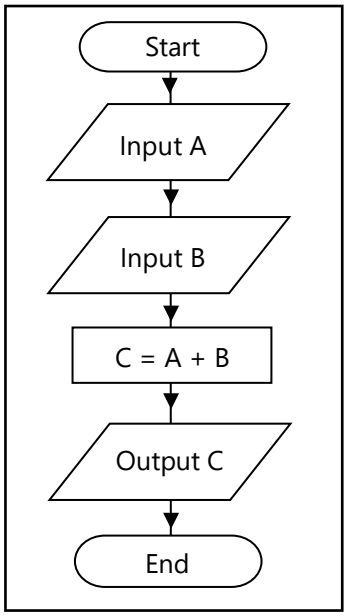
Programming in Java

An Object Oriented Approach
To SEC Level

By Marlene Galea
Head of Department Computing

PROGRAMMING IN JAVA

Introduction to Programming



Flowchart for an algorithm to input 2 numbers and output their sum.

Computers can only perform very simple operations like comparing two values or adding two numbers. They perform complex tasks by carrying out large numbers of simple operations after each other. Therefore computerised tasks need to be specified in perfect detail by programs.

What is an algorithm?

An algorithm is a step by step sequence of instructions designed to solve a problem.

Algorithms can be expressed in different ways including structure charts, pseudocode, flowcharts and programming languages.

What is a programming language?

A programming language involves a *vocabulary* and *set of grammatical rules* for instructing a computer to perform specific tasks.

There are many programming languages and many means of creating a program. Examples of programming languages include Cobol, C# and Java.

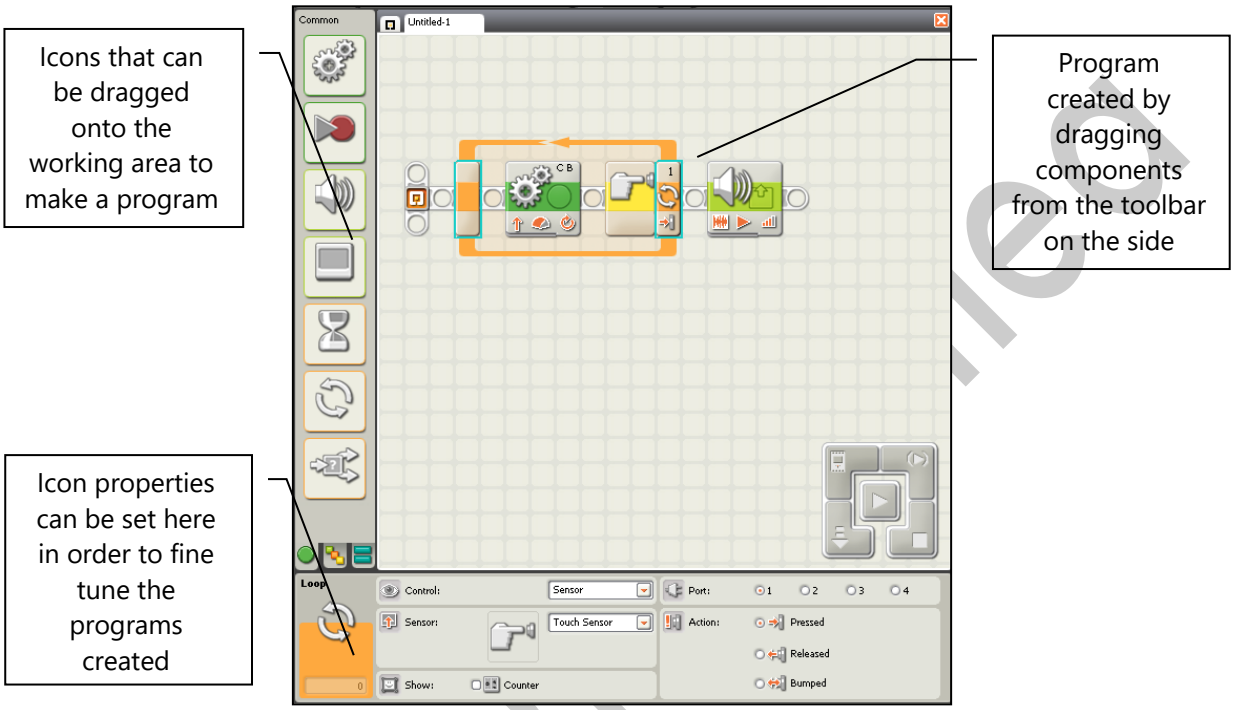
What is a program?

A program is an *organized list of instructions* that the computer can handle and when executed, causes the computer to behave in a predetermined manner.

Without programs, computers are useless.

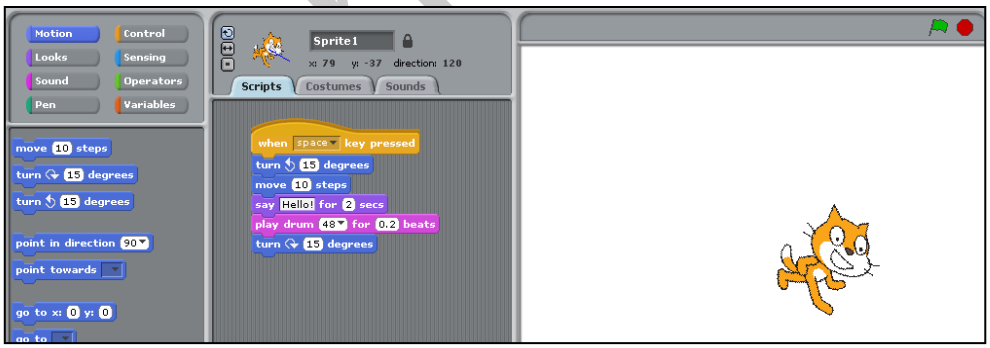
Icon-based programming

Certain programming interfaces make creating programs easier because they allow us to create programs using icon-based programming. NXT-G developed for Lego Mindstorms Robotics (shown below) offers an example of icon-based programming.



The *Lego Mindstorms NXT* Programming Interface

Other icon based programming interfaces include that of 'Scratch'



Programming using *Scratch*

Features of Icon-based programming

- Generally easier and more intuitive;
- One can create programs faster;
- Programs may require more system resources to run than equivalent text-based programs.

Text-Based Programming: Introduction to Java

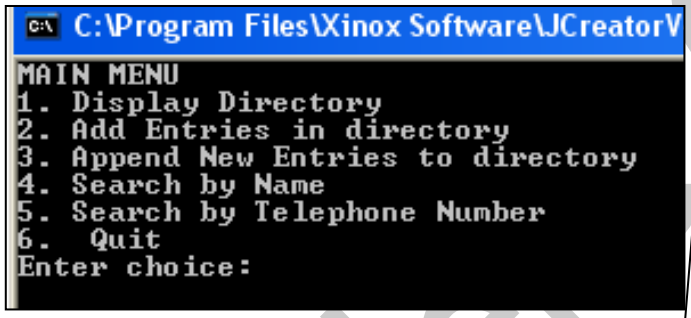
What determines the choice of a programming language?

Every language has its strengths and weaknesses. The choice of language depends on:

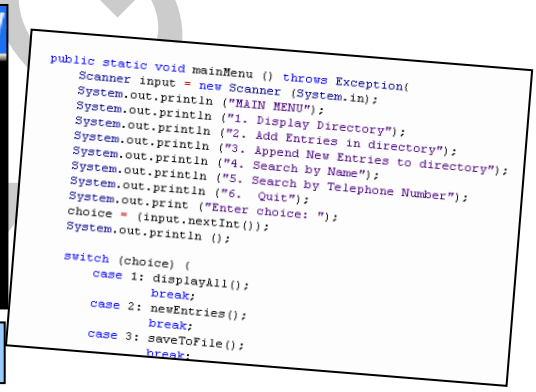
- the type of computer the program is to run on;
- what sort of program it is;
- the expertise of the programmer.

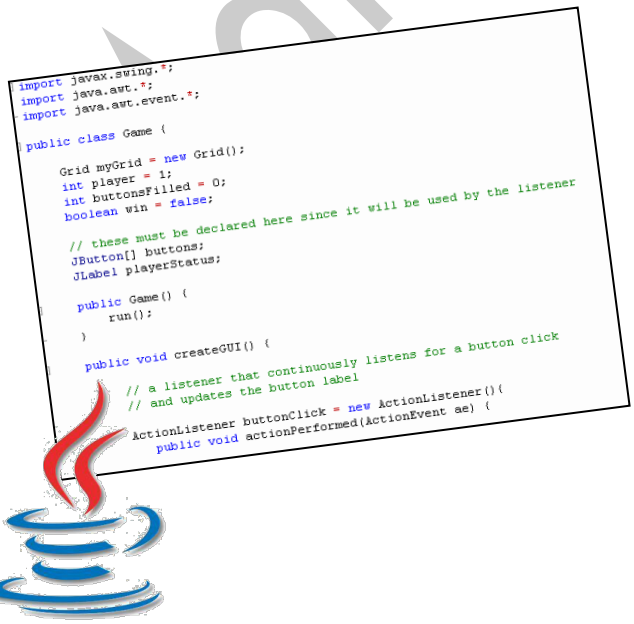
What is Java?

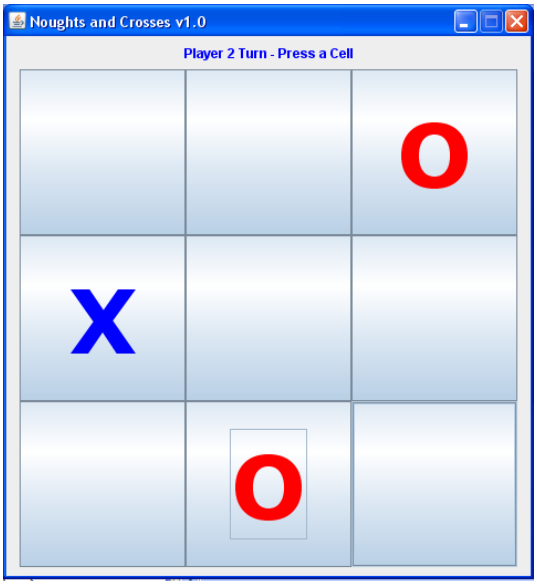
Java was originally developed by James Gosling at Sun Microsystems (now a subsidiary of Oracle Corporation) and was first released in 1995. Java is a third generation, general purpose language like Pascal, C and C#. It is one of the most important programming languages and is widely used: from making application software to web applications.



A Text-Based Java Application







A Graphic-Based Java Application

Features and Benefits of Java

- *Portability*: Java is platform independent. Java programs are portable between different types of computers that have a JVM (Java Virtual Machine). This is because Java programs are first translated into bytecode which can then run on any machine that has a Java interpreter (which is part of the JVM).
- *Object Oriented:* Because it is an object oriented language, Java allows flexible modular programming and code reusability through encapsulation, inheritance and polymorphism.
- *Java Programs are smaller in size* so they are simple, economic and efficient.
- *Robust:* Java is robust (reliable) because it emphasises early checking for errors. Java compilers can detect problems that would first show up during execution time in other languages. For instance Java has a runtime exception-handling feature.
- *Secure*: Java prohibits viruses etc because code runs inside the virtual machine.
- *Easy to write, compile and debug*: Java is easier than other object oriented programming languages like C++ because it uses automatic memory allocation and garbage collection.
- *Java is distributed*: it makes distributed computing easy as writing networked program is easy. Distributed computing involves several computers on a network working together.
- Java is multithreaded: Multithreading is the ability to perform several tasks independently and simultaneously within a program.

Disadvantages of JAVA

- *Performance:* Java is an interpreted language, so because of the translation process, programs written in Java have speed issues. However the speed of a Java program is good enough for most interactive applications (because in interactive applications the CPU is waiting for user input most of the time anyway so the speed of translation will not be the main bottle neck).

Getting Java on your computer

What you will need to develop and run Java applications

In order to develop and run Java applications you will need a JDK (Java Development Kit) and you should also get an IDE (Integrated Development Environment). The following explains these terms and their role.

Explaining JDK, JRE, JVM and IDE

Java Development Kit (JDK)

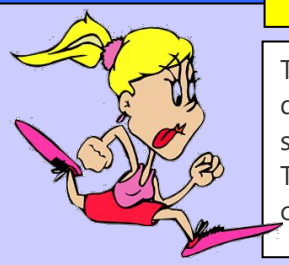
A JDK is a Java software development environment from Sun. Each new version of the JDK adds features and enhancements to the language. When Java programs are developed under the new version, the Java interpreter (Java Virtual Machine) that executes them must also be updated to that same version.
A JDK includes the JRE, Java compiler, debugger and other tools for developing Java applets and applications

Java compiler

Debugger

Other tools for developing Java applets & applications

Java Runtime Environment (JRE)



The Java Runtime Environment (JRE) is what you get when you download Java software. The The JRE is the runtime portion of Java software, which is all you need to run it in your Web browser.
The JRE is an implementation of the JVM, all the Java platform core classes and supporting libraries

Java platform core classes

Java Virtual Machine (JVM)

The Java Virtual Machine (JVM) is software that converts the Java intermediate language (bytecode) into machine language and executes it.. A JVM is a Java operating program that runs Java programs. It creates an environment for executing Java code that behaves like a computer separate from the one it is running on.

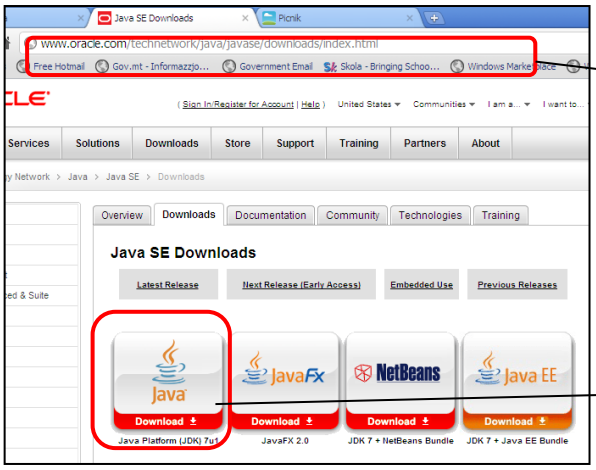
Supporting Java platform libraries

Integrated Development Environment (IDE)

An IDE e.g. *BlueJ* or *JCreator* is used to edit, compile and debug Java code.

Installing the JDK (Java Development Kit)

Download the JDK from:



1 Go to the URL given above.

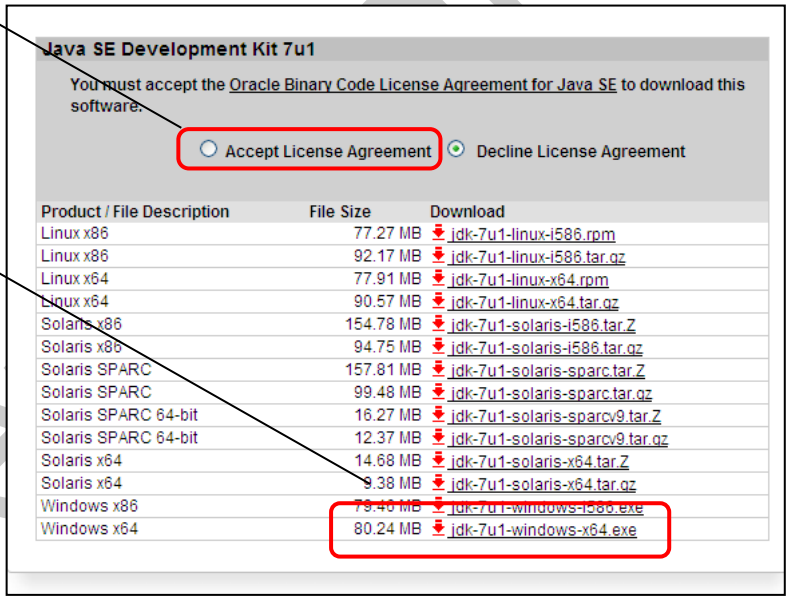
2 Select 'Download'

Accept License Agreement

3

Select product to download

4

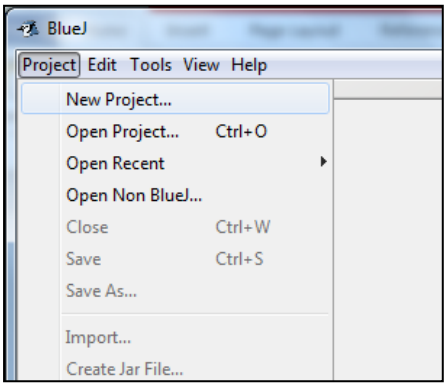


Using BlueJ to create Java programs

Downloading BlueJ

Download and install from: http://www.bluej.org/

(You may choose to download both BlueJ and the JDK from here).

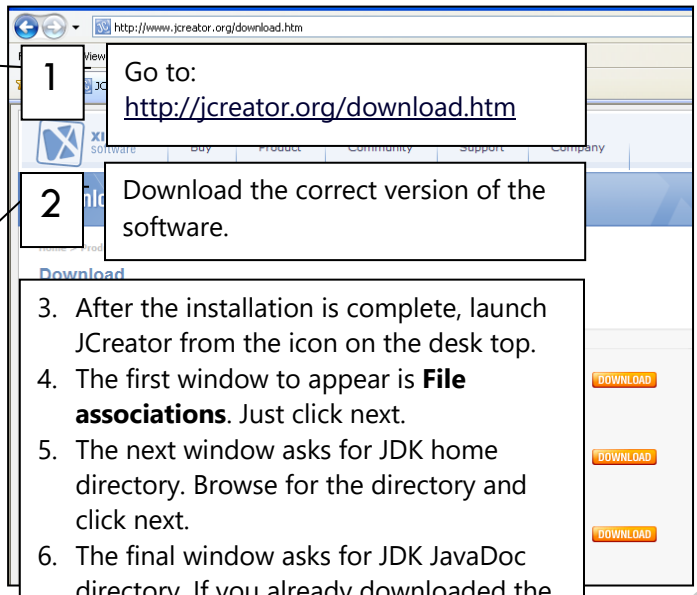


Starting a Project

Select Project>New Project

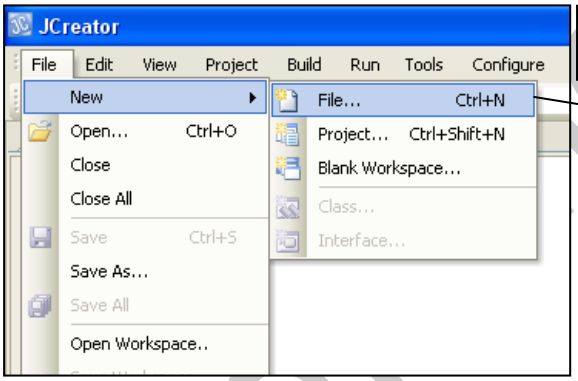
Using JCreator to create Java programs

Downloading JCreator

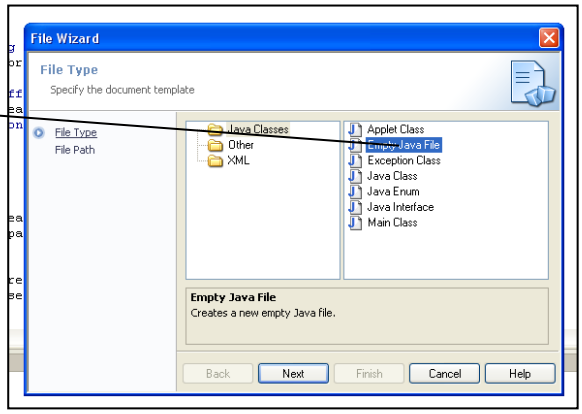


1. Go to: http://jcreator.org/download.htm
2. Download the correct version of the software.
3. After the installation is complete, launch JCreator from the icon on the desk top.
4. The first window to appear is **File associations**. Just click next.
5. The next window asks for JDK home directory. Browse for the directory and click next.
6. The final window asks for JDK JavaDoc directory. If you already downloaded the JavaDoc file, browse for it and click finish. Otherwise click finish to continue without the documentation.

