable each time round the loop (loopVal++). The only difference is that the starting value is now 1 (loopVal=1).

In between the curly brackets, we only have one line of code:

**addition = addition + loopVal;**

This single line of code adds up the numbers 1 to 10. If you're confused as to how it works, start to the right of the equals sign:

**addition + loopVal;**

The first time round the loop, the **addition** variable is holding a value of 0. The variable **loopVal**, meanwhile, is holding a value of 1 (its starting value). Java will add 0 to 1. Then it will store the result to the variable on the left of the equals sign. Again, this is the addition variable. Whatever was previously being held in the addition variable (0) will be erased, and replaced with the new value (1).

The second time round the loop, the values in the two variables are these (the values are between round brackets):

**addition (1) + loopVal (2);**

1 + 2 is obviously 3. So this is the new value that will be stored to the left of the equals sign.

The third time round the loop, the new values are these:

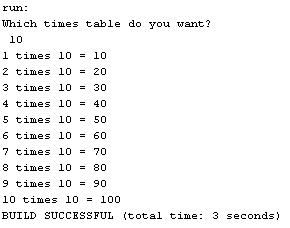
**addition (3) + loopVal (3);**

Java adds the 3 + 3 and stores 6 to the left of the equals sign. It keeps going round and round until the loop ends. The result is 55.

(Notice that our print line is outside of the for loop, after the final curly bracket of the loop.)

**Exercise**  
Change you code so that the loop adds up the numbers 1 to a 100. The answer you should get is 5050.

**Exercise**  
Write a times table programme. The programme should ask a user to input a number. This number is then used as the times table. So if the user enters 10, the 10 times table should be displayed. Your Output window should look something like this, when your programme is run.



**Help for this Exercise**

Your **for** loop only needs two lines of code between the curly brackets, and one of those is the print line. You only need a single line to calculate the answers for your times table.

You should already know how to get the number from the user. This can be used in your loop's curly brackets to work out the answer to your multiplication.

Your times table only needs to go from 1 to 10, like ours does in the image above.

**Exercise**  
Use a **for** loop to print out the odd numbers from 1 to 10. (For the easy way to do this exercise, think about the increment value of the loop, which is the third item between the round brackets.)

One of the hard ways to do the exercise above is by using an operator you haven't yet met - the modulus operator. Modulus is when you divide by a number and keep the remainder. So 10 Mod 3 is 1, because 10 divide by 3 is 3. The remainder is 1, and that's what you keep. The Modulus operator in Java is the percentage sign, rather confusingly. It's used like this:

**int remainder;  
int total = 10  
remainder = total %3**

So the number (or variable) you want to divide up goes first. You then type a percentage sign, followed by your divider number. The answer is the remainder.

In the exercise above, you could use 2 as the Mod number, and then use an IF Statement in your for loop to test what the remainder is. (Can you see why 2 should be the Mod number?)

In the next part, we'll take a look at Java **while** loops.

# **While Loops**

Another type of loop you can use in Java is called the **while** loop. While loops are a lot easier to understand than for loops. Here's what they look like:

**while (** condition **) {**

**}**

So you start with the word "while" in lowercase. The condition you want to test for goes between round brackets. A pair of curly brackets comes next, and the code you want to execute goes between the curly brackets. As an example, here's a while loop that prints out some text (Try the code out for yourself):

**int loopVal = 0;**

**while (** loopVal < 5**) {  
System.out.println("Printing Some Text");  
loopVal++;  
}**

The condition to test is between the round brackets. We want to keep looping while the variable called **loopVal** is less than 5. Inside of the curly brackets our code first prints out a line of text. Then we need to increment the **loopVal** variable. If we don't we'll have an infinite loop, as there is no way for **loopVal** to get beyond its initial value of 0.

Although we've used a counter (loopVal) to get to the end condition, while loops are best used when you don't really need a counting value, but rather just a checking value. For example, you can keep looping while a key on the keyboard is not pressed. This is common in games programme. The letter "X" can pressed to exit the while loop (called the game loop), and hence the game itself. Another example is looping round a text file while the end of the file has not been reached.

