



Brunel University  
Department for Information Systems  
and Computing  
<http://www.brunel.ac.uk/depts/cs>

Java Primer Course

Simon J.E. Taylor & Raj Sudra

# Table of Contents

<b>PREFACE</b>	<b>4</b>
<b>ABOUT THIS DOCUMENT</b>	<b>4</b>
<b>PROGRAM EXERCISES</b>	<b>4</b>
<b>LOGGING ON TO THE NETWORK</b>	<b>4</b>
<b>COMMAND PROMPT</b>	<b>4</b>
<b>WORKING AREA</b>	<b>4</b>
<b>TEXT EDITOR</b>	<b>5</b>
<b>1. INTRODUCTION</b>	<b>6</b>
<b>1.1 JAVA</b>	<b>6</b>
<b>1.2 THE HELLO WORLD PROGRAM</b>	<b>6</b>
TRY IT!	6
<b>1.3 COMPILING AND RUNNING A JAVA PROGRAM</b>	<b>6</b>
<b>1.4 COMMENTING</b>	<b>7</b>
<b>1.5 MAIN</b>	<b>7</b>
<b>2. NUTS AND BOLTS</b>	<b>8</b>
<b>2.1 VARIABLES AND DATA TYPES</b>	<b>8</b>
2.1.1 DATA TYPES	8
2.1.2 VARIABLE NAMES	9
2.1.3 SCOPE	9
2.1.4 VARIABLE INITIALISATION	10
2.1.5 FINAL VARIABLES	10
<b>2.2 OPERATORS</b>	<b>10</b>
2.2.1 TRY IT!	11
<b>2.3 EXPRESSIONS</b>	<b>11</b>
2.3.1 DEFINITION	12
<b>2.4 CONTROL FLOW STATEMENTS</b>	<b>12</b>
<b>2.5 ARRAYS AND STRINGS</b>	<b>12</b>
2.5.1 ARRAYS	12
2.5.2 STRINGS	13
2.5.3 STRING CONCATENATION	14
2.5.4 TRY IT!	14
<b>3. OBJECTS AND CLASSES</b>	<b>15</b>
<b>3.1 INTRODUCTION</b>	<b>15</b>
<b>3.2 CLASS DECLARATION</b>	<b>15</b>
3.2.1 TRY IT!	15
<b>3.3 CONSTRUCTORS</b>	<b>16</b>
3.3.1 CONSTRUCTOR NAME	16
3.3.2 PARAMETER LIST	16
3.3.3 TRY IT!	16
<b>3.4 OBJECT CREATION</b>	<b>16</b>
<b>3.5 INSTANCE &amp; CLASS VARIABLES</b>	<b>16</b>

3.5.1 TRY IT!	17
<b>4. METHODS</b>	<b>18</b>
<b>4.1 A BRIEF OVERVIEW</b>	<b>18</b>
<b>4.2 METHOD DECLARATION</b>	<b>18</b>
4.2.1 TRY IT!	18
<b>4.3 LOCAL VARIABLES</b>	<b>18</b>
<b>4.4 INSTANCE AND CLASS METHODS</b>	<b>18</b>
4.4.1 TRY IT!	18
<b>5. FLOW OF CONTROL</b>	<b>20</b>
<b>5.1 RETURNING FROM METHODS</b>	<b>20</b>
<b>5.2 IF . . THEN STATEMENTS</b>	<b>20</b>
5.2.1 TRY IT!	21
5.2.2 HINT	21
5.2.3 TRY IT!	21
<b>5.3 SWITCH STATEMENTS</b>	<b>21</b>
5.3.1 TRY IT!	22
<b>5.4 FOR LOOPS</b>	<b>22</b>
5.4.1 TRY IT!	22
<b>5.5 WHILE LOOPS</b>	<b>22</b>
<b>6. ERROR HANDLING</b>	<b>24</b>
6.0.1 DEFINITION	24
<b>6.1 THROWS CLAUSE</b>	<b>24</b>
<b>6.2 CATCH CLAUSE</b>	<b>25</b>
<b>6.3 EXCEPTION HANDLING EXAMPLE</b>	<b>25</b>
6.3.1 TRY IT!	25
<b>7. ADVANCED FEATURES</b>	<b>27</b>
<b>7.1 ENCAPSULATION</b>	<b>27</b>
<b>7.2 INHERITANCE</b>	<b>27</b>
7.2.1 THE BENEFITS OF INHERITANCE	28
7.2.2 TRY IT!	29
<b>8. ASSIGNMENT</b>	<b>32</b>
<b>8.1 AIMS</b>	<b>32</b>
<b>8.2 REQUIREMENTS</b>	<b>32</b>
8.2.1 PROGRAM CONTENT	32
8.2.2 PROGRAM FLOW	32
8.2.3 SPECIAL FEATURES	32
<b>8.3 SCOPE</b>	<b>32</b>

## Preface

### About this Document

This document is designed to assist you in:

- Logging on to the network
- Using a simple text editor (Notepad)
- Introduce you to the main Java programming tools (Java & Javac)
- Learning the Java programming language
- Simple object-oriented techniques

### Program Exercises

The example code that you are expected to try are denoted by a “*Try it!*” heading against either the code or exercise description. Read the description, enter the code into an editor as described, compile and then run the code. This will be explained the section that follows.

### Logging on to the Network

You will be issued with a network login and a case-sensitive password. Your login will look something like the following:

```
cs98xxx  
or  
ci99xxx
```

where xxx are three initials.

To log into the NT machines, at the welcome screen with the Brunel Logo, hold down <ctrl> & <alt> and tap the <delete> key. This should bring up a login screen where you can enter your login id and password on separate lines. Type your login in lower-case and then your password as it appears (include case and funny characters) and hit <return> to login. Eventually you will be presented with the standard Windows NT desktop. This is where you will work using the Java 2<sup>m</sup> Software Development Kit

### Command Prompt

To activate the command prompt where the Java tools are used from:

Start Menu > Programs > Command Prompt

A command prompt will be needed throughout this course to compile and run programs. Once one has been opened, don't bother to close it.

### Working area

The files that you work on during this course can be stored in your personal network file space located on the h: drive. To access this, open up a command prompt as described above and type the following at the prompt:

```
h:\ <return>
```

It is also wise to create a separate directory to store your programs in. To create a directory called java on your h: drive, type:

```
md java <return>
```

and then:

```
cd java <return>
```

to move into that directory.

### **Text Editor**

To enter your Java programs and examples, I recommend that you use Notepad. To open Notepad:

Start menu > Programs > Accessories > Notepad.

This should bring up a separate window in which to enter your programs. Once entered, code can be saved by selecting the Save as... option from the File menu. A new window should appear that contains details regarding where you are saving your document and file name.

Ensure that you are in your h:\ drive, if not select it by choosing the drop down menu next to "Save in".

Then double click on the java folder (where you will store all your code for this tutorial).

The "Save as type" option should always be set to "All files (\*.\*)".