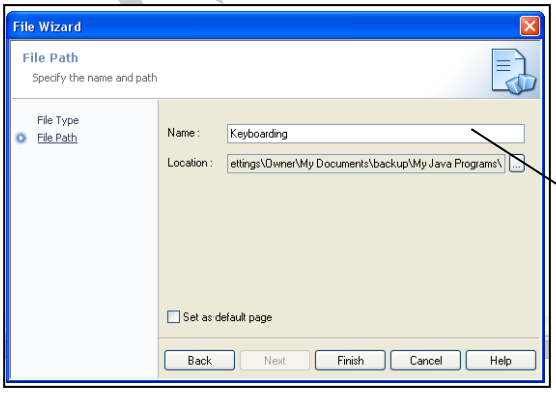
Create a folder on your desktop called 'My Java Programs'



1. Select *File/New/File*



2. Select *Empty Java File* and click *Next*



3. Specify the path to save in.
   Give the file name
   Click *Finish*

Running a Java Program

What happens when we run a Java program?

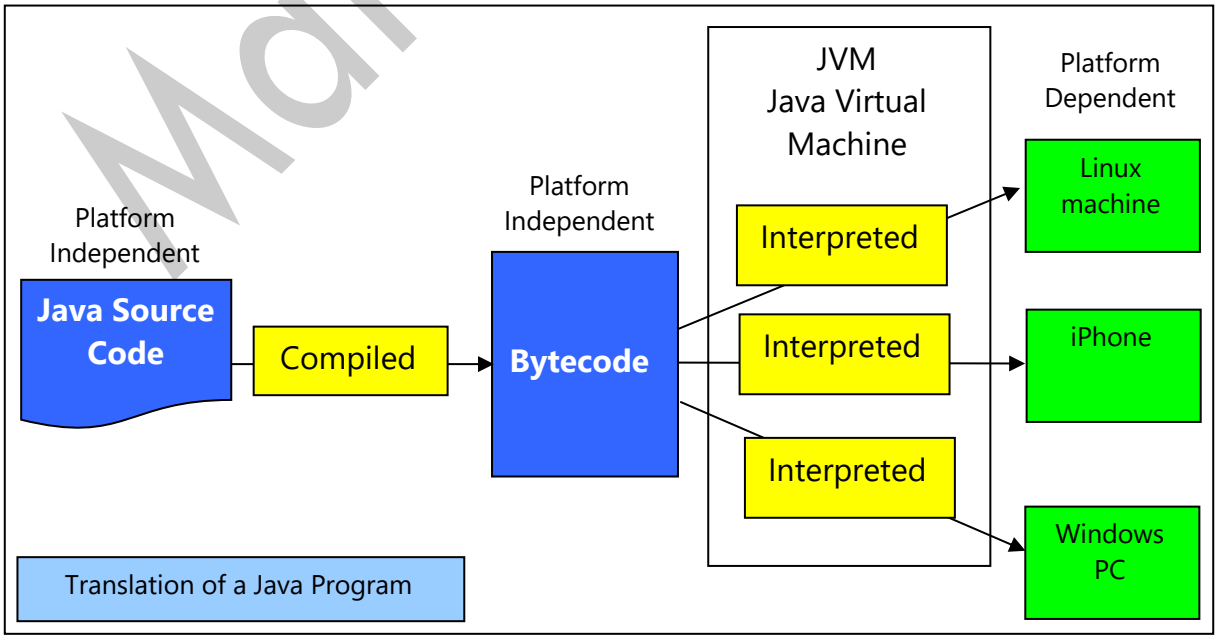
The computer does not understand Java, therefore all the programs we write need to be translated into a form the computer can handle in order for the instructions we write to be executed. Programs called 'translators' are used to do this. In Java translation is done in two steps: using first a compiler than an interpreter.

What is Platform independence?

A platform is the hardware and system software on which application software can run. Different platforms normally run different application software. However, Java is platform independent: it will run on different platforms. This is because the Java compiler does not produce executable code but *bytecode*. Then the JVM on the machine that will run the program changes the bytecode to executable code suitable for the platform it is on.

Two step translation

- The program is first compiled to produce bytecode
  - At this stage the user cannot view the actual code but the program is still platform independent. Therefore the *bytecode is ideal for distribution*.
- Then the JVM on the machine the program will run on uses an interpreter to translate the program into executable code. The executable code is platform dependent because it will be different for a Linux Machine and a Windows PC etc.

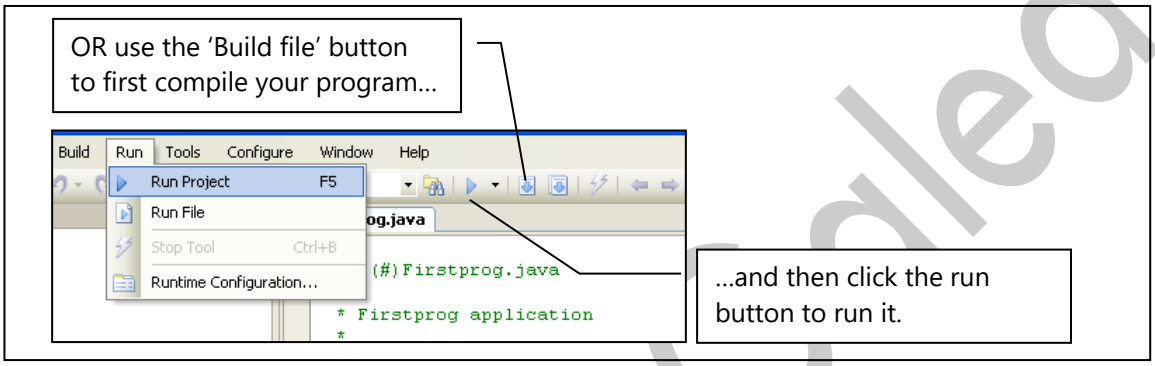


How do we run a Java program?

In order to run a program the system needs to first translate it and then execute (obey) the translated instructions one by one.

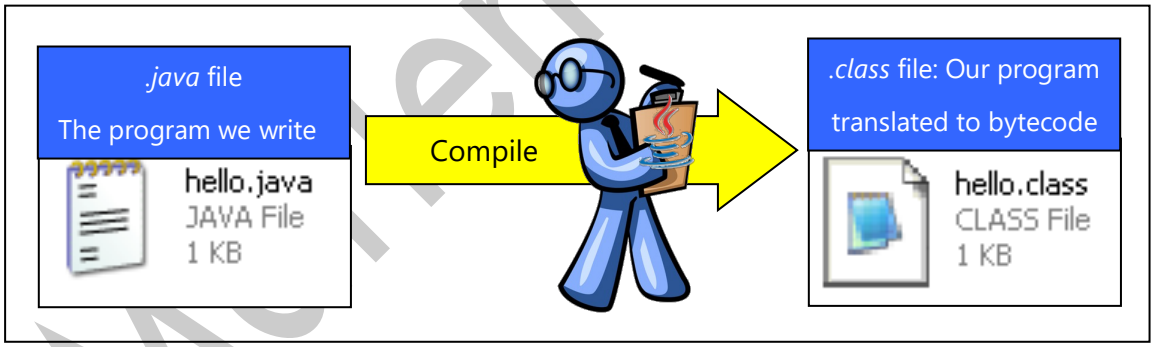
This is how you can run a program using JCreator:

1. Write your program.
2. Click the 'Run' Menu.
3. Select 'Run Project'. This will compile and then run your program.



Running an application

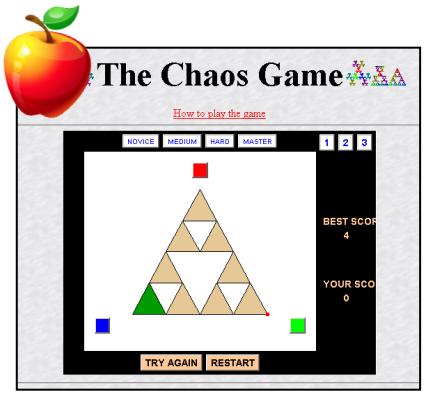
.class and .java files Files



.java files are compiled to give .class files

- The programs we write form the source code file and have a .java extension (e.g. hello.java)
- When our source program (the .java file) is compiled the equivalent .class file is created, this is made up of bytecode.
- The JVM on which the program will run will then translates the bytecode (the .class file) into something the platform it is on understands and then runs the program.

What is a Java Applet?



Applets are used to provide interactive features to web applications. A Java applet is delivered to the users in the form of bytecode. Since Java's bytecode is platform independent Java applets can be executed by browsers for many platforms, including Microsoft Windows, UNIX and Mac OS. Java applets can run in a web browser using a JVM.

Object Oriented Programming

Java is an object oriented Language. This means that data is treated as objects to which methods are applied.

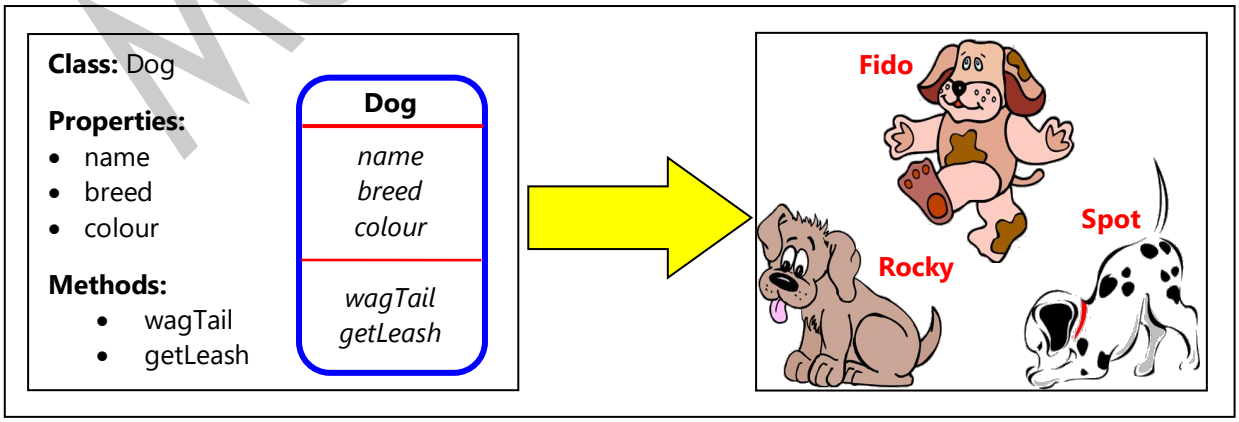
What is are classes and objects?

A *class* is the blueprint from which individual objects are created.

- The class specifies the properties (data or Object Variables) and methods (actions) that objects can work with.
- The class 'Dog' shown here has *properties* (name, breed etc) and *methods* (actions) (wagTail, getLeash etc)

An *object* is an instance of a class.

- If you have a pet dog called 'Fido', Fido is an object (an instance) of the class 'Dog'.
- Objects like 'Fido', Spot' and 'Rocky' are all instances of 'Dog'.



The class 'Dog' has Object Variables (properties) and Methods. Fido', 'Rocky' and 'Spot' are instances of the class 'Dog' .

Three Principles of Object Oriented Programming

Object Oriented Programming is characterised by the following three principles:

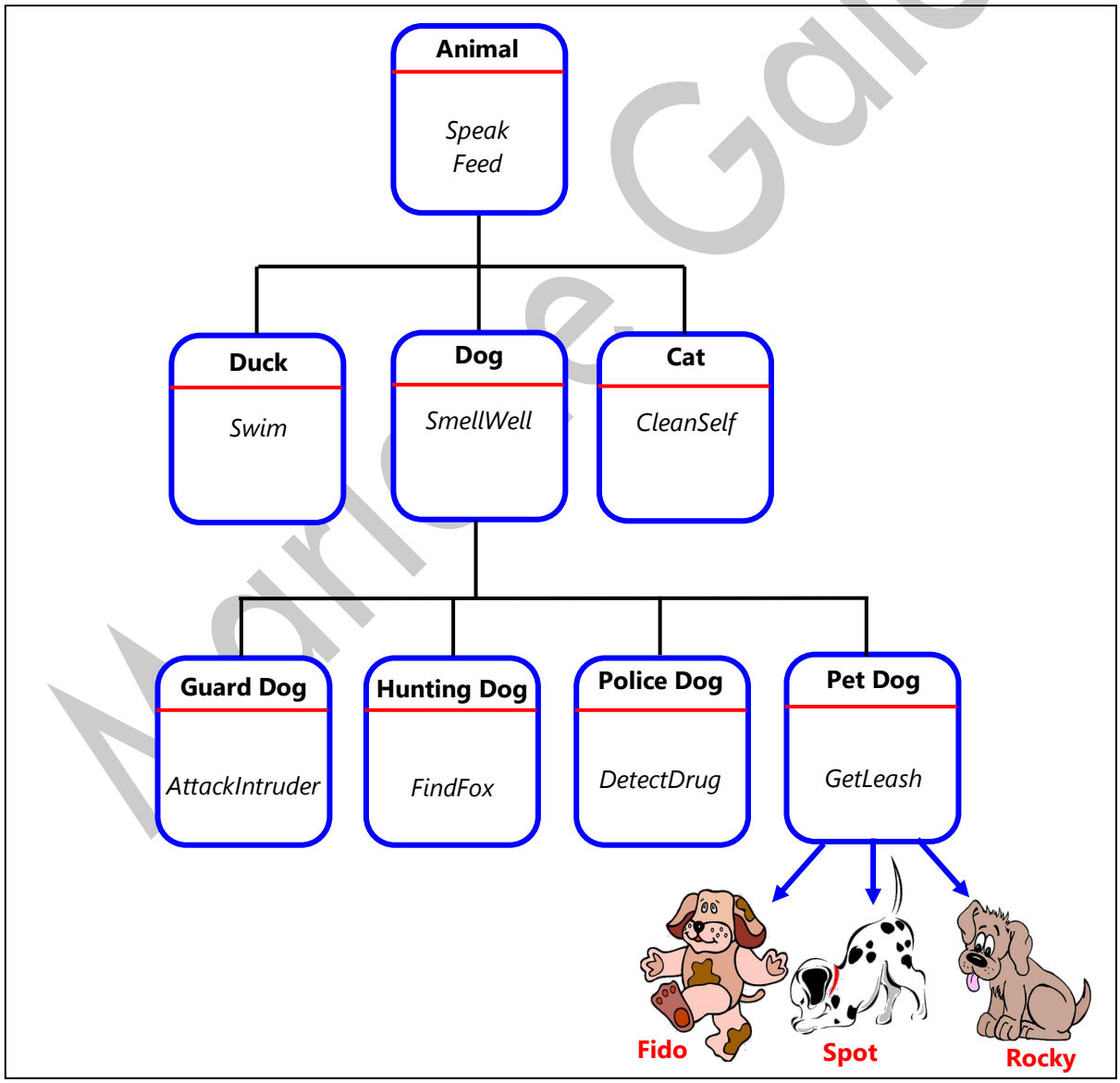
Inheritance

Polymorphism

Encapsulation

Inheritance

- The items in black are **classes**. (Animal, fish etc)
- Classes have *properties* (name, breed etc) and *methods* (actions) (Feed, swim etc)
- Child objects *inherit* the properties and methods of their parents. (E.g. All dogs inherit the feeding method from the parent class 'Animal' etc)



Inheritance: All dogs inherit the feeding method from the parent class Animal.

Polymorphism

Polymorphism is the ability of an action or method to do different things depending on the object that it is acting upon. For instance a parent class reference (e.g. 'speak') can be used to refer to a child class object: E.g. the method 'speak' in the animal class would do different things when called from the child classes, Dog, Cat etc. Overloading, overriding and dynamic method binding are three types of polymorphism.

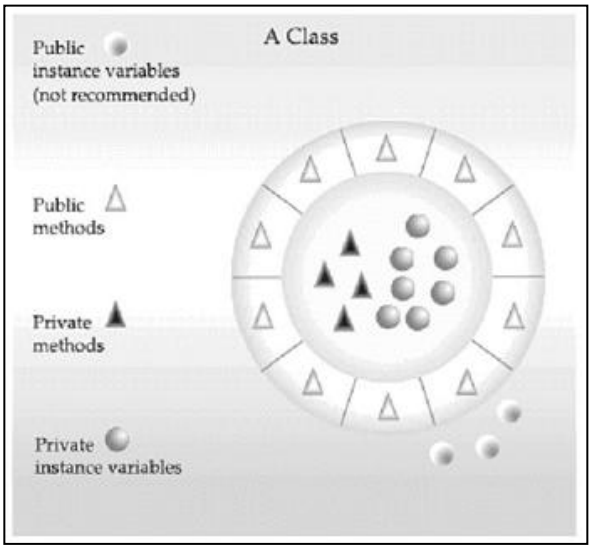
For example the three subclasses Cat, Dog and Duck are based on the Animal abstract class and can each have their own speak() method. Although each method reference is to an Animal, the program will resolve the correct method reference at runtime. This is an example of Dynamic (or late) method binding.

Polymorphism: the Animal Class could have a method called 'speak'. All animals would do different things when this method is called.

Encapsulation

Encapsulation is the ability of an object to be a container (capsule) for related properties (i.e. data variables) and methods.

Encapsulation allows *data hiding* so objects can shield variables from external access. Variables which are marked as *private* can only be seen or modified through the use of public *accessor* and *mutator* methods. Methods can also be completely hidden from external use. In Java methods that are visible externally can only be called by using the object's front door (i.e. there is no 'goto' branching concept).



Encapsulation: Private properties and methods can only be accessed through accessor or mutator methods.

Your very first Class

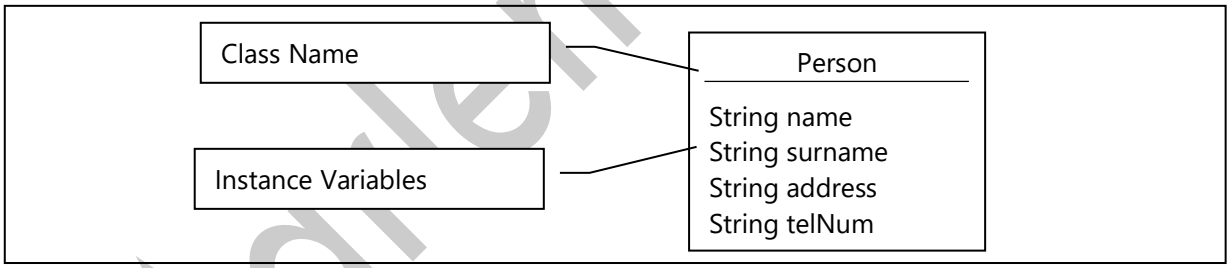
Let's say we're going to develop an application to handle a school system. The school system will need to handle persons: including ancillary staff, teachers and students. Therefore we should have a class to be a blueprint for these persons.