### Do ... While

Related to the **while** loop is the **do … while** loop. It looks like this:

**int loopVal = 0;**

**do {  
System.out.println("Printing Some Text");  
loopVal++;  
}  
while (** loopVal < 5 **);**

Again, Java will loop round and round until the end condition is met. This time, the "while" part is at the bottom. But the condition is the same - keep looping while the value inside of the variable called loopVal is less than 5. The difference between the two is the code between the curly brackets of do … while will get executed at least once. With the while loop, the condition could already be met. Java will then just bail out of your loop, and not even execute your curly bracket code. To test this out, try the while loop first. Change the value of your loopVal variable to 5, and then run the code. You should find that the text doesn't get printed. Now try the do loop with a value of 5 for loopVal. The text will print once and then Java will bail out of the loop.

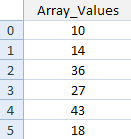
OK, we'll leave looping there. It is a subject you need to get grips with. But don't worry if you're not completely perfect with them - you'll learn as we go along. In the next section, we'll take a look at something called an array.

# **Java Arrays**

A programming concept you just have to get used to if you're to code effectively is the array. In this section, you'll learn what arrays are, and how to use them.

### What is an Array?

So far, you have been working with variables that hold only one value. The integer variables you have set up have held only one number, and the string variables just one long string of text. An array is a way to hold more than one value at a time. It's like a list of items. Think of an array as the columns in a spreadsheet. You can have a spreadsheet with only one column, or lots of columns. The data held in a single-list array might look like this:



Like a spreadsheet, arrays have a position number for each row. The positions in an array start at 0 and go up sequentially. Each position in the array can then hold a value. In the image above array position 0 is holding a value of 10, array position 1 is holding a value of 14, position 2 has a value of 36, and so on.

To set up an array of number like that in the image above, you have to tell Java what kind of data is going in to your array (integers, strings, boolean values, etc). You then need to say how many positions the array has. You set them up like this:

**int[ ] aryNums;**

The only difference between setting up a normal integer variable and an array is a pair of square brackets after the data type. The square brackets are enough to tell Java that you want to set up an array. The name of the array above is aryNums. Just like normal variables, you can call them almost anything you like (with the same exceptions we mentioned earlier).

But this just tells Java that you want to set up an integer array. It doesn't say how many positions the array should hold. To do that, you have to set up a new array object:

**aryNums = new int[**6**];**

You start with your array name, followed by the equals sign. After the equals sign, you need the Java keyword new, and then your data type again. After the data type come a pair of square brackets. In between the square brackets you need the size of the array. The size is how many positions the array should hold.

If you prefer, you can put all that on one line:

**int[ ] aryNums = new int[**6**];**

So we are telling Java to set up an array with 6 positions in it. After this line is executed, Java will assign default values for the array. Because we've set up an integer array, the default values for all 6 positions will be zero ( 0 ).

To assign values to the various positions in an array, you do it in the normal way:

**aryNums[0] = 10;**

Here, a value of 10 is being assigned to position 0 in the array called aryNums. Again, the square brackets are used to refer to each position. If you want to assign a value of 14 to array position 1, the code would be this:

**aryNums[1] = 14;**

And to assign a value of 36 to array position 2, it's this:

**aryNums[2] = 36;**

Don't forget, because arrays start at 0, the third position in an array has the index number 2.

If you know what values are going to be in the array, you can set them up like this instead:

**int[ ] aryNums = { 1, 2, 3, 4 };**

This method of setting up an array uses curly brackets after the equals sign. In between the curly brackets, you type out the values that the array will hold. The first value will then be position 0, the second value position 1, and so on. Note that you still need the square brackets after int, but not the new keyword, or the repetition of the data type and square brackets. But this is just for data types of int values, string, and char values. Otherwise, you need the new keyword. So you can do this:

**String[ ] aryStrings = {"Autumn", "Spring", "Summer", "Winter" };**

But not this:

**boolean[ ] aryBools = {false, true, false, true};**

To set up a boolean array you still need the new keyword:

**boolean[ ] aryBools = new boolean[ ] {false, true, false, true};**

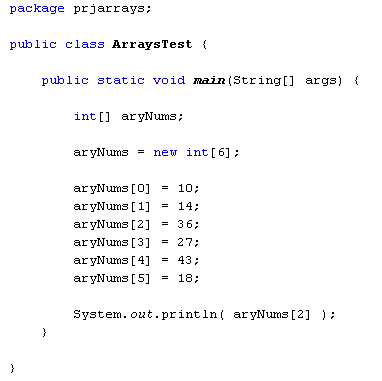
To get at the values held in your array, you type the name of the array followed by an array position in square brackets. Like this:

**System.out.println( aryNums[2] );**

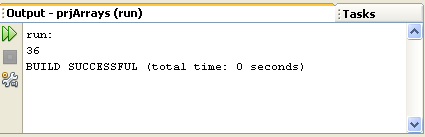
The above code will print out whatever value is held at array position 2 in the array called aryNums. But let's get some coding practice.