Finally, enter a file name in including a suffix and click on the save button:

e.g. myfile.java

## 1. Introduction

### 1.1 Java

As a language Java is

- Simple - Java has the bare bones functionality needed to implement its rich feature set. It does not add lots of syntactic sugar or unnecessary features.
- Object-Oriented - Almost everything in Java is either a class, a method or an object. Only the most basic primitive operations and data types (int, for, while, etc.) are at a sub-object level.
- Platform Independent - Java programs are compiled to a byte code format that can be read and run by interpreters on many platforms including Windows 95, Windows NT, and Solaris 2.3 and later.
- Safe - Java code can be executed in an environment that prohibits it from introducing viruses, deleting or modifying files, or otherwise performing data destroying and computer crashing operations.
- High Performance - Java can be compiled on the fly with a Just-In-Time compiler (JIT) to code that rivals C++ in speed.
- Multi-Threaded - Java is inherently multi-threaded. A single Java program can have many different things processing independently and continuously.

### 1.2 The Hello World program

Your first program, is known as “HelloWorld” and is probably the first program that anyone learning Java encounters.

*Try it!*

Create a file named HelloWorld.java (Java is case-sensitive) with the Java code shown here (use Notepad to enter it):

```
/**
 * The HelloWorld class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //Display the string.
    }
}
```

### 1.3 Compiling and running a Java program

The HelloWorld.java file should be saved into your current directory. To actually compile and run any Java program, you need open up a command prompt from the Start Menu. “cd” to the directory where the above file is located.

To compile the program, type:

```
javac HelloWorld.java
```

If your first effort won't compile, here are a few common mistakes:

1. Did you put a semicolon after System.out.println("Hello World")?
2. Did you include the closing bracket?
3. Did you type everything exactly as it appears here? In particular did you use the same capitalisation? Java is case sensitive. class is not the same as Class for example.
4. Were you in the same directory as HelloWorld.java when you typed javac HelloWorld.java?

This will produce another file called “HelloWorld.class” in the same directory. This is the Java-Bytecode or actual program that can be run using the Java interpreter.

To run the program, type:

```
java HelloWorld
```

You should see "Hello World!" displayed on the command line window. You have now successfully compiled and run your first Java program.

## 1.4 Commenting

Commenting allows the annotation of code in a similar fashion to post-it stamps. All programming languages support comments of some kind. In Java, there are two types of comments available. Java uses a `/*` to begin a header comment and a `*/` to close it. Java uses a `//` to begin inline comments and ignores everything else on the line.

For example:

```
/**
 * This is a header comment.
 */

// This is an inline comment.
```

## 1.5 Main

The file that you would like to execute (command Java “your\_program.class”) needs a place to start executing from. This is known as the main method and the Java interpreter will look for this when attempting to run the program. The structure of the main method always looks as follows:

```
public static void main(String[] args) {
    //Your code should start here.
}
```

The String array `args`, actually contains any arguments passed from the command line. If the following was used to run your program:

```
java myProgram hello goodbye
```

`args[0]` would equal “hello” and `args[1]` would equal “goodbye”

The main method contains the starting point of your program and normally passes program control to another part of your software commonly called the main program loop. At least one of your classes must contain a main method to actually get your software up and running.

## 2. Nuts and Bolts

### 2.1 Variables and Data Types

Like other programming languages, Java allows you to declare variables in your programs. You use variables to contain data that can change during the execution of the program. All variables in Java have a type, a name, and scope. Variables are the nouns of a programming language—that is, they are the entities (values and data) that act or are acted upon.

A variable declaration always contains two components: the type of the variable and its name. The location of the variable declaration, that is, where the declaration appears in relation to other code elements, determines its scope.

Here is an example of a variable declaration for a string and an integer (int), and its initialisation.

```
String myString = "We hate Java!";  
int myInt = 33;
```

#### 2.1.1 Data Types

All variables in the Java language must have a data type. A variable's data type determines the values that the variable can contain and the operations that can be performed on it. For example, the declaration `int myInt` declares that `myInt` is an integer (int). Integers can contain only integral values (both positive and negative), and you can use the standard arithmetic operators (+, -, \*, and /) on integers to perform the standard arithmetic operations (addition, subtraction, multiplication, and division, respectively).

There are two major categories of data types in the Java language: primitive and reference. The following table lists, by keyword, all of the primitive data types supported by Java, their sizes and formats, and a brief description of each.

Type	Size/Format	Description
	(integers)	
byte	8-bit two's complement	Byte-length integer
short	16-bit two's complement	Short integer
int	32-bit two's complement	Integer
long	64-bit two's complement	Long integer
	(real numbers)	
float	32-bit IEEE 754	Single-precision floating point
double	64-bit IEEE 754	Double-precision floating point
	(other types)	
char	16-bit Unicode character	A single character
boolean	true or false	A boolean value (true or false)

A variable of primitive type contains a single value of the appropriate size and format for its type: a number, character, or boolean value. For example, the value of an `int` is an integer, the value of a `char` is a 16-bit Unicode character, and so on.

Arrays, classes, and interfaces are reference types. The value of a reference type variable, in contrast to that of a primitive type, is a reference to the actual value or set of values represented by the variable. A reference is like your friend's address: The address is not your friend, but it's a way to reach your friend. A reference type variable is not the array or object itself but rather a way to reach it.

```
Reader in = new Reader();
```

The example above uses one variable of reference type, `in`, which is a `Reader` object. When used in a statement or expression, the name `in` evaluates to a reference to the object. So you can use the object's name to access its member variables or call its methods (more on this later).

## 2.1.2 Variable Names

A program refers to a variable's value by its name. For example, when a program wants to refer to the value of the Reader object above, it simply uses the name *in*.

In Java, the following must hold true for a variable name

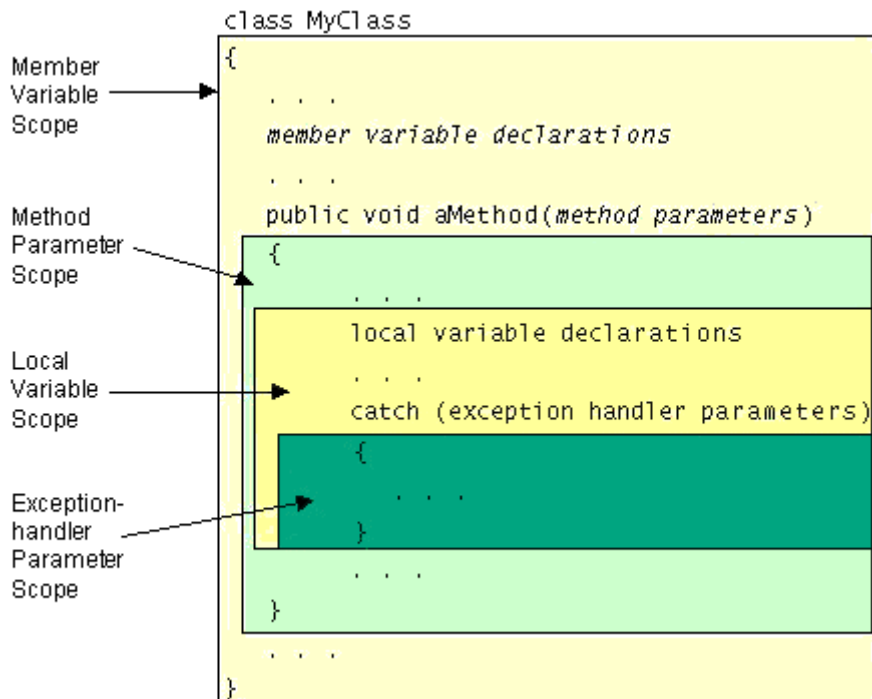
- It must be a legal Java identifier comprised of a series of Unicode characters. Unicode is a character-coding system designed to support text written in diverse human languages. It allows for the codification of up to 65,536 characters, currently 34,168 have been assigned. This allows you to use characters in your Java programs from various alphabets, such as Japanese, Greek, Cyrillic, and Hebrew. This is important so that programmers can write code that is meaningful in their native languages.
- It must not be a keyword or a boolean literal (true or false).
- It must not have the same name as another variable whose declaration appears in the same scope.