Object Properties (Object Variables)

We will therefore create a class called **Person** in order to later create instances of it to represents the objects (people) in the school. These objects will all have a name, surname, address and telephone numbers, therefore the class will have the following properties:

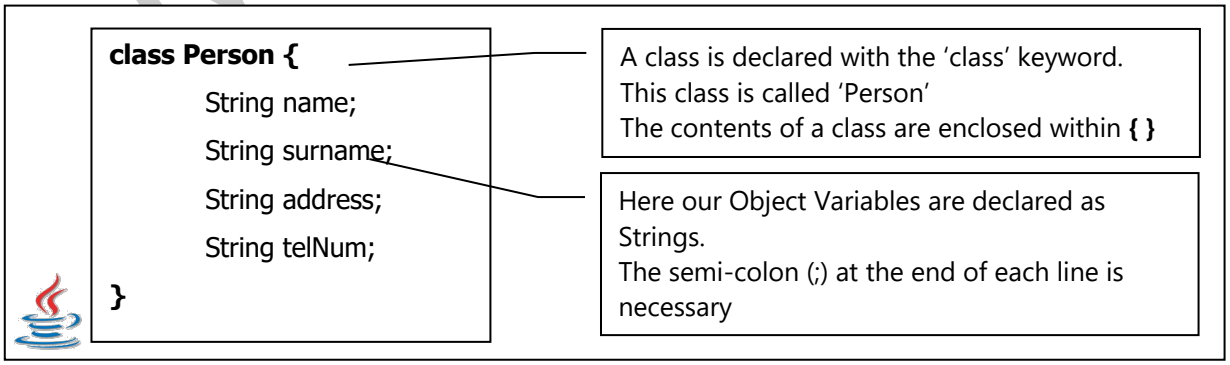
Property (Variable name)	Description	Data type **NOTE:** in Java a variable of type 'String' can hold words
name	This will hold the person's first name and will be a word.	String
surname	This will hold the person's surname and will be a word.	String
address	This will hold the person's address and will be a few words long.	String
telNum	This will hold the person's telephone number and since no arithmetic will be carried out on this number it can be treated as a non-number.	String

The Class **Person** can be described thus:



The Properties of the class called 'Person': 5 object variables of type String

In Java we implement the above using a class which we will call **Person**:

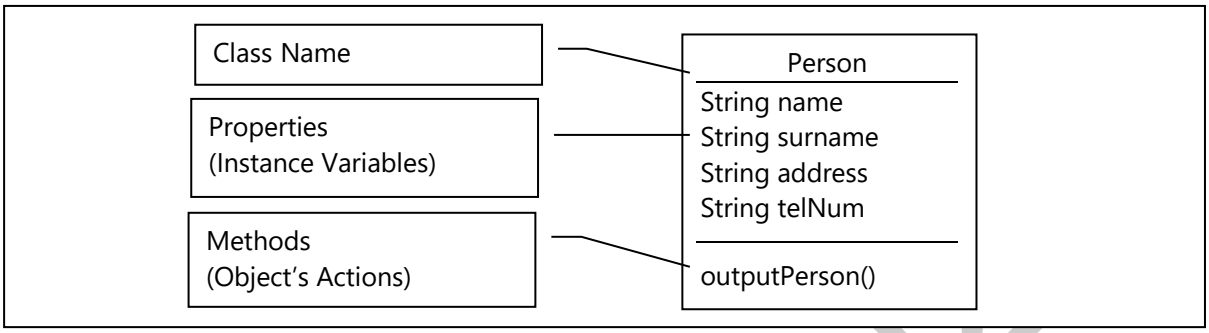


Java implementation of a simple class called 'Person'

Writing Methods: Object Actions

Our objects in the example above: Dog, Cat etc have methods like getLeash and wagTail. Methods are actions for our object.

Let's give our Class **Person** a method that outputs the person's details on the screen.



The properties and methods of the class called 'Person'

In Java we implement the above thus:

```
class Person {
        String name;
        String surname;
        String address;
        String telNum;

        public void outputPerson(){
                System.out.println (this.name);
                System.out.println (this.surname);
                System.out.println (this.address);
                System.out.println (this.telNum);
        }
}
```

This is method outputPerson(). It is a *'public'* method as it can be accessed from outside this class. It is *'void'* as it does not return a value.

The contents of the method are held between { }

This line instructs the computer to output the address of *'this'* object.

The Properties and methods of the Object called 'Person'

Structure of a method

```
public void outputPerson(){
    // body of method
}
```

Method name – can be any Java legal identifier

Parameter list – a list of type and identifier pairs that receive values of arguments passed to method. This is empty in methods with no parameters.

Public/private – a public method is available to other classes, a private method isn't.

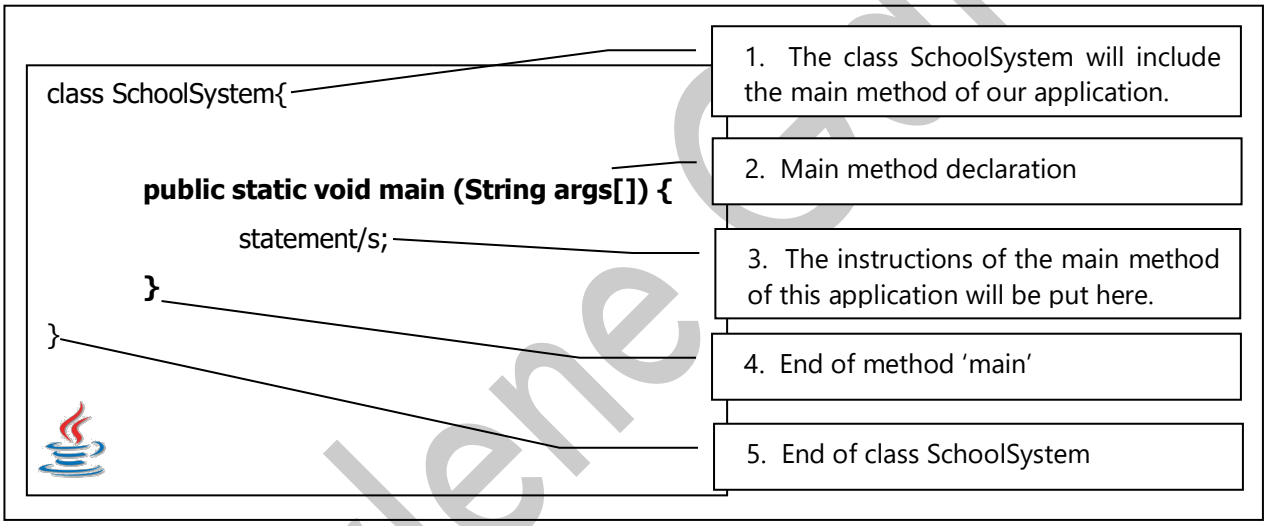
Return type – specifies the type of data returned by the method. If the method does not return any data the return type is 'void'.

The Main Class and Main Method

In the class **Person** we have created the blueprint for all persons objects that might be needed in our application.

A Java application normally consists of more than one class however. One of these classes needs to be the *main class*: the main class always includes a method called 'main' which is the *main method*. The main class is important because this is the class the Java Virtual Machine looks for to start executing the program. In an application you can have many methods in different classes, you may also have many methods in the main class but *you can only have one main method* because this is where the program starts running.

We will now create a main class and use instances of **Person** in its main method.



The Main Class and Main method

Main Method Declaration

Now let's take a closer look at the main method signature:

Public/private – a public method is available to other classes, a private method isn't. The main method is public so the JVM can begin execution of the program.

String args[] - This signifies that the user may enter parameters to the java program at command line. We can use both String[] args or String args[].

```
public static void main (String args[]) {
   // body of method
}
```

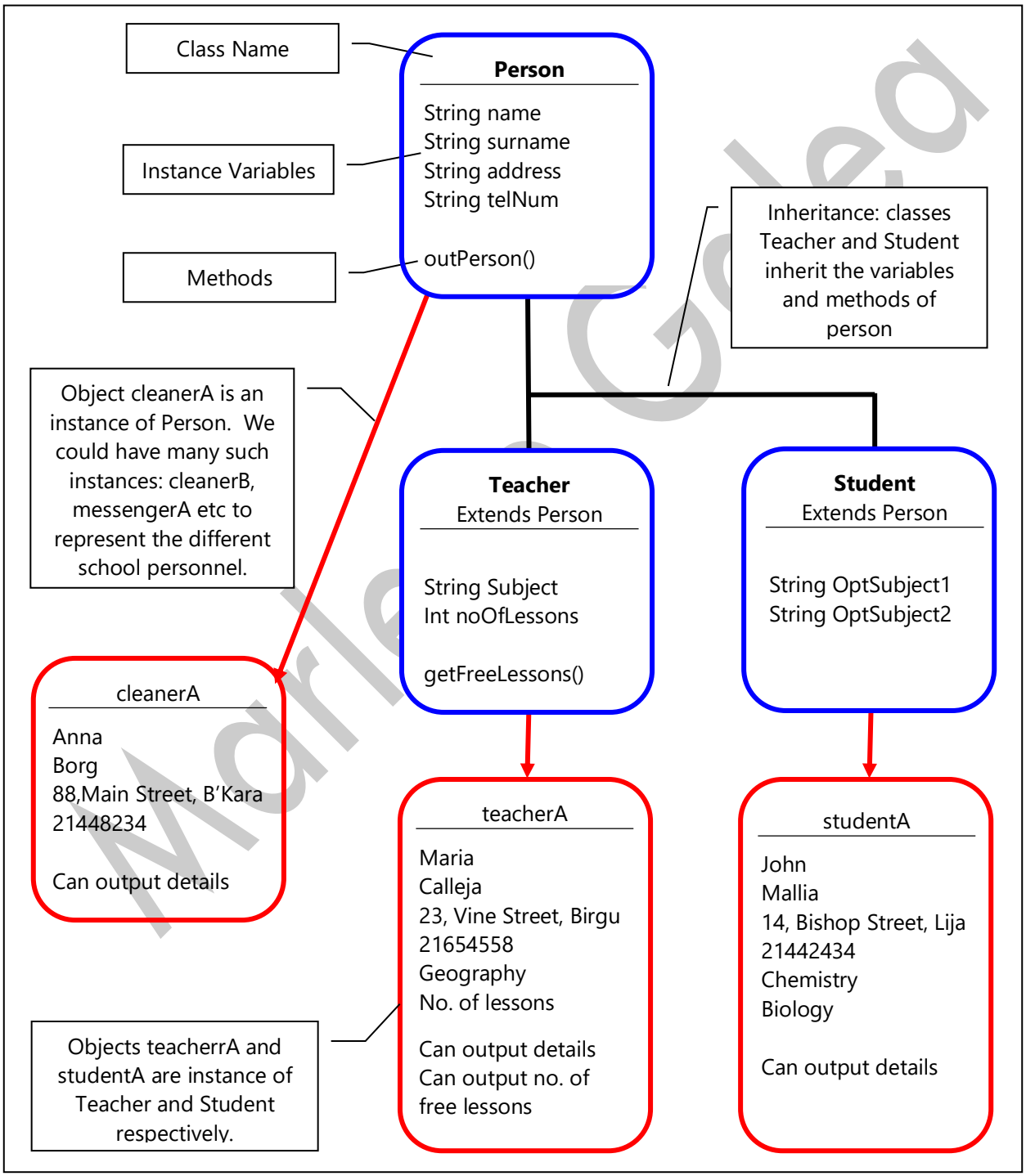
Static – the main method is 'Static' to be available for execution without an object instance. (Otherwise you need an object instance to call a method.)

Return type – specifies the type of data returned by the method. The main method does not return data, so the return type is 'void'. Once main method execution is over, the program stops. So data can't be returned by the Main method.

Creating instances of a class

Our school system application will require us to create instances of **Person:** all these objects will have the properties (variables) and methods of the class **Person** class.

The school system may also have a class called **Teacher** and another called **Student** that will inherit all the properties and methods of **Person** and possibly extend them. There would then be instances of **Teacher** and **Student** as shown below.



Creating objects: instances of a class

Now we will implement something similar in our code by creating and using objects of type **Person** in our main method.

One of the 'persons' involved in the school is the cleaner 'Anna Borg'. Therefore in our application Anna Borg will be an instance of the class **Person**.

To create cleanerA, an object of type person, we write:

```
Person cleanerA;
```

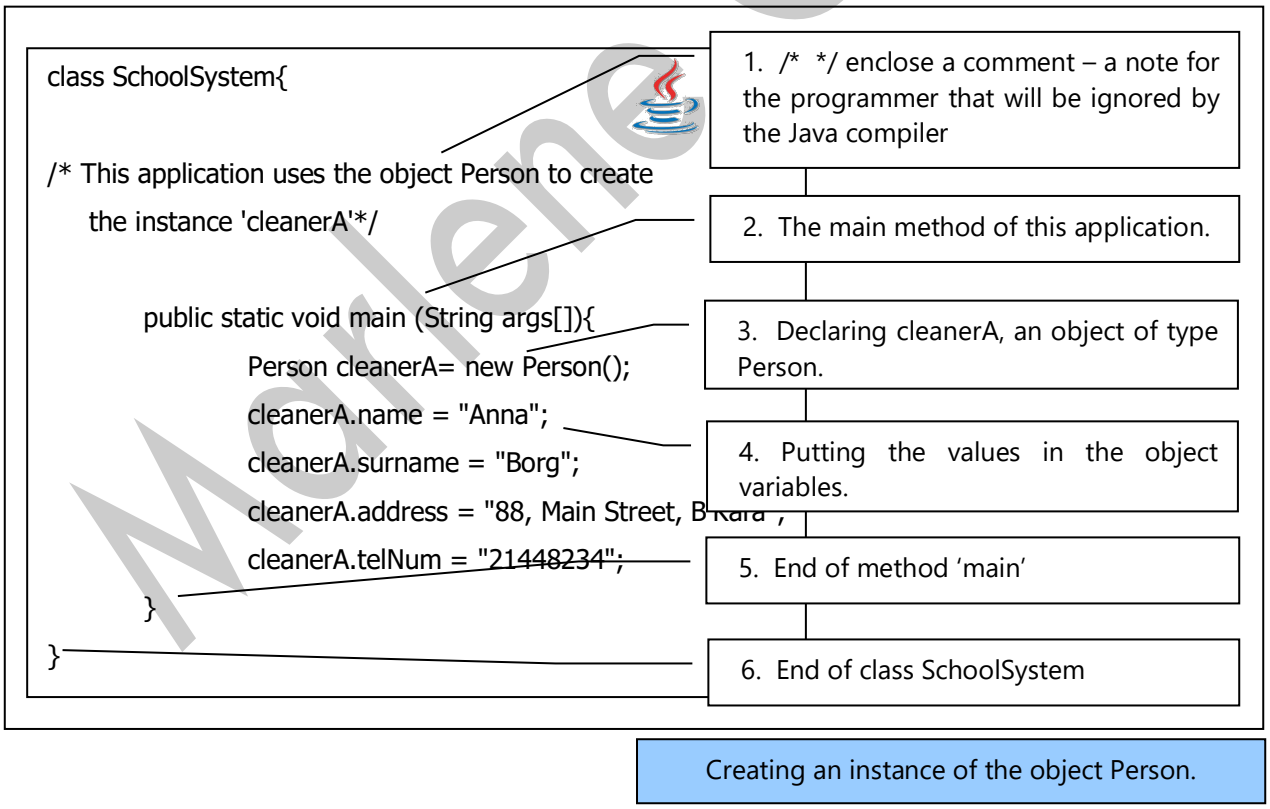
To create the memory for the object we use the keyword 'new' and then we call the default constructor method for the class, here Person(). (re constructor methods, see below.)

```
cleanerA = new Person();
```

The above can also be implemented in one line as:

```
Person cleanerA= new Person();
```

Here's what our main method will look like:



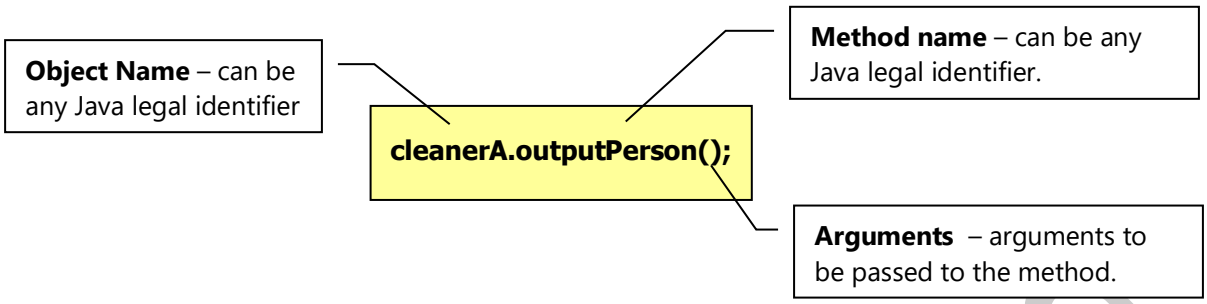
Creating an instance of the object Person.

Now we have created the object cleanerA representing Anna Borg but we have not seen anything on the screen to show it.

We could do this by using the method outPerson() in the class **Person**. Therefore now we will see how we can call and use methods in our applications.

Calling a Method

Here's how we will call the method outputPerson() to output the details of cleanerA.



Hence the code shown below will:

- Create an instance of class **Person** called 'cleanerA'
- Place the shown data in the object variables of cleanerA
- Call the method outputPerson() for cleanerA
- Execute the method outputPerson() to give the output shown here.

```
class SchoolSystem{

/* This application uses the object Person to create the
   instance 'cleanerA' and then calls the method
   outputPerson()   to output cleanerA's details.*/

        public static void main (String args[]){
            Person cleanerA= new Person();

            cleanerA.name = "Anna";
            cleanerA.surname = "Borg";
            cleanerA.address = "88, Main Street, B'Kara";
            cleanerA.telNum = "21448234";

        cleanerA.outputPerson();
        }
}
```

Output:

```
Anna
Borg
88, Main Street, B'Kara
21448234

Process completed.
```

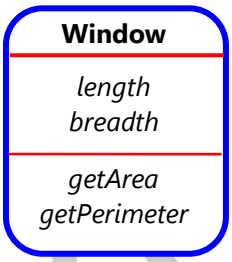
Calling the method outputPerson(); for the object cleanerA,

Calling the method 'outputPerson()'

The Constructor method

A constructor method creates an object of the class it is in: it initialises the instance variables and creates a place in memory to hold the object. Java has a default constructor that is called whenever we don't write a constructor ourselves. We can also write our own constructors.

Let's take an application that will be used to take window orders. One of the classes in this application will be **Window** and two of its instance variables would be 'length' and 'breadth'. It can be time-consuming to initialize all variables in a class every time an instance of it is created.



We may set default values for the variables length and breadth so when we initialize a Window object it would have those default dimensions. A constructor method will let us do this. A Constructor method for class Window is shown here:

```
class Window{
        float length;
        float breadth;

        public Window(){
                length = 1;
                breadth = 2;
        }
}
```

Constructor Method

Constructor Method has **no return type** because it never returns a value

Constructor Method name – same as class name

Initialising object variables

```
class winApp{
        public static void main (String args[]){
                Window kitchenWin = new Window();
                System.out.println("Length: "+kitchenWin.length);
                System.out.println("Breadth: "+kitchenWin.breadth);
        }
}
```

Calling the Constructor Method.
If we hadn't written a constructor, this would call the Java default constructor. The default constructor also takes no arguments and performs no special actions or initializations.