Start a new project and call it anything you like. Don't forget to change the name of the Class to something relevant.

Type the following code into your new Main method:



When you run the programme you should see this in the Output window:



Change the array position number in the print line from 2 to 5 and 18 should print out instead.

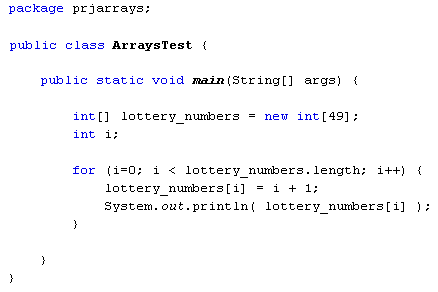
In the next part, we'll take a look at how to use arrays with loops.

# **Arrays and Loops**

Arrays come into their own with loops. You have seen in the previous section that to assign values to array positions, you did this:

**aryNums[0] = 10;**

But that's not terribly practical if you have a lot of numbers to assign to an array. As an example, imagine a lottery programme that has to assign the numbers 1 to 49 to positions in an array. Instead of typing a long list of array positions and values you can use a loop. Here's some code that does just that:



So we set up an array to hold 49 integer values. Then comes the loop code. Notice the end condition of the loop:

**i < lottery\_numbers.length**

**Length** is a property of array objects that you can use to get the size of the array (how many positions it has). So this loop will keep going round and round while the value in the variable i is less than the size of the array.

To assign values to each position in the array, we have this line:

**lottery\_numbers[i] = i + 1;**

Instead of a hard-code value between the square brackets of the array name, we have the variable called **i**. This increases by 1 each time round the loop, remember. Each array position can then be accessed just by using the loop value. The value that is being assigned to each position is i + 1. So again, it's just the incremented loop value, this time with 1 added to it. Because the loop value is starting at 0, this will give you the numbers 1 to 49.

The other line in the loop just prints out whatever value is in each array position.  
(If you wanted, you could then write code to jumble up the numbers in the array. Once you have jumbled up the values, you could then take the first 6 and use them as the lottery numbers. Write another chunk of code that compares a user's 6 numbers with the winning numbers and you have a lottery programme!)

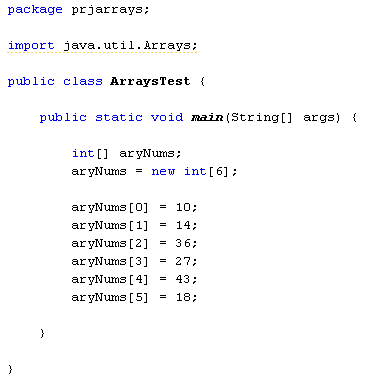
In the next part, you'll see how to sort your arrays.

# **Sorting Arrays**

Other inbuilt java methods allow you to sort your arrays. To use the sort method of arrays, you first need to reference a Java library called **Arrays**. You do this with the import statement. Try it with your aryNums programme. Add the following import statement:

**import java.util.Arrays;**

You code should look like ours below:

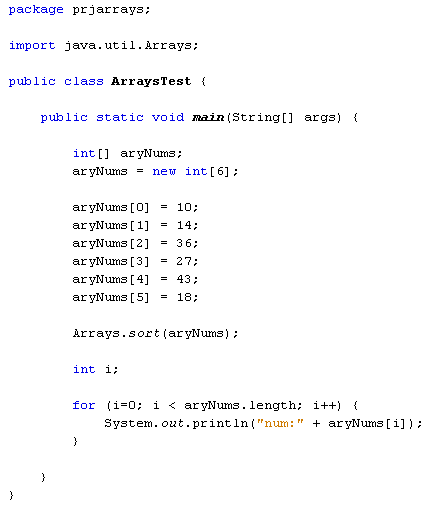


Now that you have imported the Arrays library, you can use the sort method. It's quite easy:

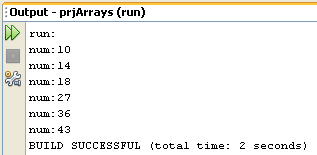
**Arrays.sort(** aryNums **);**

First you type the word "Arrays", then a dot. As soon as you type a dot, NetBeans will display a list of things you can do with arrays. Type the word "sort". In between a pair of round brackets, you then put the name of the array you want to sort. (Notice that you don't need any square brackets after the array name.)