**By Convention:** Variable names begin with a lowercase letter and class names begin with an uppercase letter. If a variable name consists of more than one word, such as `isVisible`, the words are joined together and each word after the first begins with an uppercase letter.

## 2.1.3 Scope

A variable's scope is the block of code within which the variable is accessible and determines when the variable is created and destroyed. The location of the variable declaration within your program establishes its scope and places it into one of these four categories:

- Member variable
- Local variable
- Method parameter
- Exception-handler parameter



A member variable is a member of a class or an object. It can be declared anywhere in a class but not in a method. The member is available to all code in the class. You can declare local variables anywhere in

a method or within a block of code in a method. In general, a local variable is accessible from its declaration to the end of the code block in which it was declared. Method parameters are formal arguments to methods and constructors and are used to pass values into methods. The scope of a method parameter is the entire method for which it is a parameter. Exception-handler parameters are similar to method parameters but are arguments to an exception handler rather than to a method or a constructor.

Many of the concepts introduced above (methods, method parameters and exceptions) will be looked at more closely in later sections.

### 2.1.4 Variable Initialisation

Local variables and member variables can be initialised with an assignment statement when they're declared. The data type of both sides of the assignment statement must match. The example above provides an initial value of 33 for `myInt` when declaring it:

```
int myInt = 33;
```

The value assigned to the variable must match the variable's type.

Method parameters and exception-handler parameters cannot be initialised in this way. The value for a parameter is set by the caller.

### 2.1.5 Final Variables

You can declare a variable in any scope to be final, including parameters to methods and constructors. The value of a final variable cannot change after it has been initialised.

To declare a final variable, use the `final` keyword in the variable declaration before the type:

```
final int aFinalVar = 0;
```

The previous statement declares a final variable and initialises it, all at once. Subsequent attempts to assign a value to `aFinalVar` result in a compiler error. You may, if necessary, defer initialisation of a final variable. Simply declare the variable and initialise it later, like this:

```
final int blankfinal;  
.  
.  
.  
blankfinal = 0;
```

A final variable that has been declared but not yet initialised is called a blank final. Again, once a final variable has been initialised it cannot be set and any later attempts to assign a value to `blankfinal` result in a compile-time error.

## 2.2 Operators

The Java language provides a set of operators, which you use to perform a function on one or two variables. Operators perform some function on either one or two operands or three operands. Operators that require one operand are called unary operators. For example, `++` is a unary operator that increments the value of its operand by 1. Operators that require two operands are binary operators. For example, `=` is a binary operator that assigns the value from its right-hand operand to its left-hand operand. And finally ternary operators are those that require three operands. The Java language has one ternary operator, `?:`, which is a short-hand if-else statement.

In addition to performing the operation, an operator also returns a value. The value and its type depends on the operator and the type of its operands. For example, the arithmetic operators, which perform basic arithmetic operations such as addition and subtraction, return numbers-the result of the arithmetic operation. The data type returned by the arithmetic operators depends on the type of its operands: If you add two integers, you get an integer back. An operation is said to evaluate to its result.

Arithmetic operators (which will be the main operator that you will encounter during this course) can be used in the following manner:

```
addThis + toThis
divideThis % byThis
...
```

and so on.

The outputs on these operations can be put together into a String and printed to the command line:

```
int a = 534;
int b = 732;
System.out.println("Multiplication Result = " + (a * b));
```

## 2.2.1 Try it!

Modify the HelloWorld's "main" method to print out the correct value.

Incrementing values by one step at a time can also be handled in the following format:

```
int a = 0;
a++;
System.out.println(a);
a++;
System.out.println(a);
a++;
System.out.println(a);
```

Again, modify HelloWorld to demonstrate this.

To test the condition or relation of two values:

```
int a = 50;
int b = 100;

if(a==b) {
    System.out.println("Condition True");
}
else {
    System.out.println("Condition False");
}
```

Other conditions include:

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal
!=	not equal

Again, modify HelloWorld using the code above as a guide to demonstrate your understanding of these concepts.

## 2.3 Expressions

Operators, variables, and method calls can be combined into sequences known as expressions. The real work of a Java program is achieved through expressions. Expressions perform the work of a Java program. Among other things, expressions are used to compute and assign values to variables and to help control the execution flow of a program. The job of an expression is two-fold: perform the computation indicated by the elements of the expression and return some value that is the result of the computation.

## 2.3.1 Definition

An expression is a series of variables, operators, and method calls (constructed according to the syntax of the language) that evaluates to a single value.

Take for example this compound expression:

```
x * y * z
```

In this particular example, the order in which the expression is evaluated is unimportant because the results of multiplication is independent of order--the outcome is always the same no matter what order you apply the multiplication's. However, this is not true of all expressions. For example, this expression gives different results depending on whether you perform the addition or the division operation first:

```
x + y / 100
```

You can direct the Java compiler explicitly how you want an expression evaluated with balanced parentheses ( and ). For example to make the previous expression unambiguous, you could write  $(x + y) / 100$ .

## 2.4 Control Flow Statements

There are also a group of functions known as control flow statements. As the name implies, control flow statements control the flow of the program. In other words, control flow statements govern the sequence in which a program's statements are executed. These are dealt with in a section on their own (see below).

## 2.5 Arrays and Strings

Two important data types in any programming language are arrays and strings. In Java, both arrays and strings are full-blown objects. Other languages force programmers to manage data in arrays and strings at the memory level using direct pointer manipulation.

Like other programming languages, Java allows you to collect and manage multiple values through an array object. You manage data comprised of multiple characters through a String object.

### 2.5.1 Arrays

As for other variables, before you can use an array you must first declare it. Again, like other variables, a declaration of an array has two primary components: the array's type and the array's name. An array's type includes the data type of the elements contained within the array. For example, the data type for an array that contained all integer elements is array of integers. You cannot have a generic array--the data type of its elements must be identified when the array is declared. Here's a declaration for an array of integers:

```
int[] arrayOfInts;
```

The `int[]` part of the declaration indicates that `arrayOfInts` is an array of integers. The declaration does not allocate any memory to contain the array elements.

To allocate memory for the elements of the array, you must instantiate the array. You do this using Java's `new` operator. (Actually, the steps you take to create an array are similar to the steps you take to create an object from a class: declaration, instantiation, and initialisation. The `new` operator is covered in greater depth in a later section. For now, imagine it as allocating the memory required to store your data structure.