The Constructor Method we wrote initialises Length and Breadth to 1 and 2 respectively.
Therefore when the output statements in the main method are executed, we obtain the output shown here.

```
Length: 1.0
Breadth: 2.0

Process completed.
```

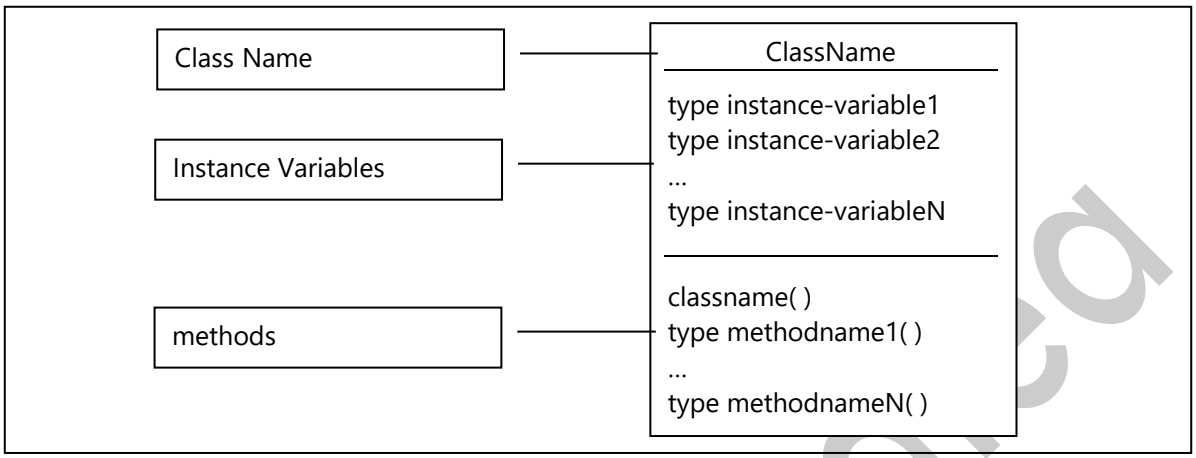
Writing a constructor method

Constructors may include parameters of various types. When the constructor is invoked using the 'new' operator, the types must match those specified in the constructor definition.

General Structure of a class

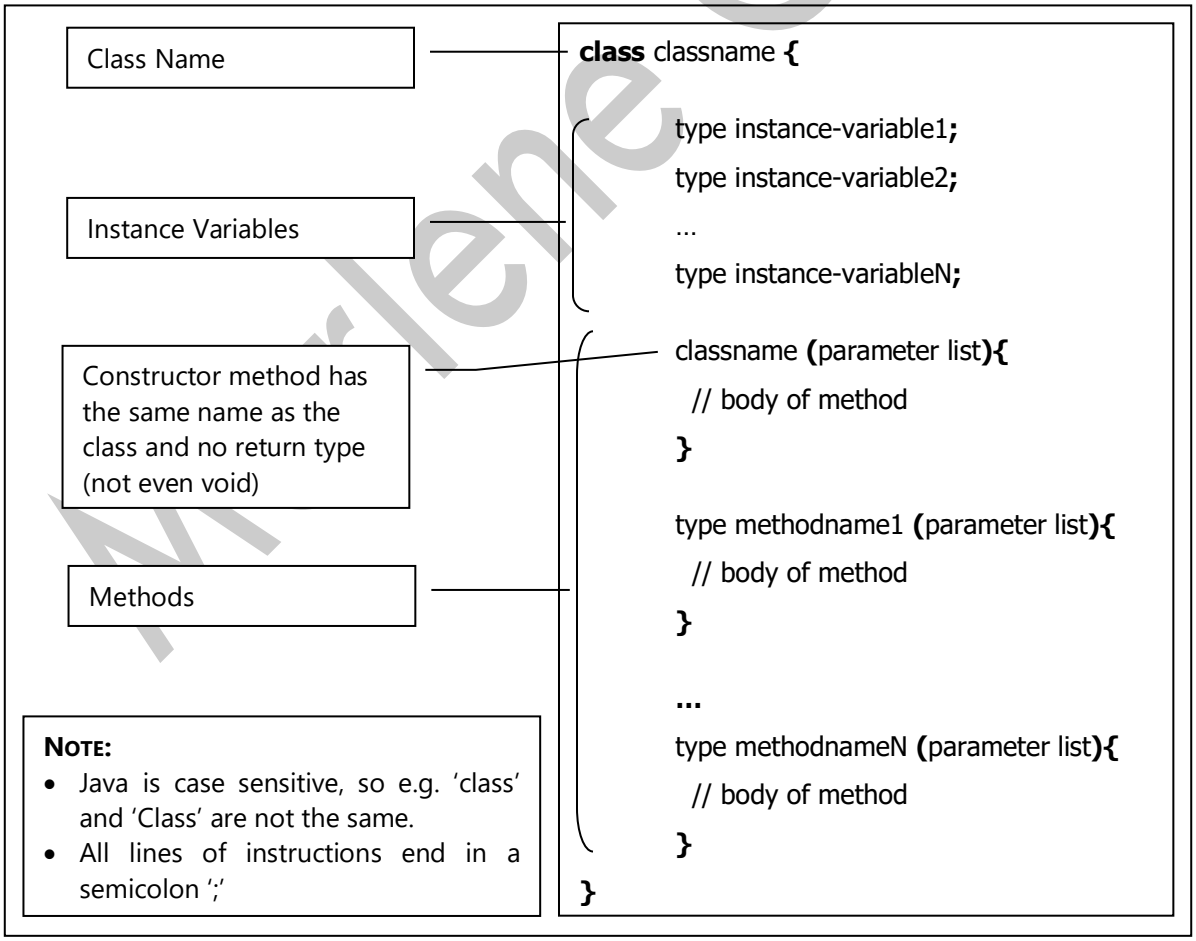
Therefore we have now seen that a class is implemented by:

- A number of instance variables that describe its properties
- A number of methods that describe its actions



Generalised Structure of a Class

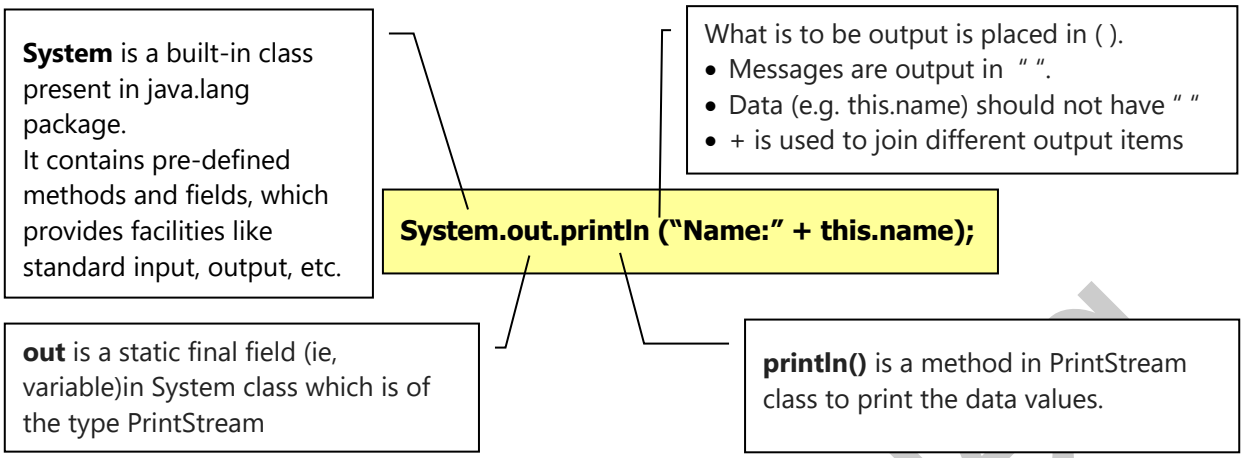
This is the general form and syntax of a Java class:



Generalised Syntax of a Class

Screen output in Java

Let's take a closer look at the output instruction in our application

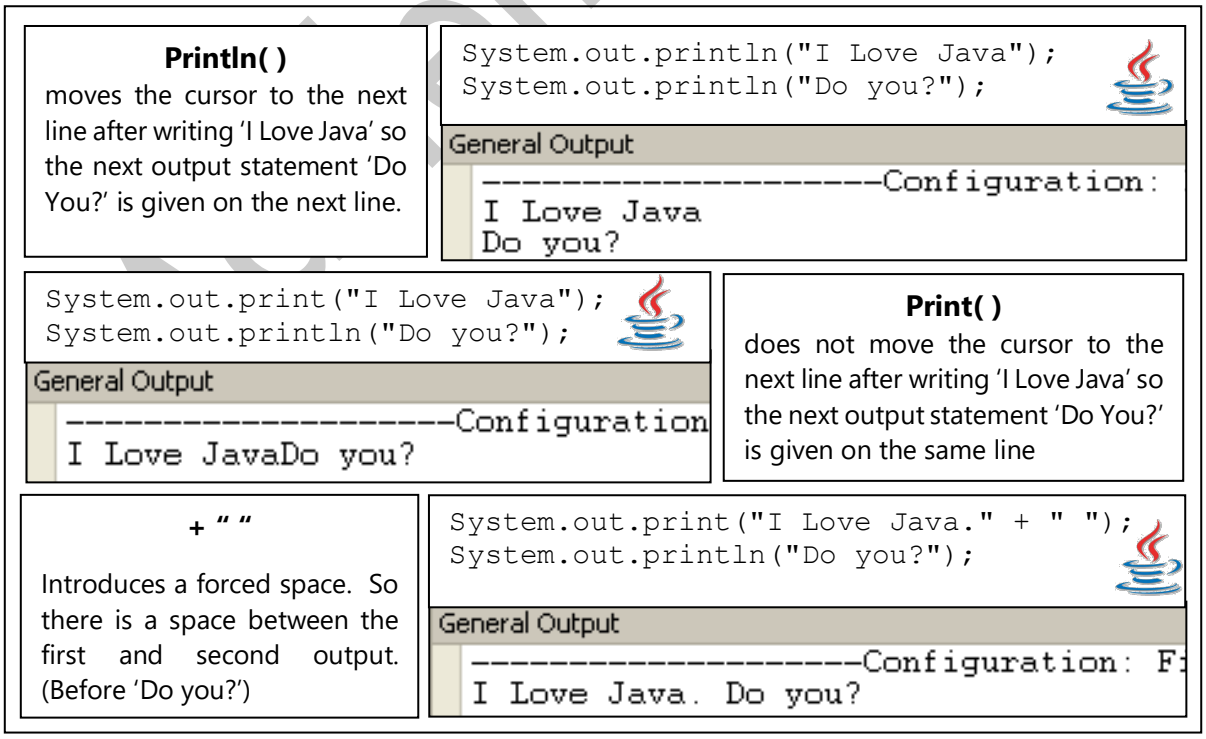


print/println

The syntax for printing the words 'I Love Java' on your screen is the following:

System.out.println ("I Love Java!");

- The *println* command causes the item in brackets to be displayed and then moves the cursor to the next line
- The *print* command causes the item in brackets to be displayed, leaving the cursor after the last letter displayed.



Print( ) and Println( )

Printing Literals

A literal is a notation for representing a fixed value in source code.

There are few character literals which are not readily printable through a keyboard. The table below shows the codes that can represent these special characters. The letter d such as in the octal, hex etc. represents a number. Escape characters

Escape	Meaning
\n	New line
\t	Tab
\b	Backspace
\r	Carriage return
\f	Formfeed
\\	Backslash
\'	Single quotation mark
\"	Double quotation mark
\d	Octal
\xd	Hexadecimal
\ud	Unicode character

Printf

Printf can be used instead of 'print' or 'println' to set the number of decimal places to print. Therefore the following program using these escape characters will give the output shown below.

```
class escchar {
        public static void main (String[] args){
            System.out.println (" Java \n is a great \n programming language.");
            System.out.println (); //to skip a line
           System.out.println ("Column \t column\t column");
            System.out.println ("Java \t Java\t Java");
            System.out.println ();
            System.out.println ("Now I ask you, \'Don't you think Java is really fun?\'");
            System.out.println ();
           System.out.printf ("Look Java can even print this number to two decimal
           places:%10.2f",3.1415 );
        }
}
```

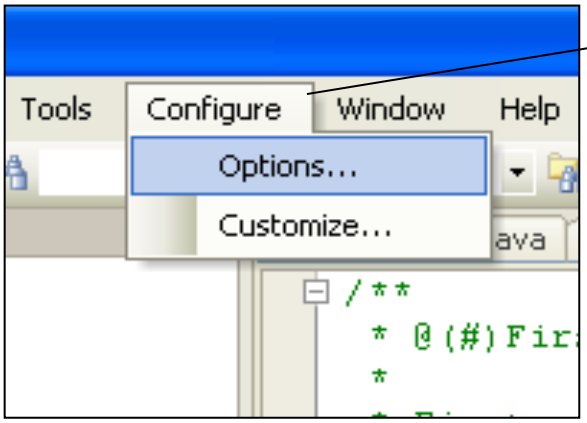
Output:

```
 Java
 is a great
 programming language.

Column    column  column
Java      Java    Java

Now I ask you, 'Don't you think Java is really fun?'

Look Java can even print this number to two decimal places:      3.14
```

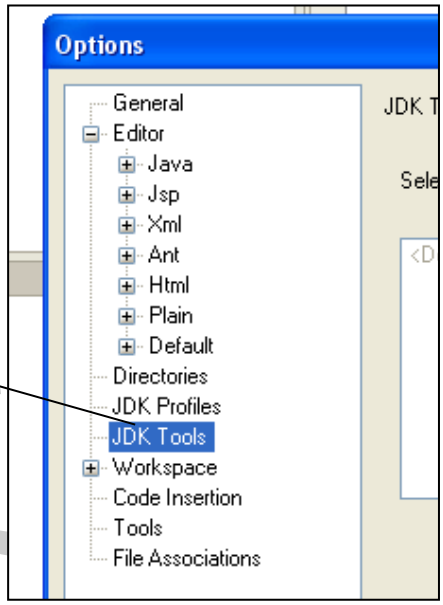
Using the escape characters \n, \t,\' and printf

Changing the output in JCreator

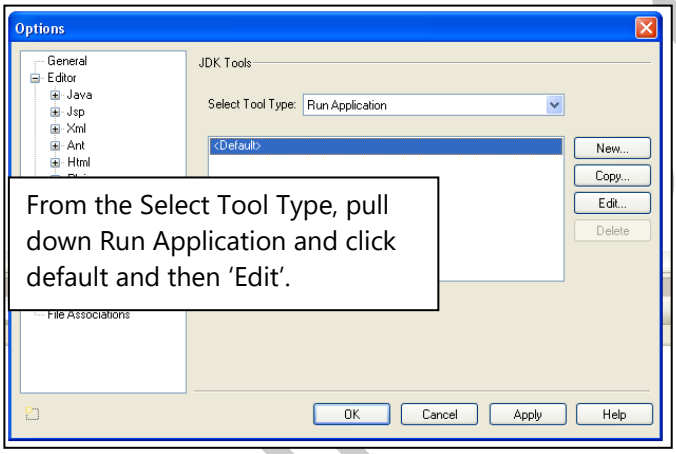
To obtain the output in the same window



1 Go to Configure\Options



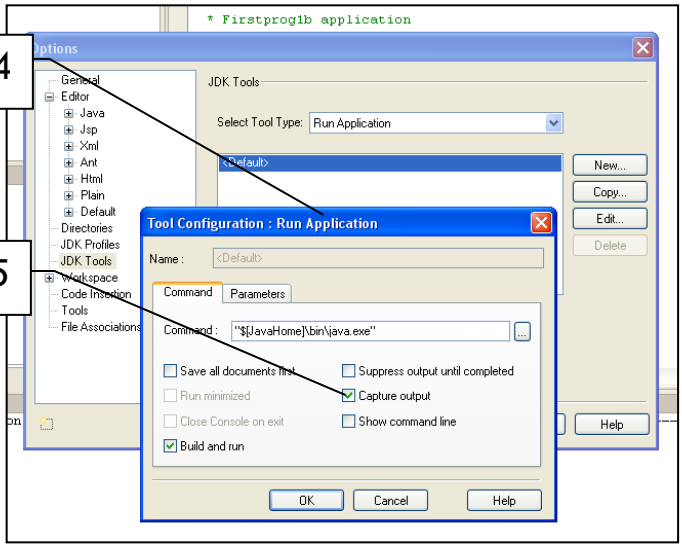
Select JDK Tools 2



3 From the Select Tool Type, pull down Run Application and click default and then 'Edit'.

Click Edit and the Tool Configuration: Run Application dialog box appears. 4

Click Capture output. 5



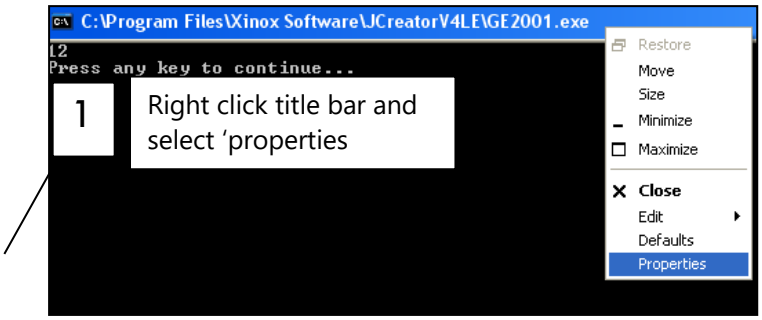
Now your output will appear in the window below your program and you will be able to scroll the output if necessary.

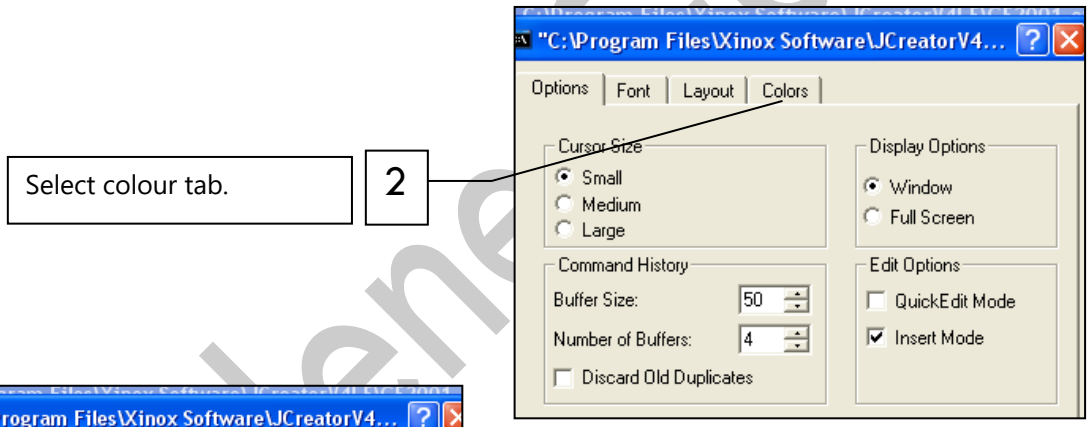
To obtain the output in a new window

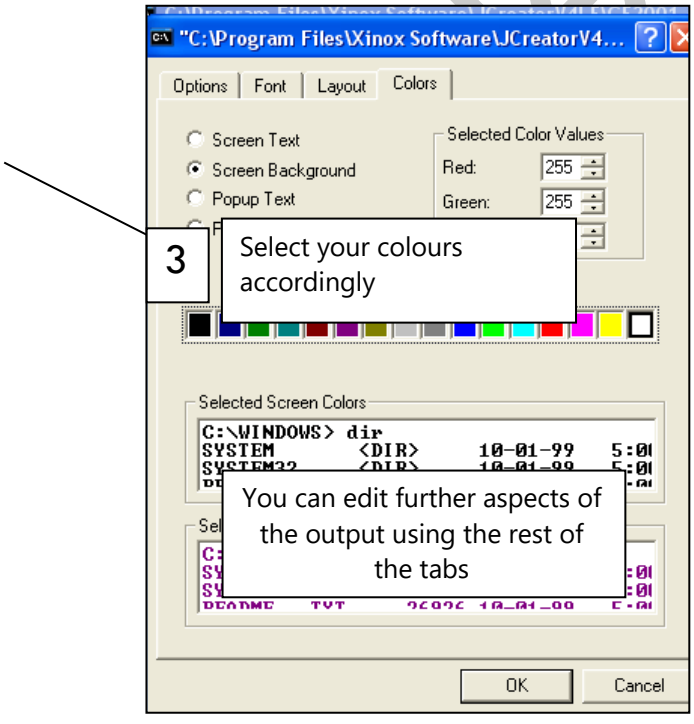
In the above dialog box, uncheck 'Capture Output' and click 'OK',.