And that's it - that's enough to sort the array! Here's some code to try out:

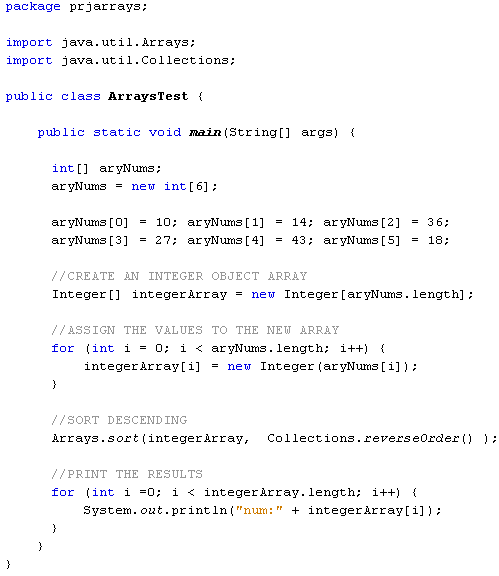


The for loop at the end will go round and round printing out the values in each array position. When the code is run, the Output will look like this:



As you can see, the array has been sorted in ascending order.

Sorting in descending order, however, is only possible either by writing your own sorting code, or converting your array to Integer objects then importing from the Collections library. If you need to a descending sort, here's some code that does just that (skip this code, if you want):



All a bit messy, I'm sure you'll agree!

In the next lesson, we'll take a look at arrays and strings.

# **Arrays and Strings**

You can place strings of text into arrays. This is done in the same way as for integers:

**String[ ] aryString = new String[5] ;**

**aryString[0] = "This";  
aryString[1] = "is";  
aryString[2] = "a";  
aryString[3] = "string";  
aryString[4] = "array";**

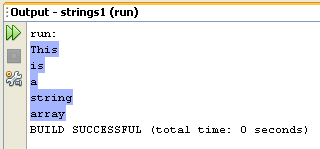
The code above sets up a string array with 5 positions. Text is then assigned to each position in the array.

Here's a loop that goes round all the positions in the array, printing out whatever is at each position:

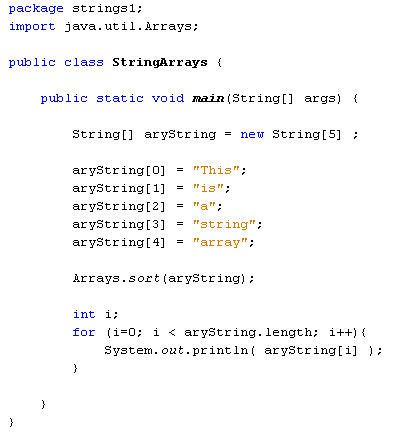
**int i;  
for ( i=0; i < aryString.length; i++ ) {  
System.out.println( aryString[i] );  
}**

The loop goes round and round while the value in the variable called i is less than the length of the array called aryString.

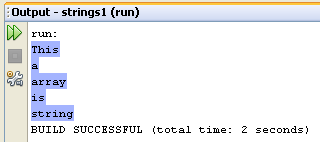
When the above programme is run, the Output window will look like this:



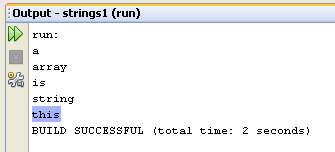
You can perform a sort on string arrays, just like you can with integers. But the sort is an alphabetical ascending one, meaning that "aa" will come first over "ab". However, Java uses Unicode characters to compare one letter in your string to another. This means that uppercase letter will come before lowercase ones. Try the following code:



When the programme is run, the Output window will display the following:



Although we've sorted the array, the word "This" comes first. If this were an alphabetical sort, you'd expect the word "a" to come first." And it does if all the letters are lowercase. In your programming code, change the capital "T" of "This" to a lowercase "t". Now run your programme again. The Output window will now display the following:



As you can see, the word "this" is now at the bottom. We'll have a closer look at strings in the next section, so don't worry too much about them now. Instead, try these exercises.

**Exercise**  
Set up an array to hold the following values, and in this order: 23, 6, 47, 35, 2, 14. Write a programme to get the average of all 6 numbers. (You can use integers for this exercise, which will round down your answer.)

**Exercise**  
Using the above values, have your programme print out the highest number in the array.

**Exercise**  
Using the same array above, have your programme print out only the odd numbers.