The following statement allocates enough memory for `arrayOfInts` to contain ten integer elements.

```
int[] arrayOfInts = new int[10]
```

In general, when creating an array, you use the new operator, plus the data type of the array elements, plus the number of elements desired enclosed within square brackets ( '[' and ']' ).

```
elementType[] arrayName = new elementType[arraySize]
```

Now that some memory has been allocated for your array, you can assign values to its elements and retrieve those values:

```
arrayOfInts [0] = 3;
arrayOfInts [1] = 2;
arrayOfInts [2] = 16;
arrayOfInts [3] = 40;
arrayOfInts [4] = 33;
arrayOfInts [5] = 23;
arrayOfInts [6] = 76;
arrayOfInts [7] = 100;
arrayOfInts [8] = 50;
arrayOfInts [9] = 25;
arrayOfInts [10] = 1;
```

There are faster ways of assigning values to an array by using loops (see Flow of Control section). This example shows that to reference an array element, you append square brackets to the array name. Between the square brackets indicate (either with a variable or some other expression) the index of the element you want to access. Note that in Java, array indices begin at 0 and end at the array length minus 1.

Arrays can contain any legal Java data type including reference types such as objects or other arrays. For example, the following declares an array that can contain ten String objects.

```
String[] arrayOfStrings = new String[10];
```

The elements in this array are reference types, that is, each element contains a reference to a String object. At this point, enough memory has been allocated to contain the String references, but no memory has been allocated for the Strings themselves. If you attempted to access one of arrayOfStrings elements at this point, you would get a NullPointerException because the array is empty and contains no Strings and no String objects. This is often a source of some confusion for programmers new to the Java language. You have to allocate the actual String objects separately:

```
for (int i = 0; i < arrayOfStrings.length; i++) {
    arrayOfStrings[i] = new String("Hello " + i);
}
```

Don't worry about the supporting bits (the "for" operator and others). This will be covered later.

### 2.5.2 Strings

A sequence of character data is called a string and is implemented in the Java environment by the String class provided by the Java developers. A simple String declaration and initialisation looks like this:

```
String myString = "Java";
```

HelloWorld's main method uses Strings in its the declaration of the args array:

```
String[] args
```

This code explicitly declares an array named args that contains String objects. The empty brackets indicate that the length of the array is unknown at compilation time because the array is passed in at runtime.

A literal string (a string of characters between double quotation marks) can be used in a program:

```
"String "
...
```

" chars."

String objects are immutable--that is, they cannot be changed once they've been created.

### 2.5.3 String Concatenation

Java lets you concatenate strings together easily using the + operator. The following code snippet concatenates three strings together to produce its output:

```
"String " + myInt + " chars."
```

Two of the strings concatenated together are literal strings: "String " and " chars." The third string--the one in the middle--is actually an integer that first gets converted to a string and then is concatenated to the others.

### 2.5.4 Try it!

Take the HelloWorld program, edit the main method to initialise and output the values for the following types of variables:

- A string with your name.
- An integer with your age.
- A boolean value.
- An integer array of size two, initialised with any two numbers.
- An expression that will evaluate to true if the first number from your array is larger than your second, otherwise it will evaluate to false.

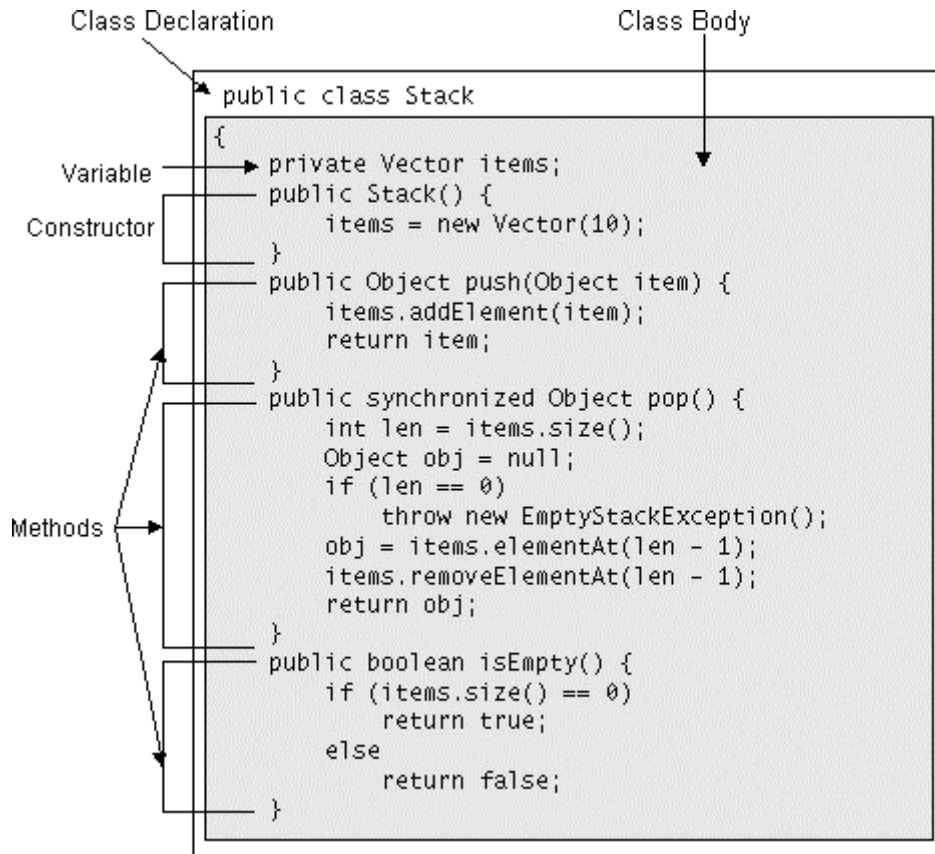
N.B. To output each of the results of the above, use the following code:

```
System.out.println("What your trying to output :"+yourExpression);
```

## 3. Objects and Classes

### 3.1 Introduction

The diagram below describes the structure of a class, the blueprint for an object in both Java and object-oriented methodology. Each component will be looked at in more detail through this and subsequent sections.



### 3.2 Class Declaration

The syntax for a Java class declaration is as follows:

```

[class_modifiers] class <class_name> [extends <superclass_name>
                                     [implements <interface_1>,
                                     <interface_2>,
                                     . . . ] {
    // class body
}
    
```

#### 3.2.1 Try it!

Here is a skeleton class, type it in and compile it. You should be left with a `.class` file, but it won't do anything yet.

```

class Vehicle {
    public String colour;
    // class body
}
    
```

## 3.3 Constructors

A constructor is a special kind of method that is used to initialise newly created objects. Java has a special way to declare the constructor and a special way to invoke the constructor, as described in the next section. A class can actually have multiple constructors, to allow different types of initialisation based on parameters passed.

### 3.3.1 Constructor name

The name of the constructor must be the same as the name of the class it constructs. So, if the class is called `Vehicle` the constructor must be called `Vehicle` as well. The constructor has no return value. So its return type is implicitly `void` (see `Methods` section for return type).

### 3.3.2 Parameter List

The parameter list for a constructor is identical in syntax to the parameter list for an ordinary method.

### 3.3.3 Try it!

Let us extend our `Vehicle` class to include constructors...

```
public class Vehicle {  
    public String colour;  
    public Vehicle() {  
        // does something  
    }  
    public Vehicle(String colour) {  
        // does something and..  
        this.colour = colour;  
    }  
    // class body  
}
```

As you can see, the class above has two constructors, one that requires no parameters, and one that requires a string. The second constructor initialises the object based on the parameter passed. This will become apparent as we pad out the contents of the class.