Changing the output Window properties

If you would like your output in a new window, you can set its properties to your liking. For instance to show as black text on a white background follow these steps:



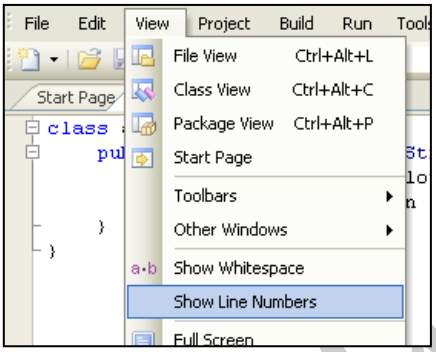




How to display line numbers in JCreator

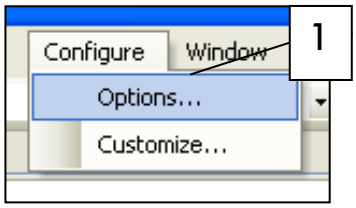
As our programs grow larger, we may appreciate having JCreator supply us with the line numbers for the lines of code on the side.

To display line numbers select 'show line numbers' from the 'View' Menu as shown.

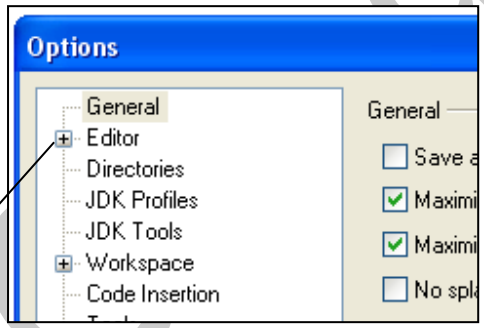


Else

To always show line numbers follow the steps below:

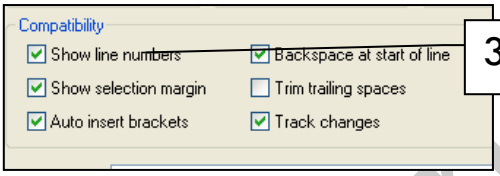


Choose 'Options from the Configure Menu



Click on Editor (+) and double click Java

2



In compatibility pane, select 'Show line numbers'.
Click 'Apply' and 'OK'.

Comments in Java Programs

Comments within programs allow us to describe the code so that later we (or anyone else) will find the program easier to follow.

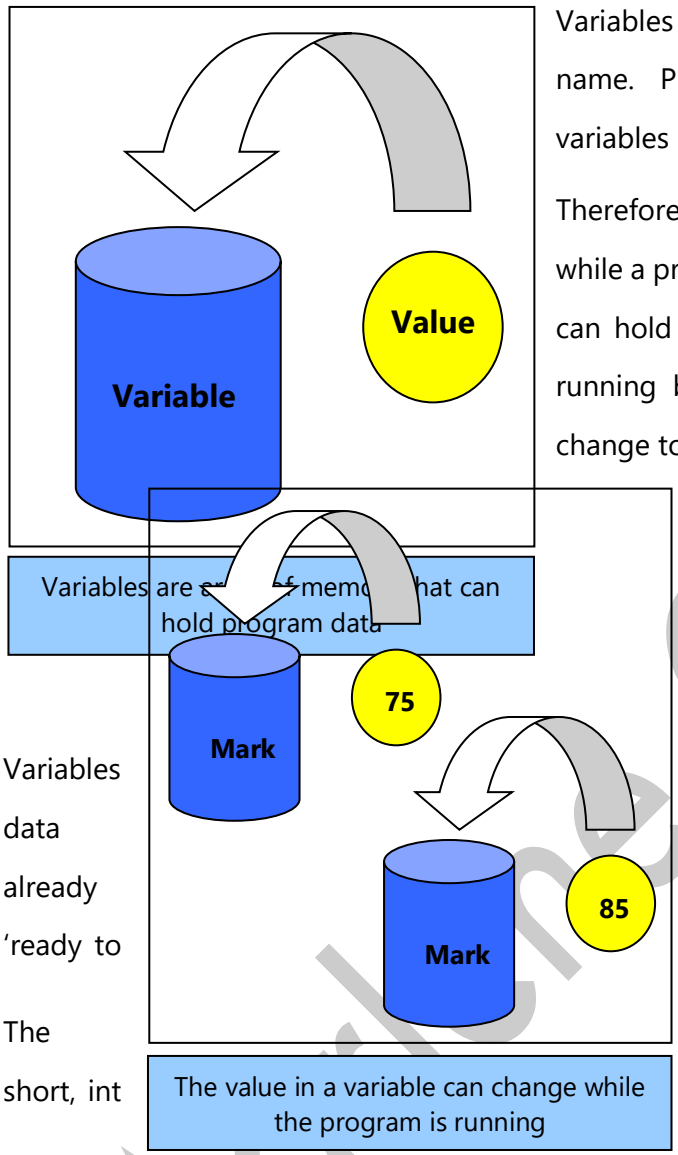
There are 3 types of commenting in Java:

- // text : the compiler ignores everything from // to the end of the line
- /*text*/ :the compiler ignores everything from /* to */
- /** documentation */ : can span over several lines as shown below:

```
/**This is a comment
* that spans more
* than one line
*/
```

Variables

What are variables?



Variables are areas of memory identified by a name. Program instructions can read data from variables or write data to them.

Therefore the contents of variables are changed while a program is running. For instance a variable can hold a null value (0) when a program starts running but later in the program its value can change to 5 or 7 etc.

Variables have different types, depending on the data they can hold. A primitive type is already defined by the language and therefore 'ready to use'.

Variable types – Primitive Types

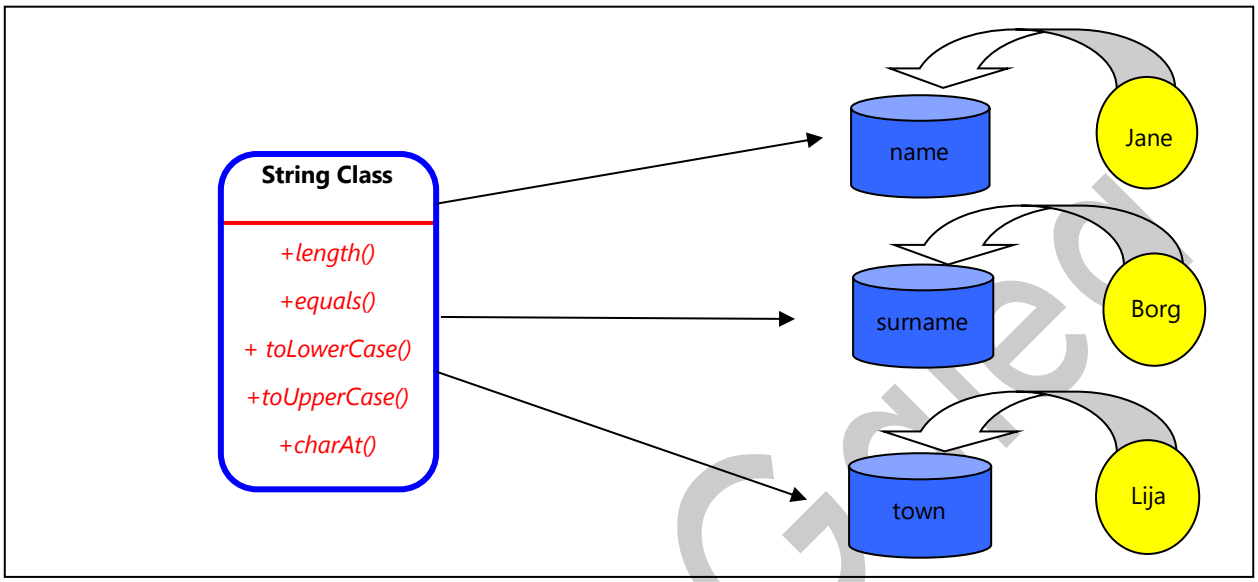
The names of these variable types (eg. Byte, short, int etc) are reserved keywords.

Variable type	Variable size	Accepted Data Range
byte	8 bits	-128 to 127
short	16 bits	-32,768 to 32,767
int	32 bits	-2,147,483,648 to -2,147,483,647
long	64 bits	-9,223,372,036,854,755,808 to 2^{63} - 1
float	32 bits	1.4e045 to 3.4e38
double	64 bits	4.9e324 to 1.8e308
char	16 bits	O to 65,535
boolean	1 bit	*True* or *false*

Variable types - Strings

Java allows us to use character strings because it has the 'java.lang.String' class. The String class is not really a primitive data type, but Java allows us to use it like a primitive type.

The String class



When we create a String we are creating an object of type String.

Enclosing strings within double quotes automatically creates a new String object; for example,

String s = "this is a string";

Variable names

Variable names must obey certain rules and conventions:

- Variable names must not be a Java keyword or reserved word

Java reserved words

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

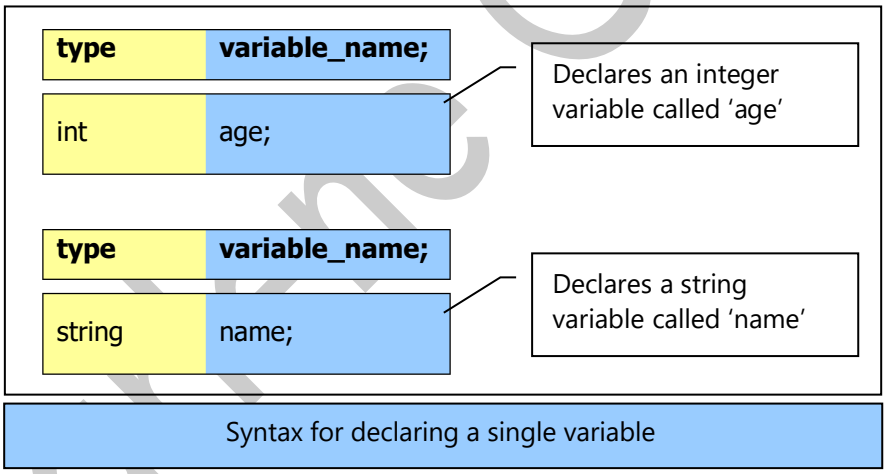
A list of Java Reserved Words and that therefore cannot be variable names

- Variable names must not be Boolean literal ('true'/'false') or 'null'
- Variable names must be unique within their scope.
    - E.g. you cannot have two variables of the same name declared at class level.
- Variable names must not contain blank spaces
    - E.g. The variable name 'client name' is not acceptable but 'client_name' is.
- Variable names cannot start with a number. They must start with a letter, underscore (_) or dollar sign ($).
    - E.g. '2num' is not acceptable but 'num2' is acceptable.

Declaring variables

Before a variable can be used, it needs to be declared. This means one needs to state its type and name.

Declaring a single variable:



Syntax for declaring a single variable

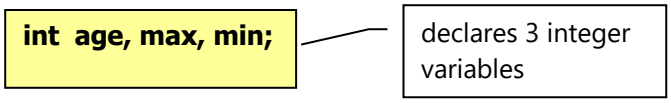
The following gives the output shown here:

```
public class declarevar1 {

    public static void main(String[] args) {
        String name;
        name = "Flower Shop";
        System.out.print("The shop name is" + " " + name) ;
    }
}
```

```
The shop name is Flower Shop
```

Declaring a single variable

Declaring a list of variables:



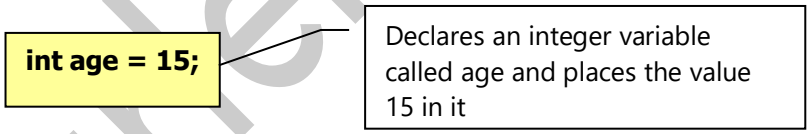
The following gives the output shown here:

```
public class declarevar1 {

    public static void main(String[] args) {
        String name, village;
        name = "Flower Shop";
        village = "Lija";
        System.out.println("The shop name is" + " " + name) ;
        System.out.print("Village:" + " " + village) ;
    }
}
```

```
The shop name is Flower Shop
Village: Lija
```

Declaring a list of variable

Declaring a variable and simultaneously putting a value in it:



The following gives the output shown here:

```
public class declarevar1 {

    public static void main(String[] args) {
    String name = "Flower Shop";
    System.out.print("The shop name is" + " " + name) ;
    }
}
```

```
The shop name is Flower Shop
Process completed.
```

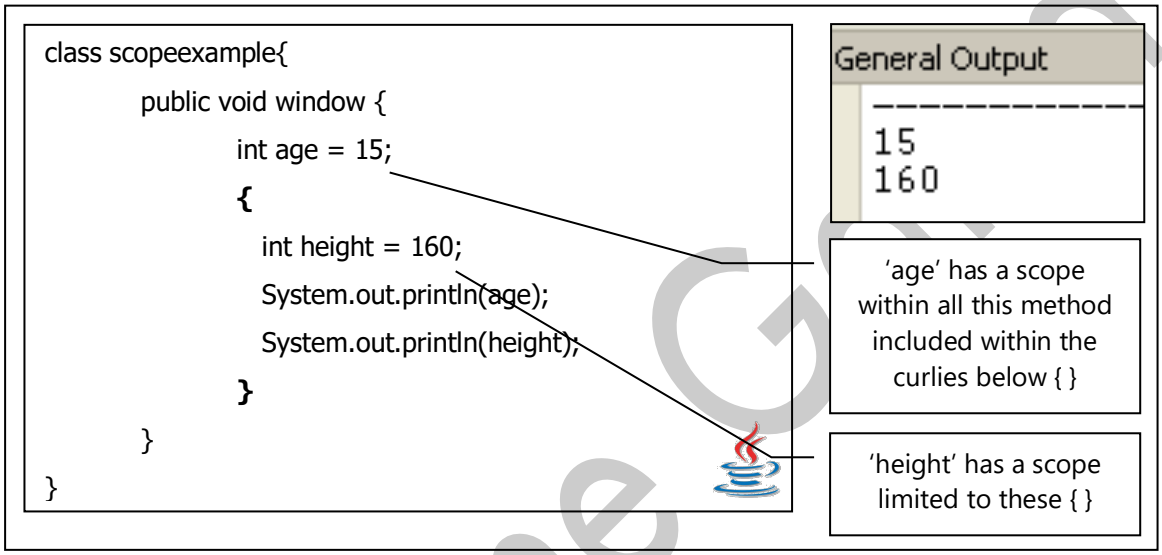
Declaring a variable and putting a value in it simultaneously

Scope of variables

A scope is created within a block of statements enclosed in curly brackets. A variable declared within curly brackets is only 'visible' within those curly brackets and cannot be used outside them.

1. Variables declared at class level and those declared enclosed in curly brackets can both be accessed within those curly brackets.

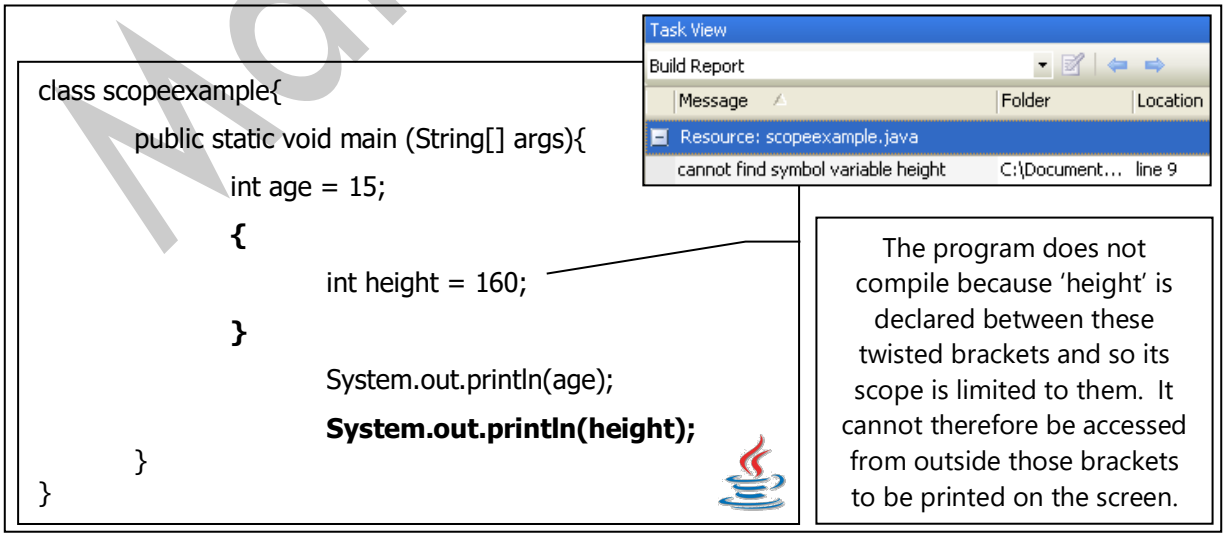
This program gives the output shown:



Scope of a variable

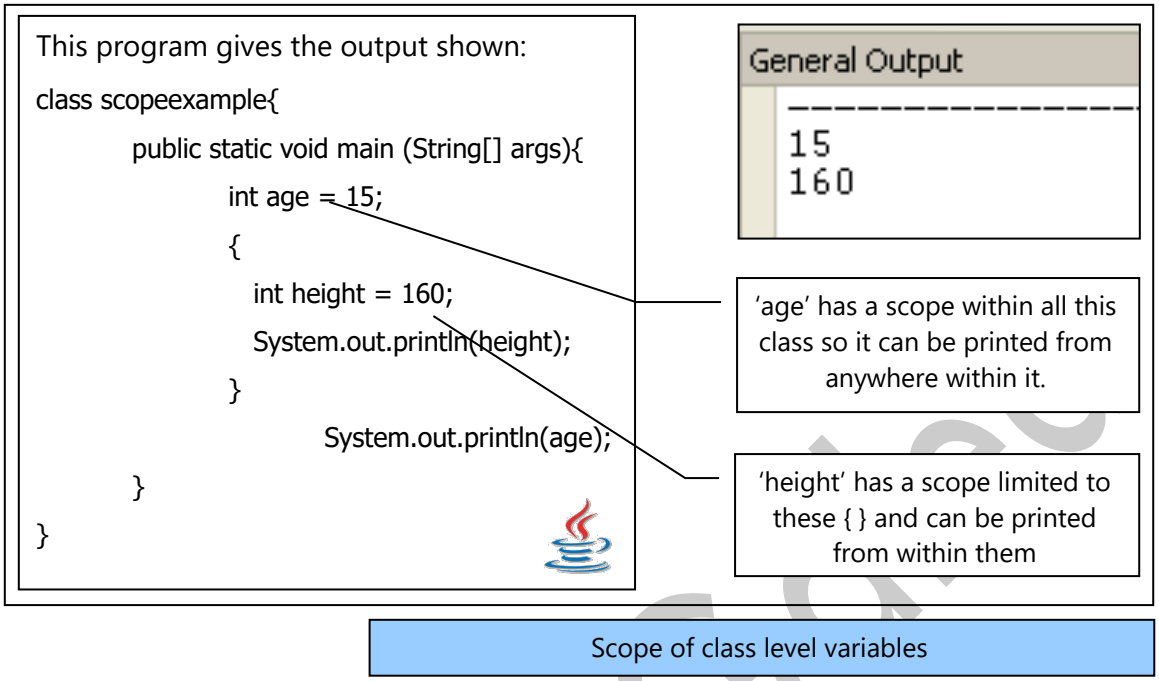
2. Variables declared within curly brackets can not be accessed outside those curly brackets

This program gives the error message shown:



Trying to access a variable from outside its scope

3. Variables declared enclosed in curly brackets can be accessed within those curly brackets (variables declared at class level can be accessed from anywhere within that class).

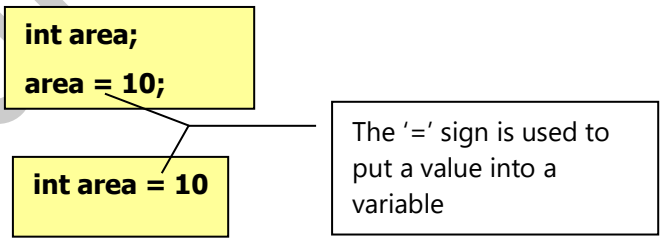


Variable Initialisation

Normally all variables are initialised to zero or null but one can always initialise a variable.

The compiler never assigns a default value to an uninitialized local variable. So assign it a value before using it. Accessing an uninitialized local variable gives a compile-time error.

To initialize a variable to a value we use the '=' sign. Both these examples declare the variable 'area' as an integer and initialize it to '10'.



Dynamic initialisation

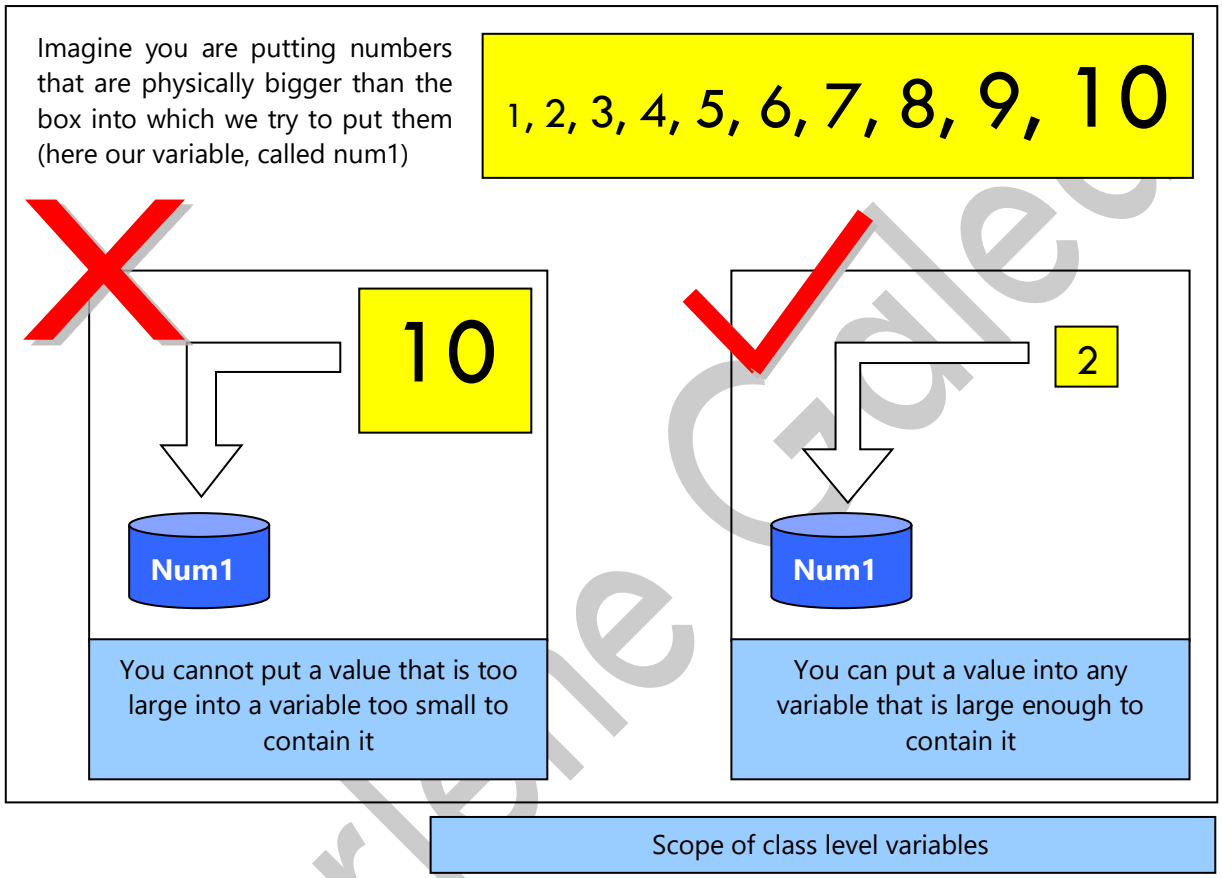
In Java variables can be initialised dynamically, this means that *a variable can be initialised at the time it is declared using a valid expression.*

E.g. here area is being initialised dynamically as the product of 'length' and 'breadth':

```
int area = length * breadth;
```

Variable types and value ranges

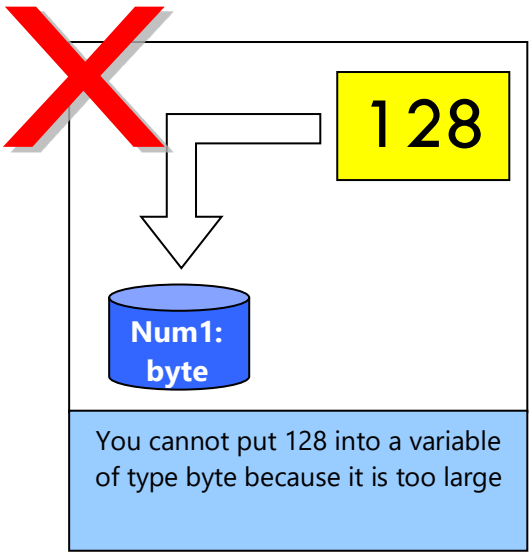
The relationship between a variable and its value is the same as that between a container and its contents. The variable and the value have to match in their *size* and *type*: you cannot put a value that is too large into a variable too small to contain it and the value type has to be acceptable for that variable type.



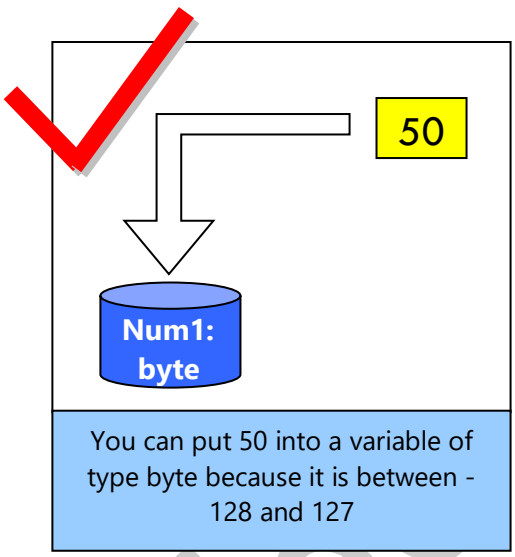
Each primitive data type has a fixed number of bits in which to accommodate values, hence each data type has its acceptable data range.

Variable type	Variable size	Accepted Data Range
byte	8 bits	-128 to 127
short	16 bits	-32,768 to 32,767
int	32 bits	-2,147,483,648 to -2,147,483,647
long	64 bits	-9,223,372,036,854,755,808 to 2^{63} - 1
float	32 bits	1.4e045 to 3.4e308
double	64 bits	4.9e324 to 1.8e308
char	16 bits	O to 65,535
boolean	1 bit	*True* or *false*

So for example the number 50 can fit into a variable of type 'byte' but the number 128 cannot



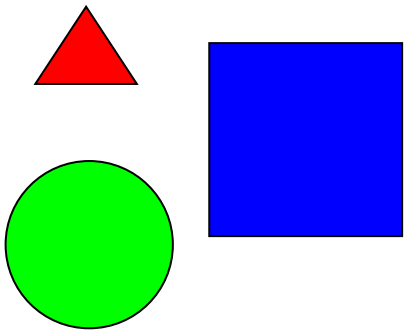
You cannot put 128 into a variable of type byte because it is too large



You can put 50 into a variable of type byte because it is between -128 and 127

Scope of class level variables

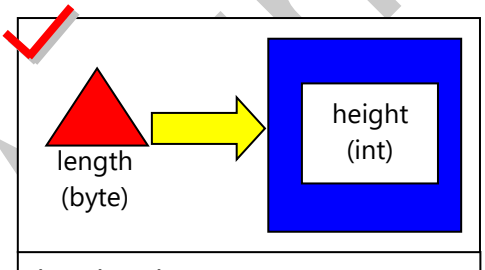
Automatic type conversion



The triangle, the circle and the square are different shapes.
However because one is smaller than the other, the triangle can easily be placed within the circle, and the square.
The circle can also easily be placed inside the square. But there's no way anyone could put this square into the triangle.

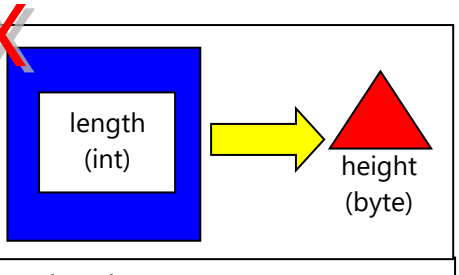
Automatic type conversion

Similarly it is possible to put the contents of smaller variables into larger variable types but not vice versa.



byte length = 5;
int height = length;

Putting a byte variable into a larger int variable is allowed



int length = 5;
byte height = length;

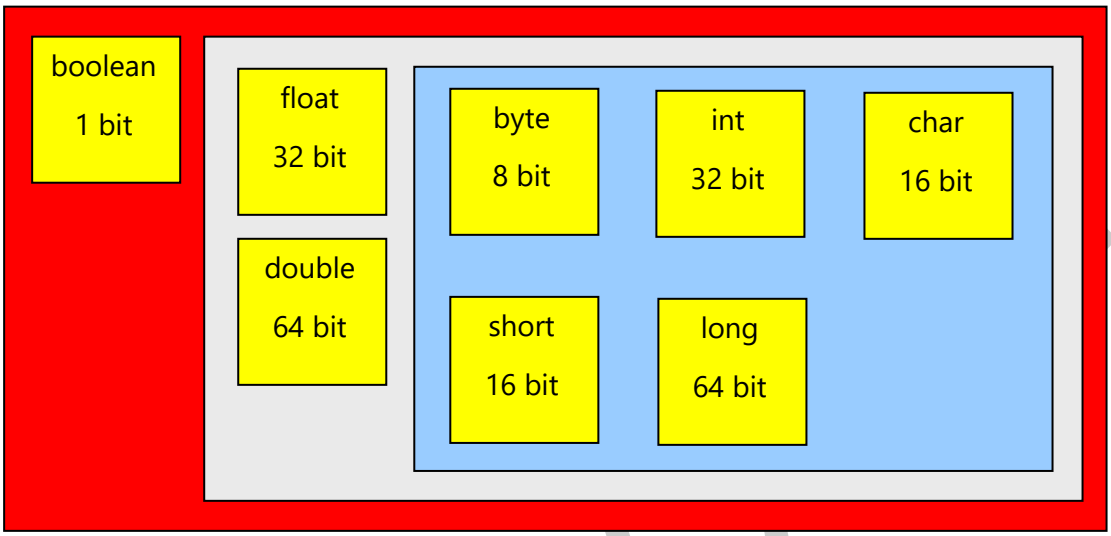
Putting an int variable into a smaller byte variable is not allowed

Automatic type conversion

Variable Type Compatibility Chart

The following chart summarises variable compatibility, that is, what variable types can be copied to which.

The inner box can fit into the outer box but not vice versa



Variable types in the inner boxes can fit in the outer boxes but not vice versa

Type casting

Java can automatically cast (change) from one type to another as long as the type fits into the above compatibility chart.

The syntax to typecast one variable into another is

```
double pi = 3.1415;
int x = (int) pi;
```

Forcing the data item to fit into an integer variable.

```
int pi = (int) 3.1415;
```

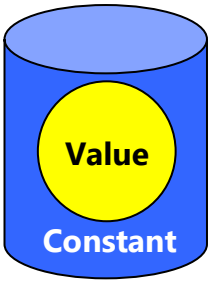
Forcing the data item to fit into an integer variable.

However the following does not fit in with the above compatibility chart and so is not correct and will give an error:

ERROR!

```
boolean pass = true;
int a = int(pass);
```

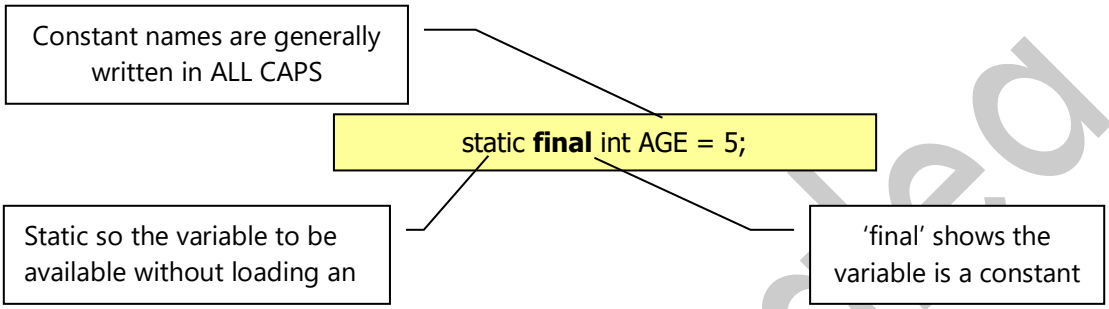
Final (Constant) Variables



Final Variables or Constants are variables whose value cannot be changed while the program is running.

Declaring a constant

In Java a constant is declared as a static and final variable



The following gives the output shown here:

```
public class multiply {
        static final int MULTIPLE = 5;
        public static void main (String args[]){
                System.out.println (1 * MULTIPLE);
                System.out.println (2 * MULTIPLE);
                System.out.println (3 * MULTIPLE);
                System.out.println (4 * MULTIPLE);
                System.out.println (5 * MULTIPLE);
                System.out.println (6 * MULTIPLE);
                System.out.println (7 * MULTIPLE);
                System.out.println (8 * MULTIPLE);
                System.out.println (9 * MULTIPLE);
                System.out.println (10 * MULTIPLE);
        }
}
```

```
General Output
--------------------Confi
5
10
15
20
25
30
35
40
45
50

Process completed.
```

Using a Constant

Had the program been written like this it would have given the same output:

```
class multiply {
        public static void main (String args[]){
            final int MULTIPLE = 5;
            System.out.println (1 * MULTIPLE);
            ...
        }
```

Keyboard input

Sometimes we need our programs to accept input from the user. The Java JDK includes a class called 'Scanner class' which has some very useful methods for inputting data through the keyboard.

1. Import Scanner Utility

In order to use this class we have to import it using the following line:

2. Create a Scanner object

To use the Scanner utility, we need to create an object using the following syntax:

Type	Variable name	=	new	type	();
Scanner	**input**	**=**	**new**	**Scanner**	**(System.in);**

To get user input we need to use the system input stream

3. Reading data from keyboard into variables

The table below explains how to read data into the different variable types using the scanner class.

Reading an integer	int a = (input.nextInt());
Reading a double variable	double a = (input.nextDouble());
Reading a float variable	float a = (input.nextFloat());
Reading a String variable	String a = (input.nextLine ());
Reading a short variable	short a = (input.nextShort());
Reading a byte variable	byte a = (input.nextByte());
Reading a long variable	long a = (input.nextLong());
Reading a Boolean variable	boolean a = (input.nextBoolean());
Reading a char variable Characters cannot be read directly so we read a string and take the first letter only.	String a = (input.nextLine ()); char b = a.charAt(0)l

Coding convention Rules

Naming classes, variables and constants

Programmers tend to stick to certain basic rules when naming their classes, variables and constants as well as when structuring their programs. Following code conventions make programs more understandable by making them easier to read. They can also give information about the function of the identifier; for example, whether it is a constant or a class. This can be helpful in understanding the code.

Identifier Type	Rules for Naming	Examples
Classes	Class names should be nouns in singular. If there's more than one word the first letter of each internal word is capitalized. Keep class names simple and descriptive.	class Test; class MathsTest;

Methods	Methods names should be verbs
	If there's more than one word the first letter is lowercase, with the first letter of each internal word capitalized.
		run();
		runFast();
		selectFromMenu();

Variables	Variable names should be short yet meaningful. The choice of a variable name should indicate the intent of its use. One-character variable names should be avoided.	int num; char letter; float myWidth;
Constants	The names constants should be all uppercase with words separated by underscores ("_").	final int MIN_WIDTH = 4; final int SPEED_LIMIT = 60;

Code blocks

Opening first curly bracket same line as construct and closing on a separate line.

```
class multiply {
        public static void main (String args[]){
            final int MULTIPLE = 5;
            System.out.println (1 * MULTIPLE);
            ...
        }
```

Arithmetic Operators

Basic Arithmetic

To do basic arithmetic in Java we use the following syntax:

```
c = a + b;
```

Work out a + b and store the answer in c

Adding a and b and storing answer in c.	**c = a + b;**
Subtracting b from a and storing answer in c.	**c = a - b;**
Multiplying a and b and storing answer in c.	**c = a * b;**
Dividing a by b and storing answer in c.	**c = a / b;**
Returns the remainder obtained after dividing a by b. This is called modulus.	**c = a % b;**

Basic Arithmetic in Java

Let's take another look at the class 'Window'

Basic arithmetic in an application

```
class Window{
        float length;
        float breadth;

        public Window(){
                length = 1;
                breadth = 2;
        }

        public float getArea(){
                float area = length * breadth;
                return area;
        }

        public float getPerimeter(){
                float perimeter = (length+breadth) * 2;
                return perimeter;
        }
}
```

Can also be written in two lines:

```
float area;
area = length * breadth;
```

Returns the area to the method that calls 'getArea()'

Here the backets ( ) are needed so that (length + breadth) is evaluated before the multiplication

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

                Window kitchenWin = new Window();

                System.out.println("Length: "+kitchenWin.length);
                System.out.println("Breadth: "+kitchenWin.breadth);
                System.out.println("Area: "+kitchenWin.getArea());
                System.out.println("Perimeter: "+kitchenWin.getPerimeter());
        }
}
```

Calls the constructor in class Window

Outputs the value returned by getArea for the object kitchenWin

Unary Operators

The unary operators require only one operand; they perform various operations such as incrementing/decrementing a value by one, negating an expression, or inverting the value of a boolean.

The first two below are examples of unary operators:

Operator	This is...	...equivalent to	Operation
++	**n++**	**n = n + 1**	Add 1
--	**n--**	**n = n - 1**	Subtract 1
+=	**n+=x**	**n = n + x**	Addition
-=	**n-=x**	**n = n - x**	Subtraction
*=	**n*=x**	**n = n * x**	Multiplication
/=	**n/=x**	**n = n / x**	Division
%=	**n%=x**	**n = n % x**	Remainder

The Math Class

The Math is one of the libraries or classes found in the JDK. It has a number of methods and the following are some of the more useful:

Method	Description
abs(int x)	Returns the absolute value of x
pow(int y, int x)	Returns y to the power of x.
sqrt(double x)	Returns the square root of x.
random()	Returns a pseudo random number between 0 and 1.
round(float x)	Returns x rounded up to the nearest integer.
ceil(double x)	Returns the smallest whole number greater than or equal to x.
floor(double x)	Returns the largest whole number less than or equal to x.

Whenever we use any of these methods we have to first include the class name 'Math' because these are static methods.

What are static methods?

Static methods use no instance variables of any object of the class they are defined in. Static methods typically take all the data from parameters and compute something from those parameters, with no reference to variables. This is typical of methods which do some kind of generic calculation. A good example of this are the many utility methods in the predefined Math class.

These both perform the same function and output '3.0'.