## 3.4 Object Creation

In Java the only way to create a new class is with the `new` operator. The `new` operator actually creates the object and then calls the constructor in the class body. If there are multiple constructors, then the one that matches the number of parameters passed is used. To create (or instantiate) an object based on the `Vehicle` class above, the `new` operator is used in the following format:

```
Vehicle myVehicle = new Vehicle(); // no parameter constructor called
```

Or...

```
Vehicle myVehicle = new Vehicle("black"); // single parameter constructor called
```

## 3.5 Instance & Class Variables

Inside a class there are two types of variables, instance and class. To explain how these work, we need to first see the syntax distinction between the two.

```
public class Vehicle {  
    public static int version = 1; // class variable  
    public String colour; // instance variable  
    public Vehicle() {
```

```
        // does something
    }

    public Vehicle(String colour) {
        // does something and...
        this.colour = colour;
    }

    // class body
}
```

### 3.5.1 Try it!

As the above class shows, a class variable is defined by placing *static* just before the *type*. The instance variable does not have the *static* modifier. As new objects based on the `Vehicle` class are created using the `new` operator (see object creation above), each individual object is assigned its own version of `colour`. But the objects must share the value of `version` as there can only be one copy of it within the Java Virtual Machine. The following example illustrates how to access these variables and should clarify things. Try it!

```
public class Vehicle {

    public static int version = 1           // class variable
    public String colour;                  // instance variable

    public Vehicle() {
        // does something
    }

    public Vehicle(String colour) {
        // does something and...
        this.colour = colour;
    }

    public static void main(String[] args) {
        Vehicle carA = new Vehicle();
        Vehicle carB = new Vehicle();

        carA.colour = "Green";             //instance variable for carA
        carB.colour = "Blue";              //instance variable for carB

        System.out.println("carA :"+carA.colour);
        System.out.println("carB :"+carB.colour);

        System.out.println("Version :"+Vehicle.version);
        Vehicle.version = 2;
        System.out.println("Version :"+Vehicle.version);
    }
}
```

Both `carA` and `carB` have their own versions of `colour` but share `version`.

## 4. Methods

### 4.1 A Brief Overview

Each calculation part of a program is called a method. Methods are logically the same as C's functions, Pascal's procedures and functions, and Fortran's functions and subroutines. In general, they are chunks of code that can be called on a particular class. They can accept parameters as arguments. Every method you declare belongs to a specific class and must appear somewhere in the class body. Methods have two primary parts: the method declaration itself and the method body.

### 4.2 Method Declaration

Methods are declared using the following syntax:

```
[method_modifiers] <return_type> <method_name> ([<param_type param_name>,  
                                                <param_type param_name>,  
                                                . . .]) {  
    // method body  
}
```

A method definition must specify the type of value it will return. If the method does not return a value, then the keyword `void` must be used. To return a value in Java you must use the `return` keyword. A method's parameters are defined in a comma-separated list enclosed in parentheses after the name of the method. A parameter consists of two parts: the parameter type and the parameter name. If a method has no parameters, then only an empty pair of parentheses is used. An actual method description looks like this, add it to the `Vehicle` class:

#### 4.2.1 Try it!

Add these methods to your `Vehicle` class. Make sure it all compiles.

```
public void changeColour(String newColour) {  
    this.colour = newColour;  
}
```

This method returns the current colour of the car:

```
public String getColour() {  
    return colour;  
}
```

### 4.3 Local Variables

Methods may have local variables. These variables are declared in the body of the method. They exist only during the execution of the method and then are garbage collected (the memory is returned to the system) once the method is complete. Local variables are very similar to instance variables. Local variables can be either base types (i.e. `int`, `float`, `double`) or they can be references to other classes which you or other people have created.

### 4.4 Instance and Class Methods

The difference between instance and class methods is exactly the same as instance and class variables. This example will explain the difference:

#### 4.4.1 Try it!

Complete the `Vehicle` class based on the code below:

```
class Vehicle {  
    public static int version = 1           // class variable  
    public String colour;                 // instance variable  
  
    public Vehicle() {
```

```
        // does something
    }

    public Vehicle(String colour) {
        // does something and..
        this.colour = colour;
    }

    public void changeColour(String newColour) { // instance method
        this.colour = newColour;
    }

    public String getColour() { // instance method
        return colour;
    }

    public static int getVersion() { // static method
        return Vehicle.version;
    }
}
```

To use instance methods, first create an instance of the class:

```
Vehicle carA = new Vehicle();
```

Then to call an instance method:

```
carA.changeColour("Yellow");
System.out.println(carA.getColour());
```

To use a static method, there is no need to create the class, simple type:

```
System.out.println(Vehicle.getVersion());
```

Add a main method to the above class, add the above four lines into the main method, compile and run it to see this in action.

## 5. Flow of Control

All but the most trivial computer programs need to make decisions. They need to test some condition and operate differently based on that condition. This is quite common in real life. For instance you stick your hand out the window to test if it's raining. If it is raining then you take an umbrella with you. If it isn't raining then you don't.

### 5.1 Returning From Methods

After calling a method, the program may need some way of returning to the calling segment of code. You use return to exit from the current method and jump back to the statement within the calling method that follows the original method call. There are two forms of return: one that returns a value and one that doesn't. To return a value, simply put the value (or an expression that calculates the value after the return keyword:

```
return aValue+anotherValue;
```

The value returned by return must match the type of method's declared return value.

When a method is declared void use the form of return that doesn't return a value:

```
return;
```

If a method does not designate a return type (i.e. void) then it does not need a return command at the end of the code. The terminating } will also cause program flow to return to the calling code as well.

### 5.2 If . . Then Statements

All programming languages have some form of an if statement that allows you to test conditions. Java's if-else statement provides your programs with the ability to selectively execute other statements based on some criteria. For example, suppose that your program printed debugging information based on the value of some boolean variable named DEBUG. If DEBUG were set to true, then your program would print debugging information such as the value of some variable like x. Otherwise, your program would proceed normally. A segment of code to implement this might look like this:

```
. . .  
if (DEBUG)  
    System.out.println("DEBUG: x = " + x);  
. . .
```

This is the simplest version of the if statement: the statement governed by the if is executed if some condition is true. Generally, the simple form of if can be written like this:

```
if (expression)  
    statement
```

So, what if you wanted to perform a different set of statements if the expression is false? Well, you can use the else statement for that. Consider another example. Suppose that your program needs to perform different actions depending on whether the user clicks on the OK button or the Cancel button in an alert window. Your program could do this using an if statement:

```
. . .  
// response is either OK or CANCEL depending  
// on the button that the user pressed  
if (response == OK) {  
    . . .  
    // code to perform OK action  
    . . .  
}  
else {  
    . . .  
    // code to perform Cancel action  
    . . .  
}
```

### 5.2.1 Try it!

Take the Vehicle class, and create a new method called `doorTest(int numberDoors)`. The method should have a return type of `void` and take a single `int` parameter. Take the above code and modify it to allow for the following conditions:

- If the number of doors parameter passed is 4, then the Vehicle is valid, and will output a message to that effect.
- Else the Vehicle is not valid and will inform the user to that effect.

### 5.2.2 Hint

The if test should look something like: (parameter == a number)

This particular use of the else statement is the catch-all form. The else block is executed if the if part is false. There is another form of the else statement, else if which executes a statement based on another expression. For example, suppose that you wrote a program that evaluate the type of car this is based on the number of doors. You could use an if statement with a series of companion else if statements, and an else to write this code:

```
if (numberDoors == 2) {
    System.out.println("Two door car");
}
else if (numberDoors == 3) {
    System.out.println("Two door car and a boot");
}
else if (numberDoors == 4) {
    System.out.println("Four door car");
}
else if (numberDoors == 5) {
    System.out.println("Two door car and a boot");
}
else {
    System.out.println("I don't think this is a car mate!");
}
```

### 5.2.3 Try it!

Modify the Vehicle class by adding another method called `void type(numberDoors)` that uses and demonstrates the code shown here by passing alternative values of `numberDoors`. The method declaration should be the same as the `doorTest` method.

## 5.3 Switch Statements

Use the switch statement to conditionally perform statements based on some expression. For example, suppose that your program contained an integer named `month` whose value indicated the month in some date. Suppose also that you wanted to display the name of the month based on its integer equivalent. You could use Java's switch statement to perform this feat:

```
int month;
.
.
switch (month) {
    case 1: System.out.println("January"); break;
    case 2: System.out.println("February"); break;
    case 3: System.out.println("March"); break;
    case 4: System.out.println("April"); break;
    case 5: System.out.println("May"); break;
    case 6: System.out.println("June"); break;
    case 7: System.out.println("July"); break;
    case 8: System.out.println("August"); break;
    case 9: System.out.println("September"); break;
    case 10: System.out.println("October"); break;
}
```

```
        case 11: System.out.println("November"); break;
        case 12: System.out.println("December"); break;
    }
```

### 5.3.1 Try it!

Implement this code into a NEW class called Calendar to confirm that it works. This code ideally should appear in its own method and the month value passed to it as a parameter. Then modify the code further by creating another method to display the day of the week where day one is equal to Monday and day seven equal to Sunday. The code above should be modified to do this.

The resulting class Calendar should contain two methods, month(int month) and day(int day). Both should return nothing (i.e. void). Add a main method and test the methods by passing varying values to the methods to confirm that the code works.

The switch statement evaluates its expression, in this case, the value of month, and executes the appropriate case statement. This is actually an alternative to the if...then statement.

## 5.4 For Loops

Use the for loop when you know the constraints of the loop (its initialisation instruction, termination criteria, and increment instruction). For instance, for loops are often used to iterate over the elements in an array, or the characters in a string.

```
// a is an array of some kind
. . .
int i;
int length = a.length;
for (i = 0; i < length; i++) {
    . . .
    // do something to the i th element of a
    . . .
}
```

You know when writing the program that you want to start at the beginning of the array, stop at the end, and hit every element. Thus the for statement is a good choice. The general form of the for statement can be expressed like this:

```
for (initialisation; termination; increment)
    statements
```

*initialisation* is a statement that initialises the loop--its executed once at the beginning of the loop. *termination* is an expression that determines when to terminate the loop. This expression is evaluated at the top of each iteration of the loop. When the expression evaluates to false, the for loop terminates. Finally, *increment* is an expression that gets invoked for each iteration through the loop. Any (or all) of these components can be empty statements (a single semi-colon by itself).

### 5.4.1 Try it!

The following code will show how this works, modify the main method of the HelloWorld program to contain only the following:

```
for(int i=0; i<10; i++) {
    System.out.println(i);
}
```

## 5.5 While Loops

Java provides another loop, the do-while loop, which is similar to the while loop you met earlier except that the expression is evaluated at the bottom of the loop:

```
do {
    statements
```

```
} while (booleanExpression);
```

The do-while statement is a less commonly used loop construct in programming but does have its uses. For example, the do-while is convenient to use when the statements within the loop must be executed at least once.

```
int c=0;
. . .
do {
    System.out.println(c++); // prints and increments the value of c
    . . .
} while (c != 10);
```

## 6. Error Handling

When an error occurs within a Java method, the method can throw an exception to indicate to its caller that an error occurred and the type of error that occurred. The calling method can use the try, catch, and finally statements to catch and handle the exception.

The goal of exception handling is to be able to define the regular flow of the program in part of the code without worrying about all the special cases. Then, in a separate block of code, you cover the exceptional cases. This produces more legible code since you don't need to interrupt the flow of the algorithm to check and respond to every possible strange condition. The runtime environment is responsible for moving from the regular program flow to the exception handlers when an exceptional condition arises.

In practice what you do is write blocks of code that may generate exceptions inside try-catch blocks. You try the statements that generate the exceptions. Within your try block you are free to act as if nothing has or can go wrong. Then, within one or more catch blocks, you write the program logic that deals with all the special cases.

### 6.0.1 Definition

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

### 6.1 Throws Clause

In Java terminology, creating an exception object and handing it to the runtime system is called throwing an exception. After a method throws an exception, the runtime system leaps into action to find someone to handle the exception. The set of possible "someones" to handle the exception is the set of methods in the call stack of the method where the error occurred. The runtime system searches backwards through the call stack, beginning with the method in which the error occurred, until it finds a method that contains an appropriate exception handler. An exception handler is considered appropriate if the type of the exception thrown is the same as the type of exception handled by the handler. Thus the exception bubbles up through the call stack until an appropriate handler is found and one of the calling methods handles the exception. The exception handler chosen is said to catch the exception. If the runtime system exhaustively searches all of the methods on the call stack without finding an appropriate exception handler, the runtime system (and consequently the Java program) terminates.

Before you can catch an exception, some Java code somewhere must throw one. Any Java code can throw an exception: your code, code from a package written by someone else (such as the packages that come with the Java development environment), or the Java runtime system. Regardless of who (or what) throws the exception, it's always thrown with the Java throw statement.

Here's an example of a throw statement:

```
throw someThrowableObject;
```

Let's look at the throw statement in context. The following method is taken from a class that implements a common stack object. The pop method removes the top element from the stack and returns it:

```
public Object pop() throws EmptyStackException {
    Object obj;

    if (size == 0)
        throw new EmptyStackException();

    obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size--;
    return obj;
}
```

The pop method checks to see if there are any elements on the stack. If the stack is empty (its size is equal to 0), then pop instantiates a new EmptyStackException object and throws it. The EmptyStackException class is defined in the java.util package.

### 6.2 Catch Clause

A method can catch an exception by providing an exception handler for that type of exception. To handle the pop method above in case of an exception:

```
...
try {
    Object returnedObject = stack.pop();
}
catch(EmptyStackException e) {
    System.err.println("Stack empty, I think we may have a problem!" + e);
}
...
```

If a method is capable of throwing more than one exception than the calling code must handle all possible exceptions:

```
...
try {
    Object return aObject.aMethod();
}
catch(AnException e) {
    System.err.println("First Exception: " + e);
}
catch(AnotherException e) {
    System.err.println("Second Exception: " + e);
}
...
```

The object aObject must declare that it can throw both types of exceptions in the aMethod definition.

### 6.3 Exception Handling Example

#### 6.3.1 Try it!

The code below describes a complete error-handling example. There are three classes in all. ExampleException creates a simple exception object for our program to throw. All this really contains is an error message to tell the offending code and the user what the problem is. The Maths class (make sure that you put the "s" on the end of Maths) contains a static method used to calculate a division operation. This method is capable of throwing an exception as can be seen by the method signature. The last class called ExceptionTest is the one that you actually execute. It tries to make use of the divide method in the Maths class and is able to catch any exceptions that might arise in the process.

```
/*
 * This class represents an Exception that can be thrown
 * by another class
 */

// This line declares that we will be using a predefined class
// provided by the Java developers
import java.lang.Exception;

// Creates a subclass of Exception
public class ExampleException extends Exception {

    // constructor
    public ExampleException() {
        super();
    }

    // constructor
    public ExampleException(String message) {
        super(message);
    }
}
```

## Java Primer Course

---

```
}

/*
 * The Maths class contains only one method, divide which takes two
 * int numbers as arguments, and attempts to divide them. If the
 * denominator (second argument) is equal to zero, it will create
 * an exception of type ExampleException and throw it to the calling
 * method.
 */

public class Maths {

    public static int divide(int first, int second) throws ExampleException {
        if(second==0) {
            throw new ExampleException("You cannot divide by zero.");
            /*
             * The above could also be as follows:
             * ExampleException e = new ExampleException("You cannot divide by zero");
             * throw e;
             */
        }
        // If there is no problem, then return the division of the two int's
        return first/second;
    }
}

/*
 * This class tests the divide class in maths to see what happens when
 * everything works fine and also when there is a problem
 */

public class ExceptionTest {
    public static void main(String[] args) {
        // Attempt 1, should be okay
        // lets try the method divide...
        try {
            System.out.println(Maths.divide(10,2));
        }
        // If it fails, catch the exception thrown by divide...
        catch(ExampleException e) {
            // ...and handle it
            System.err.println(e);
        }

        // Attempt 2, should create a problem
        // lets try the method divide...
        try {
            System.out.println(Maths.divide(2,0));
        }
        // If it fails, catch the exception thrown by divide...
        catch(ExampleException e) {
            // ...and handle it
            System.err.println(e);
        }

        // Attempt 3, should be okay
        // lets try the method divide...
        try {
            System.out.println(Maths.divide(100,5));
        }
        // If it fails, catch the exception thrown by divide...
        catch(ExampleException e) {
            // ...and handle it
            System.err.println(e);
        }
    }
}
}
```

## 7. Advanced Features

### 7.1 Encapsulation

To understand what encapsulation means, we need to comprehend the reasons behind object-oriented design. The definition of an object is a software bundle of variables and related methods. For example, if we were creating an object that described a car for example, it would need variables that portray the cars state and methods that define actions or behaviour of the car.

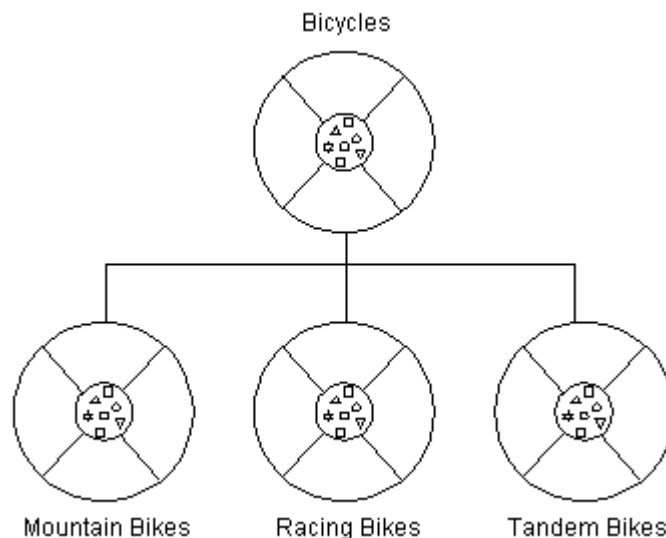
Encapsulating related variables and methods into a neat software bundle is a simple yet powerful idea that provides two primary benefits to software developers:

- Modularity-The source code for an object can be written and maintained independently of the source code for other objects. Also, an object can be easily passed around in the system. You can give your bicycle to someone else and it will still work.
- Information hiding-An object has a public interface that other objects can use to communicate with it. But the object can maintain private information and methods that can be changed at any time without affecting the other objects that depend on it. You don't need to understand the gear mechanism on your bike in order to use it.

### 7.2 Inheritance

A class inherits state and behaviour from its superclass. Inheritance provides a powerful and natural mechanism for organising and structuring software programs. Generally speaking, objects are defined in terms of classes. You know a lot about an object by knowing its class. Even if you don't know what a penny-farthing is, if I told you it was a bicycle, you would know that it had two wheels, handle bars, and pedals.

Object-oriented systems take this a step further and allow classes to be defined in terms of other classes. For example, mountain bikes, racing bikes, and tandems are all different kinds of bicycles. In object-oriented terminology, mountain bikes, racing bikes, and tandems are all subclasses of the bicycle class. Similarly, the bicycle class is the superclass of mountain bikes, racing bikes, and tandems.



Each subclass inherits state (in the form of variable declarations) from the superclass. Mountain bikes, racing bikes, and tandems share some states: cadence, speed, and the like. Also, each subclass inherits methods from the superclass. Mountain bikes, racing bikes, and tandems share some behaviours: braking and changing pedalling speed, for example.

However, subclasses are not limited to the state and behaviours provided to them by their superclass. What would be the point in that? Subclasses can add variables and methods to the ones they inherit from the superclass. Tandem bicycles have two seats and two sets of handle bars; some mountain bikes have an extra set of gears with a lower gear ratio.

Subclasses can also override inherited methods and provide specialised implementations for those methods. For example, if you had a mountain bike with an extra set of gears, you would override the "change gears" method so that the rider could actually use those new gears.

You are not limited to just one layer of inheritance. The inheritance tree, or class hierarchy, can be as deep as needed. Methods and variables are inherited down through the levels. In general, the further down in the hierarchy a class appears, the more specialised its behaviour.

### 7.2.1 The Benefits of Inheritance

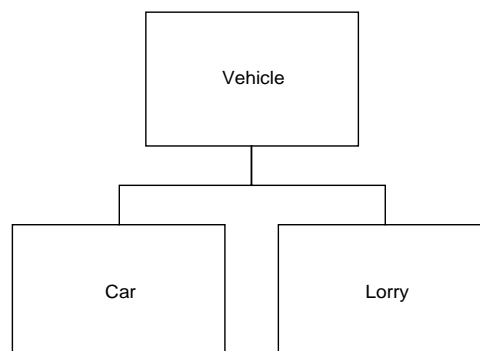
Subclasses provide specialised behaviours from the basis of common elements provided by the superclass. Through the use of inheritance, programmers can reuse the code in the superclass many times. Programmers can implement superclasses called abstract classes that define "generic" behaviours. The abstract superclass defines and may partially implement the behaviour but much of the class is undefined and unimplemented. Other programmers fill in the details with specialised subclasses.

How do we create a subclass? This is done by using the extends modifier. The following example shows how to define two subclasses of the Vehicle class.

```
public class Car extends Vehicle {
    public Car() {
        super(); // calls Vehicles constructor
        // now do Lorry specific initialisation
    }
    // class body
}

public class Lorry extends Vehicle {
    public Lorry() {
        super(); // calls Vehicles constructor
        // now do Lorry specific initialisation
    }
    // class body
}
```

Both the Car and Lorry classes would be able to access methods and variables that are in the Vehicle class. There are ways of protecting some methods and variables in the superclass by using modifiers (public, private, etc) but are out of scope of this tutorial.



The example also uses the `super()` method which is used to call the superclasses constructor. This is standard practice when working with inheritance and ensures that the object is created correctly.

### 7.2.2 Try it!

Consider the following example of inheritance in reference to the Vehicle, Car and Lorry classes, based on the class diagram above. First enter the code in (they will have all changed from the versions that you all currently have).

#### Vehicle Class

```
public class Vehicle {

    public static final int VERSION = 1; // class variable
    public String colour; // instance variable

    // Constructors
    public Vehicle() {
        this.colour = "Blue"; // Default colour for the Vehicle
    }

    public Vehicle(String colour) {
        this.colour = colour;
    }

    // Methods
    public void changeColour(String newColour) { // instance method
        this.colour = newColour;
    }

    public String getColour() { // instance method
        return colour;
    }

    public static int getVersion() { // static method
        return Vehicle.VERSION;
    }

}
```

#### Car Class

```
public class Car extends Vehicle {

    public static final int WHEELS = 4; // constant
    public static final int MAXGEARS = 5; // maximum number of gears
    public final double engineSize; // Engine size in litres, once initialised,
    becomes permanent
    public int currentGear = 0; // current gear

    // Constructors
    public Car(double size) {
        super();
        engineSize = size;
    }

    public Car(double size,String colour) {
        super(colour);
        engineSize = size;
    }

    // Methods
    public int getNumberWheels() {
        return Car.WHEELS;
    }

    public void changeUp() {
        if(currentGear<=MAXGEARS) {
            currentGear++; // increment the gear
        }
    }

    public void changeDown() {
        if(currentGear>=0) {
            currentGear--; // change down a gear
        }
    }

    public int getCurrentGear() {
        return currentGear;
    }

}
```

```
    }
    public double getSpeed() {
        if(getCurrentGear()==1) {
            return currentGear*5*engineSize;
        }
        else if(getCurrentGear()==2) {
            return currentGear*5*engineSize;
        }
        else if(getCurrentGear()==3) {
            return currentGear*5*engineSize;
        }
        else if(getCurrentGear()==4) {
            return currentGear*5*engineSize;
        }
        else if(getCurrentGear()==5) {
            return currentGear*5*engineSize;
        }
        else {
            return 0;
        }
    }
}
```

### Lorry Class

```
public class Lorry extends Vehicle {

    public static final int WHEELS = 6; // constant
    public static final int MAXGEARS = 10; // maximum number of gears
    public final double engineSize; // Engine size in litres, once initialised,
    becomes permanent
    public int currentGear = 0; // current gear

    // Constructors
    public Lorry(double size) {
        super();
        engineSize = size;
    }

    public Lorry(double size,String colour) {
        super(colour);
        engineSize = size;
    }

    // Methods
    public int getNumberWheels() {
        return Lorry.WHEELS;
    }

    public void changeUp() {
        if(currentGear<=MAXGEARS) {
            currentGear++; // increment the gear
        }
    }

    public void changeDown() {
        if(currentGear>=0) {
            currentGear--; // change down a gear
        }
    }

    public int getCurrentGear() {
        return currentGear;
    }

    public double getSpeed() {
        if(getCurrentGear()==1) {
            return currentGear*5*engineSize;
        }
        else if(getCurrentGear()==2) {
            return currentGear*5*engineSize;
        }
        else if(getCurrentGear()==3) {
            return currentGear*5*engineSize;
        }
    }
}
```

```
        else if(getCurrentGear()==4) {
            return currentGear*5*engineSize;
        }
        else if(getCurrentGear()==5) {
            return currentGear*5*engineSize;
        }
        else if(getCurrentGear()==6) {
            return currentGear*5*engineSize;
        }
        else if(getCurrentGear()==7) {
            return currentGear*5*engineSize;
        }
        else if(getCurrentGear()==8) {
            return currentGear*5*engineSize;
        }
        else if(getCurrentGear()==9) {
            return currentGear*5*engineSize;
        }
        else if(getCurrentGear()==10) {
            return currentGear*5*engineSize;
        }
        else {
            return 0;
        }
    }
}
```

### And finally, the Test Class

```
public class Test {

    public static void main(String[] args) {
        Car myCar = new Car(3.0); // creates a standard Car, the colour is defined in
                                // its super class (Vehicle)
        Lorry myLorry = new Lorry(1.0,"Green"); // creates a Green Lorry

        // Lets change the gears on the cars
        for(int i = 0; i<3; i++) {
            myCar.changeUp();
        }
        System.out.println("myCar's gear :"+myCar.getCurrentGear());

        for(int i = 0; i<7; i++) {
            myLorry.changeUp();
        }
        System.out.println("myLorry's gear :"+myLorry.getCurrentGear());

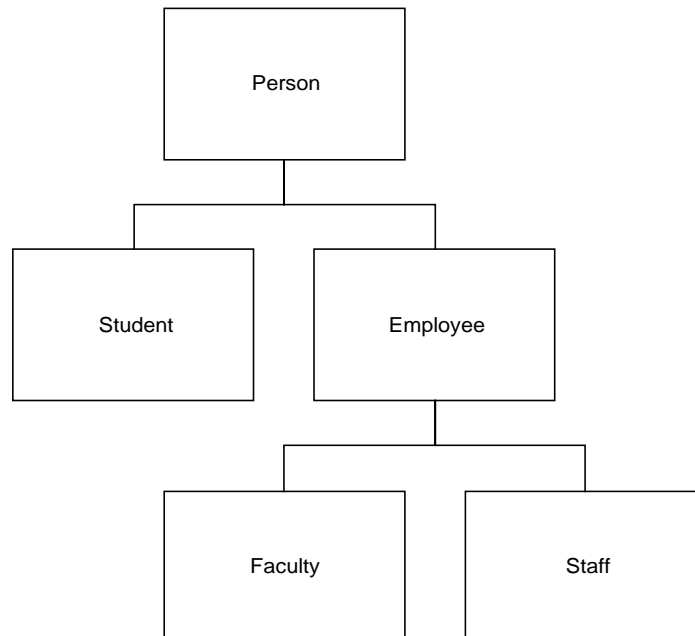
        System.out.println("The colour of myCar is "+myCar.getColour()+", its speed is
"+myCar.getSpeed()+"mph");
        System.out.println("The colour of myLorry is "+myLorry.getColour()+", its
speed is "+myLorry.getSpeed()+"mph");

        // The above expressions use an inherited method (getColour()) from the
        Vehicle
    }
}
```

## 8. Assignment

### 8.1 Aims

By using the Vehicle-Car-Lorry inheritance example as a basis, implement a program that describes the following relationship between objects:



Your program need not model the objects presented above perfectly, but rather demonstrate your newly acquired programming skills in Java. You will probably need a test class to create and output the components of your program.

### 8.2 Requirements

The software that you develop should demonstrate the following concepts that you have already encountered during the course of this tutorial:

#### 8.2.1 Program content

- Variables
- String
- Array
- Constants

#### 8.2.2 Program Flow

- If...then or Case statements
- For loops

#### 8.2.3 Special Features

- Class and member variables
- Class and member methods
- Encapsulation
- Inheritance

### 8.3 Scope

As you have not been introduced to retrieving keyboard input from the user, your program does not require you to do this. Your primary form of communication of things being done should be to use the familiar `System.out.println(thingy)` line.