```
Syst                                    9));
```

Calls method 'sqrt' in class Math and passes it the value 9

```
int x = 9;
System.out.println (Math.sqrt(x));
```

```
class MathClass {
        public static void main (String args[]){
                int x = 9;
                System.out.println(Math.sqrt(x));
        }
}
```

The class name 'Math' has to be included in every instruction that makes use of a method in it

```
import static java.lang.Math.*;

class MathClass {

        public static void main (String args[]){
                int x = 9;
                System.out.println(sqrt(x));
        }
}
```

When we import all methods in the class (using .*), we can then use the methods in that class without including the name of the class (Math) every time.

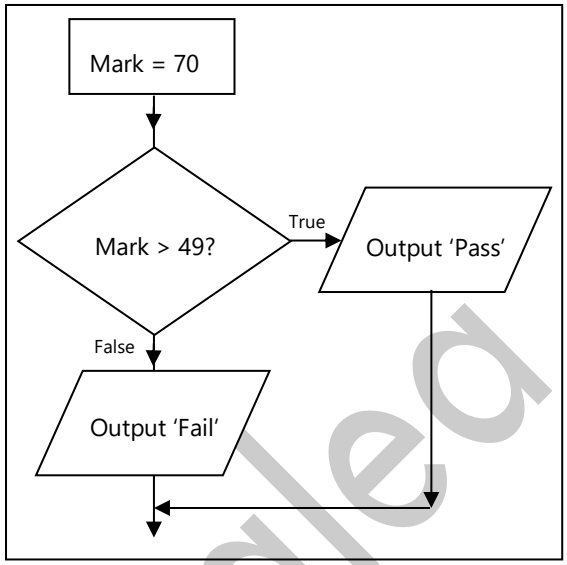
```
General Output
-------------------
3.0

Process completed.
```

Importing a class with all its methods

Conditional transfer: If, if-else and switch

Sometimes we need our program to do a different thing, depending on a condition. For instance in the example shown here we would like our program to output 'Pass' if mark is greater than 49 and otherwise output 'Fail'.

Therefore a programming language needs to allow us to create branching instructions in order to implement decisions.



Branching or decision instructions

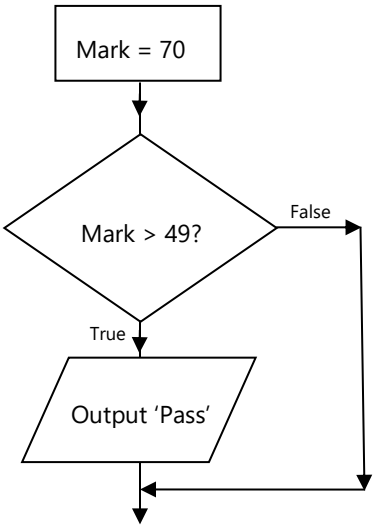
The if statement

The if statement is used when we need to route program execution through one of two paths: to take an action or to take no action.

if	(condition)	Statement/s;
if	(mark>49)	System.out.println ("Pass");

if	(condition)	Statement/s;

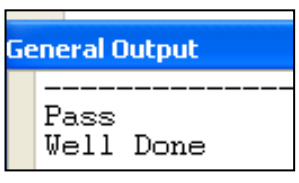
if	(mark>49)	{
		System.out.println ("Pass");
		System.out.println ("Well Done!");
		}



Basic Structure of the if statement

```
int mark = 70;
if (mark>49){
        System.out.println ("Pass");
        System.out.println ("Well Done");
}
```

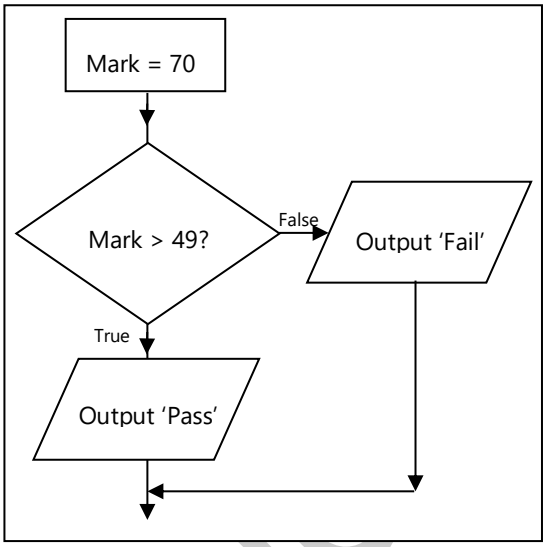
To execute 2 or more lines IF the condition is true we need to enclose these lines in { }



If statement example

The if-else statement

Very often we have a situation where if a condition is true we want to execute one set of instructions and if it is false we want to execute another set of instructions. This is where the if-else statement is used.



Flowchart for if-then else branching

Here we have the flowchart and code for representing the situation where if the mark is greater than 49 we output 'Pass' and if it's not we output 'Fail'

The table below shows that in order to execute 2 or more lines in a branching instruction, we need to enclose these lines in { }

if	(condition)	Statement/s;	else	Statement/s;
if	(mark>49)	System.out.println ("Pass");	else	System.out.println ("Fail");

if	(condition)	Statement/s;	else	Statement/s;
if	(mark>49)	**{** System.out.println ("Pass"); System.out.println ("Well Done!"); **}**	else	**{** System.out.println ("Fail"); System.out.println ("Poor!"); **}**

Basic Structure of the if-else statement

Therefore the generic structure for an if-else statement is as shown here:

```
If (condition) {
        statement/s
}
else {
        statement/s
}
```

Logical operators

Conditional statements, like the if statement jump to a section of code depending on the result of a condition. So far we have seen simple conditions (e.g. mark > 49). The following is a list of simple and compound logical expressions:

Operator	Meaning	Use	Explanation
!	unary not	**if (mark !=100)** **System.out.println("Aim Higher");**	Outputs 'Aim Higher' if the mark is not 100.
&&	and	**if (mark>49) && (mark<60)** **System.out.println ("Grade C");**	Outputs Grade C if the mark is between 50 and 59.
||	or	**if (name = "Ian") || (name = "ian"** **System.out.println ("Ian found");**	Outputs 'Ian found' if the name is 'ian' or 'Ian'.
==	equal to	**if (mark == 100)** **System.out.println ("Well Done");**	Outputs 'Well Done' if the mark is 100.
!=	not equal to	**if (mark != 0)** **System.out.println ("Not zero");**	Outputs 'Not zero' if the mark is not 0.
>	greater than	**if (mark>49)** **System.out.println("Pass");**	Outputs 'Pass' if mark is 50 or greater.
>=	greater or equal	**if (mark>=49)** **System.out.println ("Pass");**	Outputs 'Pass' if mark is 50 or greater.
<	smaller than	**If (mark<50)** **System.out.println ("Fail");**	Outputs 'Fail' if marks is less than 50 (49 or less).
<=	smaller or equal	**If (mark <=49)** **System.out.println ("Fail");**	Outputs 'Fail' if mark is less than 50 (49 or less).

Example

Let's say we have a program that randomly generates two integers a and b and outputs whether the first integer is a factor of the second.

Note: if it is a factor there will be no remainder when we divide b by a. So we will use modulus (c = b %a) here.

```
import static java.lang.Math.*;
class Factors{
        public static void main (String args[]){
            int c;
            int a = (int)(100.0 * random()) + 1;
            int b = (int)(10.0 * random()) + 1;
            c = a % b;
           if (c == 0) {
                System.out.println(b + " is a factor of " + a);
            }
            else {
                System.out.println(b + " is NOT a factor of " + a);
            }
        }
}
```

We import all methods in the Math class (using .*) so we can use its methods without including the name of the class (Math) every time. Here we will use the 'random' method.

This line generates a random number from 1 to 100, converts it to integer and stores it in integer variable a.

Divides a by b and stores the remainder in c

Outputs b is a factor of a if the remainder is 0

Outputs b is a factor of a if the remainder is not 0

Finding if one randomly generated number is a factor of a second randomly generated number,

Nested-if

A nested if statement is and if construct present inside the body of another if construct.

```
class NestedIf {
    public static void main (String args[]){
        int mark = 70;
        float average = 65;

        if (mark>50){
            if (average > mark){
                System.out.println ("Pass but below average");
            }
            else{
                System.out.println ("Pass. Mark above average");
            }
        }
        else{
                System.out.println ("Fail!");
        }
    }
}
```

General Output

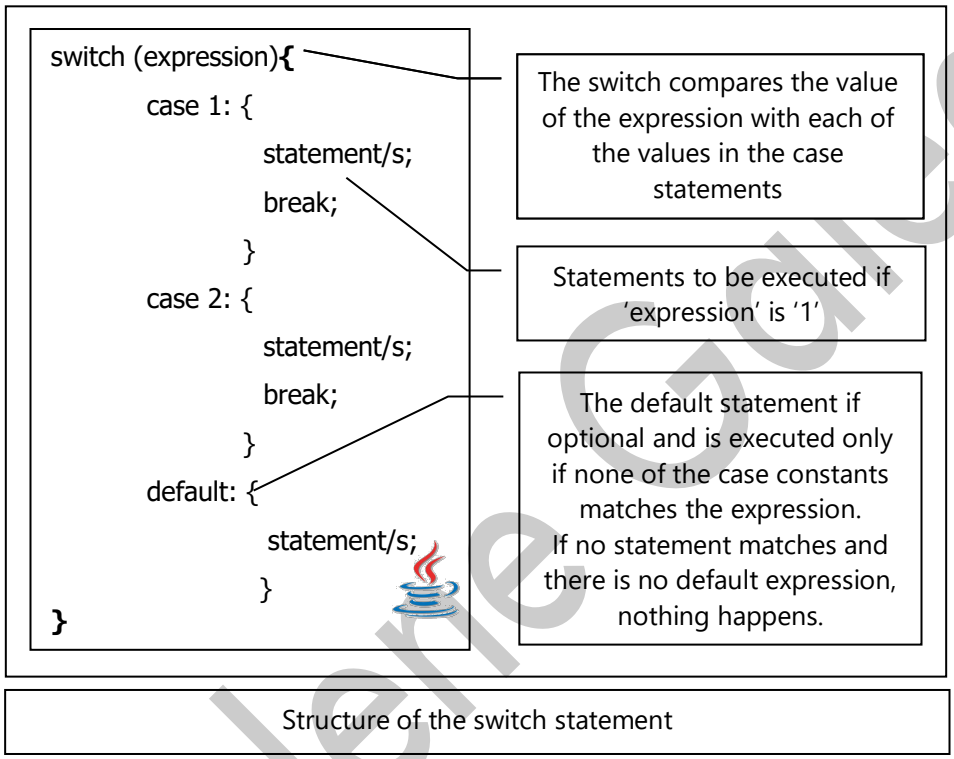
```
----------Configu
Pass. Mark above average

Process completed.
```

Nested-if

The Switch Statement

Sometimes our choice is not between two things as with 'Pass' and 'Fail'. Let's say we have a program menu that will take execution to a different part of a program depending on the user's menu choice. In this case it would be too cumbersome to implement the selection using ladders of if statements.

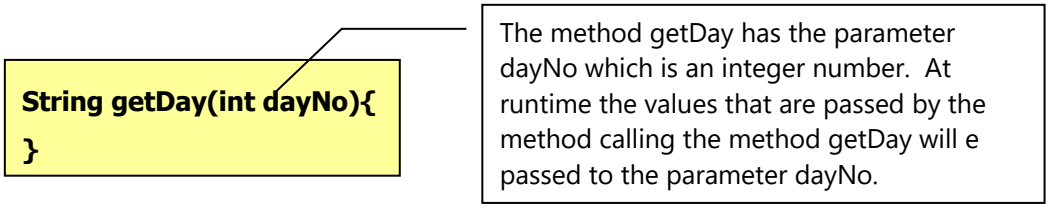
For such multi-way branching, the switch statement is better suited.



Structure of the switch statement

Using method parameters

The declaration for a method or a constructor declares the number and the type of the arguments for that method or constructor. The method getDay has one parameter dayNo which is an integer number. The parameters are used in the method body and at runtime will take on the values of the arguments that are passed to it by the method that calls the method getDay.



Parameters refers to the list of variables in a method declaration.

Arguments are the actual values that are passed in when the method is invoked. When you invoke a method, the arguments used must match the declaration's parameters in type and order.

Let's say we are dealing with a calendar application that has a method to convert the numbers 1 to 7 into the equivalent day of the week. This could be successfully implemented using a switch statement.

A switch statement being used to convert the numbers 1 to 7 into the equivalent dy of the week.
The method getDay receives a value for dayNo from the method that calls it.

```
public class DayOfWeek{

    int hours;
    String day;

    String getDay(int dayNo){
        switch (dayNo){
            case 1: {
                        day = "Monday";
                        break;
                    }
            case 2: {
                        day = "Tuesday";
                        break;
                    }
            case 3: {
                        day = "Wednesday";
                        break;
                    }
            case 4: {
                        day = "Thursday";
                        break;
                    }
            case 5: {
                        day = "Friday";
                        break;
                    }
            case 6: {
                        day = "Saturday";
                        break;
                    }
            case 7: {
                        day = "Sunday";
                        break;
                    }
            default: { day = "Invalid day";}
        }
        return day;

    }
}
```

```
General Output
--------------------(
Today it is Friday

Process completed.
```

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

        DayOfWeek today = new DayOfWeek();

        int dayNo = 5;

        today.getDay(dayNo);
        System.out.println ("Today it is " + today.day);
    }
}
```

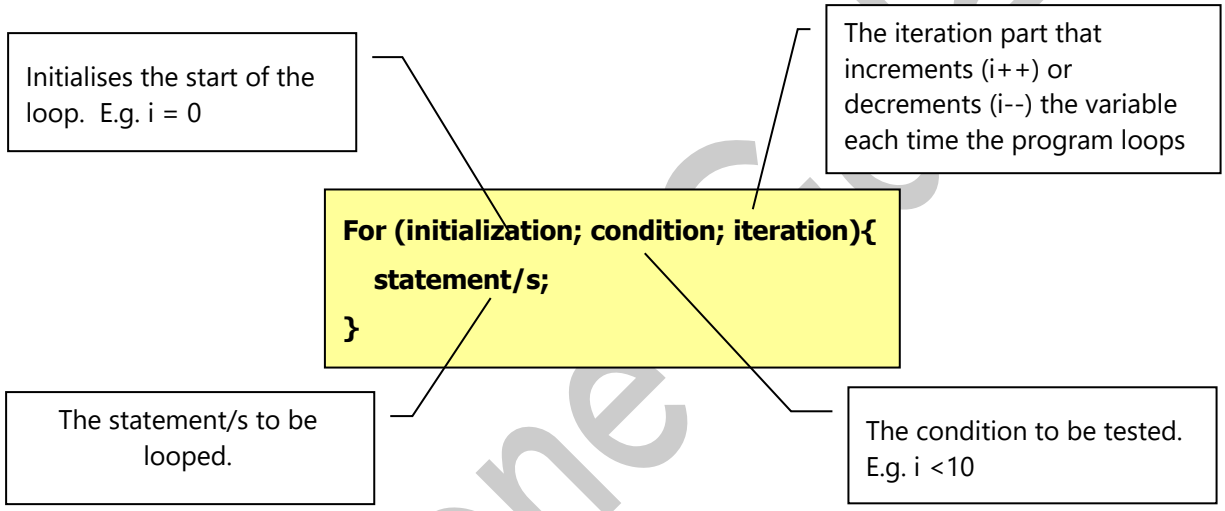
Loops

Sometimes we have one or more instructions that we would like the system to execute a number of times, usually until a condition is met or until they have been executed a given number of times. This is implemented using a loop.

Java offers us the following three looping constructs: for, while, do-while

The for loop

The for loop is used when, before we start executing the loop, we know how many times we want to repeat the loop.



for	(	initialization	;	condition	;	iteration	)
for	(	i = 0	;	i<10	;	i++	)

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

    int x;

                for (x=1; x<=10; x++)
                    System.out.println("x=" +x);

    }
}
```

When only 1 statement is being looped, { } are not needed.

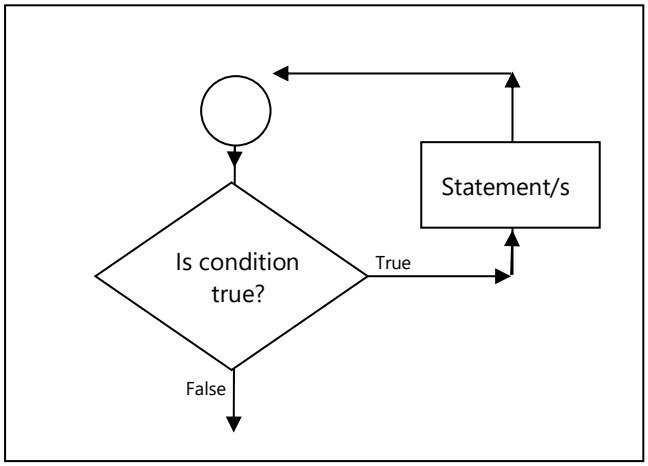
```
General Output
x=1
x=2
x=3
x=4
x=5
x=6
x=7
x=8
x=9
x=10

Process completed.
```

Using a for loop to output x = 1..x =10

While and do..while

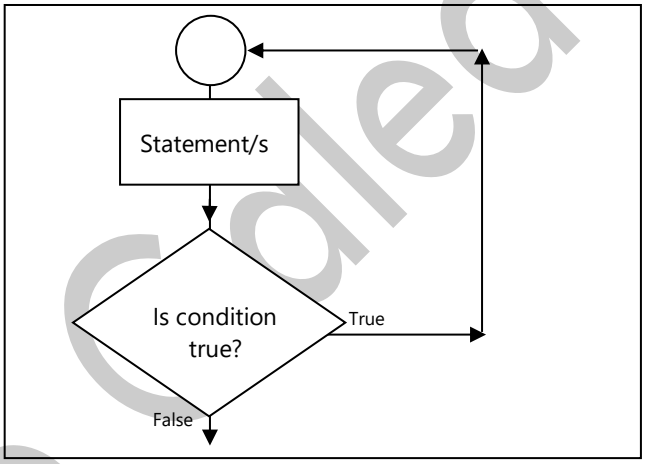
The while and do-while loops are both conditional loops: this means they loop until a particular condition is met. When a program is being created it is not always clear how many times the loop will need to be executed. Think for instance of a 'guessing game': the user may guess immediately, but he may guess after two tries or after a 100 tries…so the looping statement in such a game needs to be implemented using an indeterminate loop.

The While Loop.



- The While Loop, checks the condition before executing looping statement/s.
- If the condition is immediately satisfied the looping instructions won't be executed at all.
- The while loop is **executed 0 or more times**.

The Do..While Loop.



- The do..while Loop, checks the condition after executing the looping statement/s.
- If the condition is immediately satisfied the looping instructions still execute at least once.
- The do..while loop is **executed 1 or more times**.

```
import java.util.Scanner;

class WhileExample{

    public static void main (String args[]){
      Scanner input = new Scanner (System.in);
      int num = 7;

      System.out.print ("Enter your guess: ");
      int guess = (input.nextInt());

      while (guess != num){
        System.out.println();
        System.out.println ("Wrong guess. Retry");
        System.out.print ("Re-enter guess: ");
        guess = (input.nextInt());
      }

    System.out.println ("Correct guess");
        }
}
```

```
import java.util.Scanner;

class Menu{

    public static void main (String args[]){
      Scanner input = new Scanner (System.in);
      int choice;

      do {
        System.out.println ("MENU");
        System.out.println ("1. Enter Students
                  Details");
        System.out.println ("2. View Student
                  Details");
        System.out.println ("3. Exit");
        System.out.print   ("Enter choice:");

         choice = (input.nextInt());
      } while (choice!=6);
   }
}
```

Take a closer look at the syntax of these two loops:

```
while (guess != num){
        System.out.println();
        System.out.println ("Wrong guess. Retry");
        System.out.print ("Re-enter guess: ");
        guess = (input.nextInt());
      }
```

If the guess entered is not equal to the value of num the loop will be executed again.

These lines are executed BEFORE the condition

```
do {
        System.out.println ("MENU");
        System.out.println ("1. Enter Students
                Details");
        System.out.println ("2. View Student
                Details");
        System.out.println ("3. Exit");
        System.out.print   ("Enter choice:");

        choice = (input.nextInt());
    } while (choice!=6);
```

Nested Loops

A nested loop is a loop within another loop.

The following example shows the use of a nested for loop to create the output patterns shown here.

```
class NestedLoop{
        public static void main (String args[]){
                int i;
                int x;

                for (i = 1; i < 6 ; i++){
                        for (x = 1; x <= i; x++){
                                System.out.print ("*");
                        }
                        System.out.println ();
                }
        }
}
```

```
General Output
--------------------Cor
*
**
***
****
*****

Process completed.
```

A nested loop: a for loop inside another

Arrays

An array is a group of related data items (variables) of the same type that are referred to with a common name.

Let's say the following is an array of 5 marks: 70, 85, 65, 80 and 75

MarkList	Index (door number)	0	1	2	3	4
	Element (item in the house)	70	85	65	80	75

To refer to a particular element in the array we need to use the array's name and that element's index: Marks[4] contains the number 75.

Using arrays

Declaring an array

The syntax for declaring an array in Java is the following:

```
int[ ] markList;
```

Creates an array variable called marksList that will hold integers only.

Assigning an array

```
markList = new int [5];
```

Creates an array and assigns it to array variable

Declaring and assigning an array in one line:

array-variable		=	new	type	[size];
int[ ]	markList	=	new	int	[5];

Using array variables

Outputting a single item in an array:

Given the array shown above, the following line outputs '80'.

```
System.out.println (markList[3]);
```

Inputting a single item into an array

```
markList[3] = (input.nextInt());
```

Using an array in a for loop

For loops are ideal for manipulating arrays as shown in the example below:

```
class RunMarks{
        public static void main (String args [] ){
                TestMarks MathsTest = new TestMarks();
                MathsTest.enterMarks();
                MathsTest.getAverage();
                }
}
```

The Main program will call the method enterMarks and then getAverage so the array will first be filled with values in enterMarks and then their average is calculated and output in getAverage.

```
import java.util.Scanner;

public class TestMarks{


        Scanner input = new Scanner (System.in);


        int[] mark = new int[5];
        int i;


        public void enterMarks(){
                for (i=0;i<5;i++){
                        System.out.print ("Enter mark: ");
                        this.mark[i] = (input.nextInt());
                }
        }


        public void getAverage(){
        int t = 0;
        double average;


                for (i=0;i<5;i++){
                        t = t + mark[i];
                }
                average = t/i;
                System.out.println ("The average is: " + average);
        }
}
```

```
General Output
--------------------C
Enter mark: 75
Enter mark: 70
Enter mark: 80
Enter mark: 85
Enter mark: 70
The average is: 76.0

Process completed.
```

Using a Third Party Class: The Keyboard Class

The Keyboard is a third party class used for data input. This is the listing of the Keyboard class:

```
import java.io.*;

public class Keyboard{

	public static String readString(){
		BufferedReader br;
		try{
			br = new BufferedReader(new InputStreamReader(System.in));
			return br.readLine();
		}catch (Exception e){

		}
		return null;
	}

	public static int readInt(){
		return Integer.parseInt(readString());
	}

	public static byte readByte(){
		return Byte.parseByte(readString());
	}

	public static short readShort(){
		return Short.parseShort(readString());
	}

	public static long readLong(){
		return Long.parseLong(readString());
	}

	public static float readFloat(){
		return Float.parseFloat(readString());
	}

	public static double readDouble(){
		return Double.parseDouble(readString());
	}

	public static char readChar(){
		return readString().charAt(0);
	}

	public static boolean readBoolean(){
		return Boolean.parseBoolean(readString());
	}


}
```

However, as we've already seen, the great thing about Java is that we do not need to know exactly how a class works as long as we know how to use it. In order to input data using the keyboard class we use the following syntax:

Using the keyboard class

Syntax	Use
String name = Keyboard.readString();	Reads a string from the keyboard and stores it in variable name
int num1 = Keyboard.readInt();	Reads an integer from the keyboard and stores it in variable num1.
byte num2 = Keyboard.readByte();	Reads a byte variable and stores it in variably num2.
short num3 = Keyboard.readShort();	Reads a short value from the keyboard and stores it in variable num3.
long num4 = Keyboard.readLong();	Reads a long value from the keyboard and stores it in variably num4.
float num5 = Keyboard.readFloat();	Reads a float from the keyboard and stores it in variable num5.
double num6 = Keyboard.readDouble();	Reads a double from the keyboard and stores it in variable num6.
char ch = Keyboard.readChar();	Reads a character from the keyboard and stores it in variable ch.
boolean tf = Keyboard.readBoolean();	Reads a Boolean value from the keyboard and stores it in variable tf.

This table explains the use of the Keyboard Class to input primitive variable types and strings. Therefore the keyboard class can be used instead of the Scanner class in some of our programs.

Appendix 1: Using third party classes in LeJOS

LeJOS is a firmware replacement for Lego Mindstorms programmable bricks. It includes a Java virtual machine, which allows Lego Mindstorms robots to be programmed in the Java programming language.

Simple LeJOS features

Feature	Example	Function
Import lejos.nxt.*	import lejos.nxt.*;	Will allow us to use the LeJOS features
Using Sensors		
UltrasonicSensor(SensorPort port) assigning a new ultrasonic object	UltrasonicSensor us = new UltrasonicSensor (SensorPort.S1)	Creates 'us' a new instance of Ultrasonic
getDistance()	us.getDistance()	Returns the distance to the nearest object
LCD.drawString("Moving", 0, 0);	LCD.drawString("Moving", 0, 0);	Display on screen
Timer class		
Timer.sleep (250);	Timer.sleep (250);	Using the sleep method in the timer class to create a delay. (passing 250 as a parameter, hence delaying by 250 milliseconds).
Motor class		
regulateSpeed(boolean yes)	Motor.B.regulateSpeed(true);	Regulates motor speed (1-900)
setSpeed(int speed),	Motor.B.setSpeed(500);	Sets the motor speed
forward()	Motor.B.forward();	Rotates the motor forward
backward()	Motor.B.backward();	Rotates the motor backward
stop(),	Motor.B.stop();	Stops the motor

Here is an example LeJOS programme:

```
import lejos.nxt.*;

public class MyProg {
    public static void main (String args []) throws Exception{
        int distanceToNearestObject = 500;
        UltrasonicSensor us = new UltrasonicSensor (SensorPort.S1);

        LCD.drawString("Moving", 0, 0);
        Timer.sleep (250);

        Motor.B.regulateSpeed(true);
        Motor.C.regulateSpeed(true);
        Motor.B.setSpeed(500);
        Motor.C.setSpeed(500);

        Motor.B.forward();
        Motor.C.forward();

        while ((distanceToNearestObject = us.getDistance()) >= 20){
            Motor.B.stop();
            Motor.C.stop();
        }
        Motor.B.backward();
        Motor.C.backward();
        Timer.sleep (50);
  }

}
```

A simple LeJOS program

This program causes the NXT to:

First output 'Moving' on its LCD;

Then move forward with a speed of 500 until the ultrasonic sensor detects an object less than 20 cm away.;

At this point it stops and then moves back for a short while.

Appendix 2 – The String class

Useful methods in java.lang.String

Method	Explanation
String.length(); Returns an integer	returns the number of characters in this string
String.toLowerCase() Returns a String	returns a new string with all characters converted to lowercase
String.toUpperCase() Returns a String	returns a new string with all characters converted to uppercase
String.equals(String); Returns boolean	Returns true if string is equal to another string
String.charAt(index:int); Returns a String	returns the character at the specified index from this string
`String substring(int beginIndex`	`int endIndex)`
`To concat 2 strings`	

```
Password guessed? true
Password length is 5
The FOURTH letter in the password is o
This is the password in capital letters HELLO
This is the password in small letters hello
```

```
class String_Functions{
    public static void main (String args [ ]){
        String password = ("Hello");
        String pword = ("Hello");
        boolean guessed;
        guessed = password.equals(pword);
        System.out.println ("Password guessed? " + guessed);
        int letters = password.length();
        System.out.println ("Password length is " + letters);
        char character = password.charAt(4);
        System.out.println ("The FOURTH letter in the password is " + character);
        String capitals = password.toUpperCase();
        System.out.println ("This is the password in capital letters " + capitals);
        String smalls = password.toLowerCase();
        System.out.println ("This is the password in small letters " + smalls);
    }
}
```

Using String class methods in an application tackling password protected user accounts

```
import java.util.Scanner;

public class Account{
	//Properties
	String name;
	String surname;
	String password;
	String username;
	boolean guessed;

	//Methods
	Scanner Input = new Scanner (System.in);

	public Account(){
		this.guessed = false;
	}

	public void changePassword(){
		System.out.print ("Enter New Password:");
		this.password = Input.next();
		System.out.print ("ConfirmPassword:");
		String pword = Input.next();
		if (this.password.equals(pword)){
			System.out.println ("Password Changed");
		}
		else{
			System.out.println ("Password does not match");
		}
	}

	public void guessPassword(){
		System.out.print ("Enter Password:");
		String userPword = Input.next();
		if (this.password.equals(userPword)){
			System.out.println ("Password Correct");
			this.guessed = true;
		}
		else{
			System.out.println ("Password Incorrect");
		}
	}

	public void newAccount(){
		System.out.print ("Enter name:");
		this.name = Input.next();
		System.out.print ("Enter surname:");
		this.name = Input.next();

		//generating username

		String subname = this.name.substring(1, 3);
		String subsurname = this.surname.substring(1,3);
		this.username = subname + subsurname;

		System.out.println ("Your new Username is" + this.username);
		System.out.println ("Your new Password is \'1234\'");
	}

}
```

Appendix 3 – Simple GUI programs

Java supports graphics to enhance the looks of our applications. Here we will look at two very simple ways you can include graphics in your application.

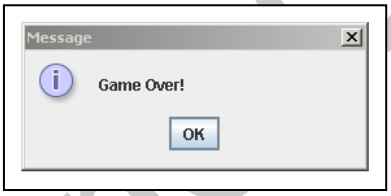
To do this we have to import the class JOptionPane using the line below:

```
import javax.swing.JOptionPane;
```

Displaying text in a Dialog Box

The syntax to display 'Game Over' in a Dialog Box is:

```
JOptionPane.showMessageDialog(null, "Game Over");
```



The code below displays the following dialog box:

```
import javax.swing.JOptionPane;

class Graphics{
        public static void main (String args[]){
                JOptionPane.showMessageDialog(null,
                        "Game Over!\n" +
                         "Press 'Enter' to play again");
        }
}
```

Escape character **\n** used to display text on multiple lines

Displaying multiple lines in a Dialog Bix

Entering text in a Dialog Box

The syntax to display 'Enter your name' in a dialog box and then read the name into a string variable is:

```
String name = JOptionPane.showInputDialog("Enter your name");
```

The variable in which the entered name will be placed.

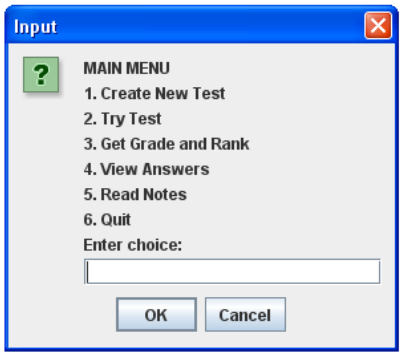
The code below displays the following dialog box:

```
public void mainMenu () throws Exception{
    String choices;

        choices = JOptionPane.showInputDialog(
                "MAIN MENU\n"+
                "1. Create New Test\n"+
                "2. Try Test\n"+
                "3. Get Grade and Rank\n"+
                "4. View Answers\n"      +
                "5. Read Notes\n"        +
                "6. Quit\n"+
                "Enter choice: \n");

        int choice = Integer.parseInt(choices);

        switch (choice) {
                case 1: enterQuestions();
                        saveTest();
                        break;
                case 2: tryTest();
                        break;
                case 3: processMark();
                        break;
                case 4: viewAnswers();
                        break;
                case 5: readNotes();
                        break;
        }
    }
```

Escape character **\n** used to display text on multiple lines

Integer.parseInt( ) is a method that converts Strings (here the variable choices) to integer

Note the use of a switch statement to deal with a menu choice



A GUI Main Menu

Appendix 4: Array of Objects

Java allows us to use an array to store objects.

Declaring an array of objects

Using an array of objects

We will now use an array of Person objects.

This is the Class Person that we used earlier on. It now includes two methods:

- *inPerson()* – reads data into the object variables of the object passed to it
- *outputPerson()* – outputs the data in the object variables of the object passed to it.

We will therefore make an array of instances of this class and have our array of objects use these methods.

Note the use of the 'this' keyword

```
import java.util.Scanner;

class Person {
   String name;
   String surname;
   String address;
   String telNum;

   public void inPerson(){
      Scanner input = new Scanner (System.in);

            System.out.print ("Name: ");
            this.name = input.nextLine();
            System.out.print ("Surname: ");
            this.surname = input.nextLine();
            System.out.print ("Address: ");
            this.address = input.nextLine();
            System.out.print ("Tel. No.: ");
            this.telNum = input.nextLine();
   }

   public void outputPerson(){
            System.out.println (this.name);
            System.out.println (this.surname);
            System.out.println (this.address);
            System.out.println (this.telNum);

   }
   }
}
```

The Person Class

Now let's take a look at the main method below which produces the output shown:

```
import java.util.Scanner;

public class School_System{

   public static void main (String args[]){

     Scanner input = new Scanner (System.in);
     System.out.print ("Enter number of new Cleaners: ");
     int y = input.nextInt();

     Person [] CleanerList = new Person[y];

     for (int i = 0; i<y; i++){
       CleanerList[i] = new Person ();
       System.out.println();
       CleanerList[i].inPerson();
      }

      System.out.println();
      System.out.println();

      System.out.println ("List of Cleaners");
      for (int i = 0; i<y; i++){
        CleanerList[i].outputPerson();
        System.out.println(  );
     }
  }
}
```

The number of elements in the array is being determined by the user.

The for loop that will help us go through the array. Remember that the first array element is number 0.

Creating the object CleanerList[i]

Calling the method inPerson( ) for the object CleanerList[i]

Using a for loop to go through the array, this time calling the method outputPerson() for each element in the array.

```
Enter number of new Cleaners: 2

Name: Maria
Surname: Borg
Address: 25, Main Street, B'Kara
Tel. No.: 21 443 554

Name: Jane
Surname: Abela
Address: 87, Well Street, Zejtun
Tel. No.: 21 553 576


List of Cleaners
Maria
Borg
25, Main Street, B'Kara
21 443 554

Jane
Abela
87, Well Street, Zejtun
21 553 576
```


Using an array of objects

Appendix 5: Text Files

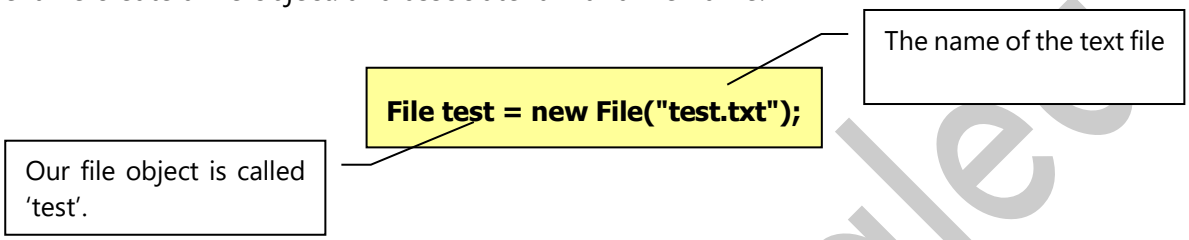
Java can read lines of text from a file and write lines of text to a text file.

Creating a text file

To import the necessary reading and writing classes we use:

```
import java.io.*;
```

Next we create a file object: and associate it with a file name:



Saving to a text file

When we save to a text file we use the keyword 'write'.

Initialising an object (writer) to write lines to a text file:

```
BufferedWriter writer = new BufferedWriter(new FileWriter(test, true));
```

Writing a line of text to a text file**:**

```
writer.write(Test[i].question);
writer.newLine();
```

Getting data from a text file

When we get data from a text file we use the keyword 'read'.

Initialising an object (reader) to read lines from a text file:

```
BufferedReader reader = new BufferedReader(new FileReader(test));
```

Reading a line of text from a text file**:**

```
Test[i].question=(reader.readLine());
```

Exception handling

When using text files, it is possible that the file we are trying to read from or write to is not found (e.g. if it is saved on a Pen Drive which is not available right then). This would cause our program to throw an input/output exception. Therefore we include 'throws IOException' in the signature of the method that will call methods in the io class. The compiler will complain if we don't do this.

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

CAL Quiz Example

We will now look at a cimple Quiz application that makes use of the class **Question** shown here.

```
import java.util.Scanner;

public class Question{
   String question;
   String answer;
   Scanner input = new Scanner(System.in);

   public void enterQuestion(){
         System.out.print ("Enter question:");
         this.question = input.nextLine();
         System.out.print ("Enter answer:");
         this.answer = input.nextLine();
    }
   public boolean askQuestion(){
         System.out.println (this.question);
         System.out.print ("Answer: ");
         String userAnswer = input.nextLine();
         if (userAnswer.equals(this.answer)){
            return true;
         }
         else{
            return false;
         }
    }
}
```

Method enterQuestion( ) is used to enter a question and correct answer for each Question object

Object,askQuestion() returns 'true' if the user's answer is correct, otherwise it returns 'false'

The class Question will be used to create instances of questions. Each of these objects will have a question and an answer and will have two methods 'enterQuestion( ) and askQuestion( )

```
import java.io.*;
import java.util.Scanner;

class TextFiles{

  public static void main (String args[])throws IOException {
     Scanner input = new Scanner(System.in);
     Question[] Test = new Question[5];
     File test = new File("test.txt");
     int choice = 0;

     do{
          System.out.println ("MAIN MENU");
          System.out.println ("1.  Enter Test Questions");
          System.out.println ("2.  Try Test");
          System.out.println ("3.  Exit");
          System.out.print ("Enter choice:");
          choice = input.nextInt();

           switch (choice){
             case 1:{
               for (int i=0;i<5;i++){
                    System.out.println();
                    Test[i] = new Question ();
                    Test[i].enterQuestion();
                    System.out.println();
                }
               BufferedWriter writer = new BufferedWriter(new FileWriter(test, true));
               for (int i = 0; i<5; i++){
                    writer.write(Test[i].question);
                    writer.newLine();
                    writer.write(Test[i].answer);
                     writer.newLine();
                }
               writer.close();
               break;
             }
```

Calling the method enterQuestion for Test[i]

Writes the contents of the array into the text filel, 1 element at a time.

It is important to close the file

[This method is continued overleaf...]

```
case 2:{
            int mark = 0;
            BufferedReader reader = new BufferedReader(new FileReader(test));
            for (int i = 0; i<5; i++){
             Test[i] = new Question ();
             Test[i].question=(reader.readLine());
             Test[i].answer=(reader.readLine());

             System.out.println();
             if (Test[i].askQuestion()){
                mark = mark + 1;
              }
           }
           reader.close();
           System.out.println ("Your mark is:" + mark + "/5");
           System.out.println();
           break;
         }

         case 3:{
            System.out.println();
            System.out.println ("Exiting");
            break;
         }
      }
  } while (choice != 3);
 }
}
```

Reads the questions and answers from the text file into the array Test, one element at a time

It is important to close the file

Repeatedly displays the Main Menu for the user to choose what to do next, until the choice is 3 (Exit)

A CAL Quiz Application using text files

When the user runs the above application:

- The Main Menu is displayed allowing the user to enter test questions, try the test or exit the application.
- When the user enters test questions these are entered into the array and then saved onto the text file

- When the user tries the test, the questions and answers are copied from the text file and into the array Test [ ], then the method askQuestion( ) is called for each element in the array.

The output given by the above application is shown here: