# Unit 1b: Java (Revision)

# 1. Java Programming Language

Java SE 8 was released in March 2014 and introduced lambda expressions; Java SE 9 release was postponed a number of times but was released on September 21st 2017 and introduces a module system.

Java is not a pure OO language and incorporates procedural ideas such as sequence, selection and iteration. Currently it has limited support for the idea of modules. Some functional programming concepts are supported in Java 8.

Most of the basic features of Java were introduced in Programming 1 and can be categorised into two main forms: Program Implementation concepts; and, Object Orientation concepts.

# 2. Program Implementation Concepts

Program execution follows an iterative cycle of *Input-Process-Output* where information is supplied and processed to produce output. Input may be a user selection to initiate a range of actions e.g. a menu choice. Alternatively, input involves data supplied from the user or an external file/stream. Output is typically to a device such as a screen, printer or audio player; or, to a file/stream. Processing involves transforming the input to the required output. In memory storage is used to hold values which are input and values to be output.

## a. Variables

In-memory storage holds data in fast memory which is only available when the program is running. Memory locations are allocated to the program as they are required and withdrawn when they are no longer accessible or required. A named location/set of locations is called a **variable**.

Java is a typed programming language therefore a **data type** is required when defining a variable; this dictates the amount of memory to be allocated, the set of valid values which can be stored in that variable and the basic operations which can operate on the variable value.

The location of the variable definition defines the scope of meaning of the variable i.e. which code statements can legitimately query or update the variable value or, in other words, the lifetime of the variable's existence. For example, if a variable is defined inside a method then it only exists while the method code is executing and goes out of scope i.e. dies, when program control moves away from the method. The Java runtime system has a garbage collection facility which reclaims memory previously allocated to now out of scope variables. An object instance is a variable whose type is specified by its class definition.

### *Example*

```
int x;
char c;
float f;
boolean flag;
Employee employee;
```

Different types of variable exist and can be classified as follows:

### *i. Instance Variable (non-static field)*

These define the attributes of an object e.g. an **Employee** object could have instance variables: employee number, name, pay grade etc.

### *ii. Class Variable (static field)*

A class variable is associated with a class definition and not a particular object instance; instance variables are associated with a particular object. All object instances can access the single value of a class variable and, commonly, a class variable is used to hold a value which can be used when a new object instance is created e.g. the next available employee number value to be allocated to a new employee. Other code may also access a class variable dependent on the access modifier specified.

### *iii. Local Variable*

Variables should be defined only where they are needed i.e. the scope of definition should be as limited as possible to ensure incorrect values are localised to the code where they are used for debugging purposes. A method/block can define a variable which is only required local to that method or block and moves out of scope when that method/block finishes execution.

### *iv. Parameter*

This is used to specify an argument to a method. When the method is called appropriate variable values or literals should be supplied to each argument for the method code to use in its execution.

## b. Primitive Data Types (Java)

Java provides a number of primitive data types to allow single-value variables to be specified. There are facilities for collecting related values together, in fact, a class definition groups together instance variable definitions representing the attributes of an object instance – the class definition creates a new 'type' from which object instances can be created.

| Type | Size |
|---|---|
| byte | 8 bits |
| short | 16 bits |
| int | 32 bits |
| long | 64 bits |
| float | 32 bits floating point |
| double | 64 bits floating point |
| boolean | 1 bit |
| char | 16 bits |

The larger the number of bits, the greater the number of different values which can be stored in memory locations associated with a variable of a specific data type. Recall a bit is a binary digit which can either hold a 1 or a 0; therefore, with a byte type we have 8 bits each of which can be a 1 or a 0 – this gives $2^8$ or 256 different combinations. The combinations represent a range of values from $-2^7$ to $2^7-1$ i.e. -128 to 127.

In binary, this would be 00000000, 00000001, 00000010, … 11111111.

Note floating point numbers split their storage into bits allocated to a fraction (mantissa) and bits assigned to a power value (exponent).

Boolean values can either be true or false and hence a single bit is sufficient to indicate which is being stored.

There are 16 bits allocated to a char variable i.e. a single character. This gives 65,536 different combinations. As different languages use different character representations; a character code such as Unicode is used to map a character to its binary representation.

Strings are often treated as a primitive data type though they are, in fact, collections of characters. The **String** class (**java.lang.String**) defines the implementation of **String** objects. A **String** variable's value is enclosed within double quotes while a single character value uses single quotes e.g. 'A' and "Java".

### c. Methods

A method is a collection of variable definitions and code statements which can be referred to by a single name and provide a single piece of code functionality. The method may produce a single value of a specified return type and may require arguments to be supplied when its execution is invoked. The method body is a block of statements which may define local variables and can include, virtually, any other Java statement. A method which has no return value is declared as **void**.

Arguments which are specified of a primitive data type are passed by value i.e. the method has its own copy of the variable which it can manipulate but any changes are not propagated back to the calling method i.e. the original value is reverted to once the method stops executing and program control returns to the place where the method is called from.

Objects which are passed to a method are also passed by value, however, the attributes can be changed dependent on the access level specified i.e. a method cannot change a reference to an object but can change the object field values i.e. the object remains the same – uses the same memory locations – but the instance variable values can be changed permanently by the method.

A method groups together code which provides a single purpose: sometimes this is simply to make the code simpler and more readable i.e. to provide abstraction; alternatively, it can be a basic way of implementing modularity by constructing an application as a set of methods, each of which carries out a single piece of the overall functionality where a method is called when it is required during program execution. Methods are also used to define the ways in which object instances can be created and instance variables can be queried and updated.

#### *Example*

The method detailed below requires two arguments to be passed to it: an integer (*salary*) and a real number (*bonusPercentage*); and, returns a real value (*bonusPay*).

```
float calculateBonus(int salary, float bonusPercentage)
{
     float bonusPay = salary * bonusPercentage;
     return bonusPay;
}
```

The method is called using its name and supplying required values; typically a variable is declared to store the results of the method execution. For example:

```
int salary = 25000;
float bonusPercentage = 0.1;

float employeeOneBonusPay = calculateBonus(salary,
                                           bonusPercentage);

float employeeTwoBonusPay = calculateBonus(30000, 0.05);
```

### d. Statements, Blocks and Expressions

A statement is a complete unit of execution terminated by a semi-colon (;). A block is a group of statements enclosed by {} e.g. a method body, control flow path etc.

An expression combines variables, literals (constant values), operators and method invocations to produce a single value. Typically a statement assigns the value of the expression to a variable of an appropriate type. For example, above we have a method call to **calculateBonus()** which returns a value that is assigned to the **float** variable defined.

Alternatively, the expression value can be used in a conditional statement using a relational operation to compare the expression value to another value.

Operators provide the basic functionality to be performed on variables as part of expressions and assignment i.e. performing 'calculations' on variable values and storing the result in a variable.

An operator is applied to one or two variables/literals to produce a single new value which is often assigned to a variable or used in an expression.

With arithmetic operators the value calculated is dependent on the type of operands i.e. we can mix numbers but not numbers & non-numbers e.g. integers and real numbers. Operator precedence is from left to right with brackets used to override precedence.

#### *Arithmetic Operators*

| Operator | Meaning |
|---|---|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| / | Division |
| % | Remainder (Integer Division) |
| ++ | Increment |
| -- | Decrement |

#### *Assignment*

An assignment statement is used to store the value of a literal, variable value or expression in a variable. The = operator is used for this with the LHS specifying an appropriate variable for the value calculated/specified on the RHS e.g. **int x=5; x = x + 1;** .

Assigning a character or **String** value to a number variable fails.

The + operator can also be applied to **String** values for concatenation e.g.

```
String fullName = "Phil" + " Collins";
```

### *Relational Operators*

Relational operators are used in conditional statements i.e. statements which control the execution flow of a method: choose between statements/blocks or repeat blocks. They are used for comparison and produce a Boolean value – true or false. Generally used with numbers and single characters.

| Operator | Meaning |
|---|---|
| == | Equals |
| != | Not equals |
| < | Less than |
| <+ | Less than or equal |
| > | Greater than |
| >= | Greater than or equal |

### *Logical Operators*

Logical operators negate or combine relational expressions; overall a Boolean value should be produced.

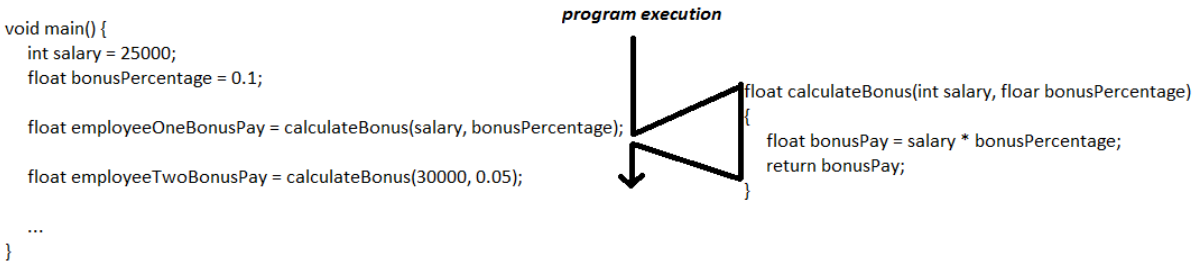
| Operator | Meaning |
|---|---|
| && | And |
| \|\| | Or |
| ! | Not |

### *instanceOf Operator*

The **instanceof** operator indicates whether an object is of a particular class.

## e. Control Flow

Java executes statements in a sequence starting from the first statement in the **main()** method. A method call jumps the execution to the start of the method body and returns to the statement after the calling statement once the method finishes executing.

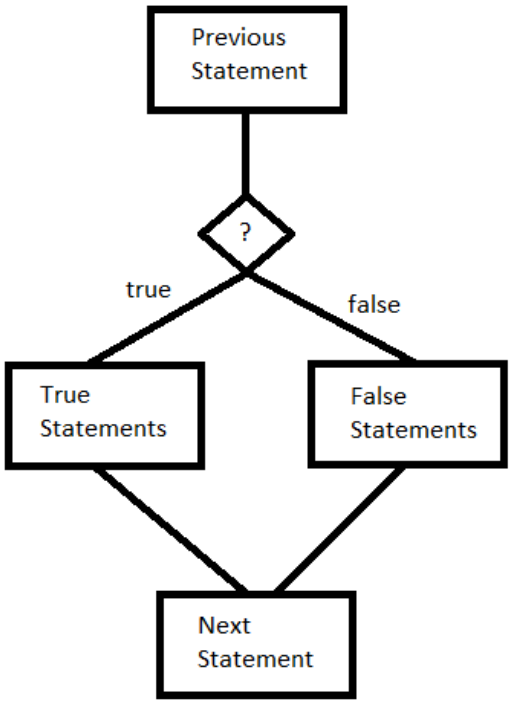



To provide alternate paths through an application Java provides selection control structures and to repeat statements Java provides iteration control structures.

### *Selection (if)*

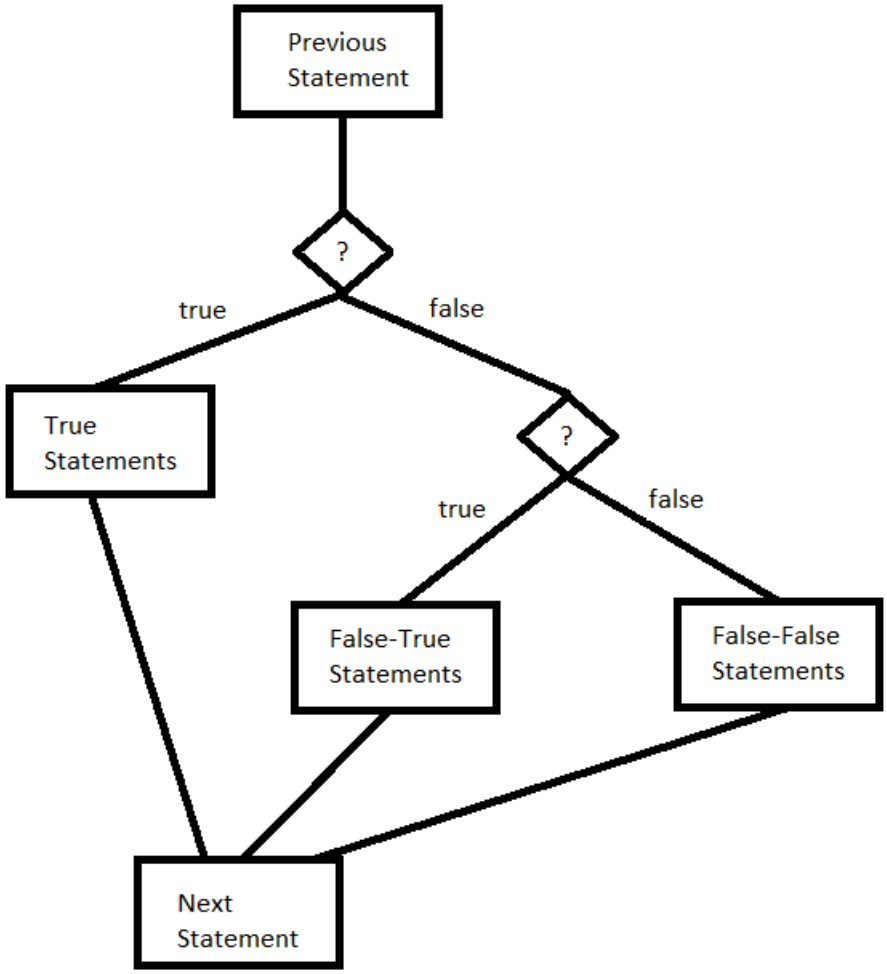
Binary selection is achieved using an **if** statement i.e. a choice between two alternatives. A condition is evaluated and statements grouped into a 'true path' and a 'false path' – commonly blocks of statements to be executed when the condition evaluates to **true** or **false**.

```
if (condition)
{
      // true-path
}
else
{
      // false-path
}
```




Multiple selection is achieved through using nested **if** statements i.e. further **if** statements inside the 'true' and/or 'false' paths.

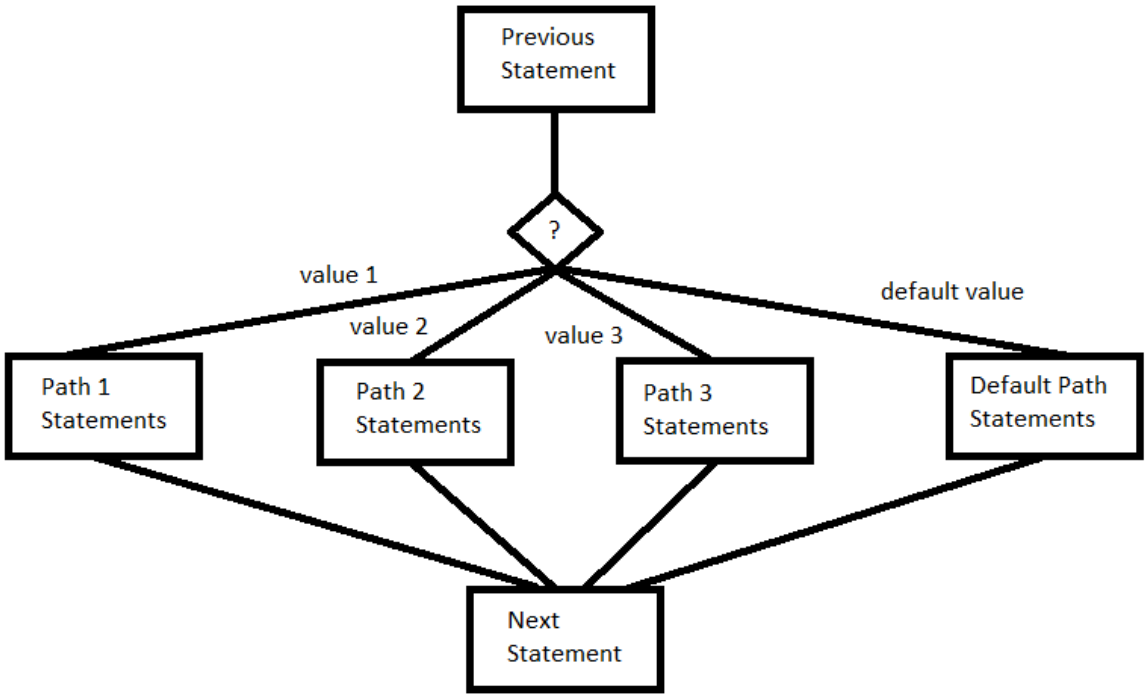
```
if (condition)
{
      // path-1
}
else
     if (condition-2)
     {
        // path-2
     }
     else
     {
        // path-3
     }
```




### *Selection (switch)*

Where paths can be labelled, a variable value is used to determine which path to execute and a **switch** statement used:

```
switch (caseVariable)
{
     case 1:
          // path-1
          break;
     case 2:
          // path-2
          break;
     default:
          // default-path
}
```




### *Iteration*

Iteration involves repeating a statement/block of statements a number of times. Java provides three mechanisms:

- **for** – used for a fixed number of iterations
- **while** – used when the loop statements may not be executed at all
- **do-while** – used when the loop body is executed at least once

In the first case the number of iterations must be determined before loop execution begins; often a counter is used to count iterations from a starting point to an end point. In the other cases a condition is evaluated to indicate repetition or termination of the loop.

#### Iteration (for)

Traditionally a **for** loop uses a variable with a starting value, an operation to modify the variable (in some form of sequence) and a terminating condition when the loop variable exceeds some end point:

```
for (loopVariable = startValue;
     loopVariableCondition;
     loopVariableOperation)
{
     // statements to be repeated
}
```

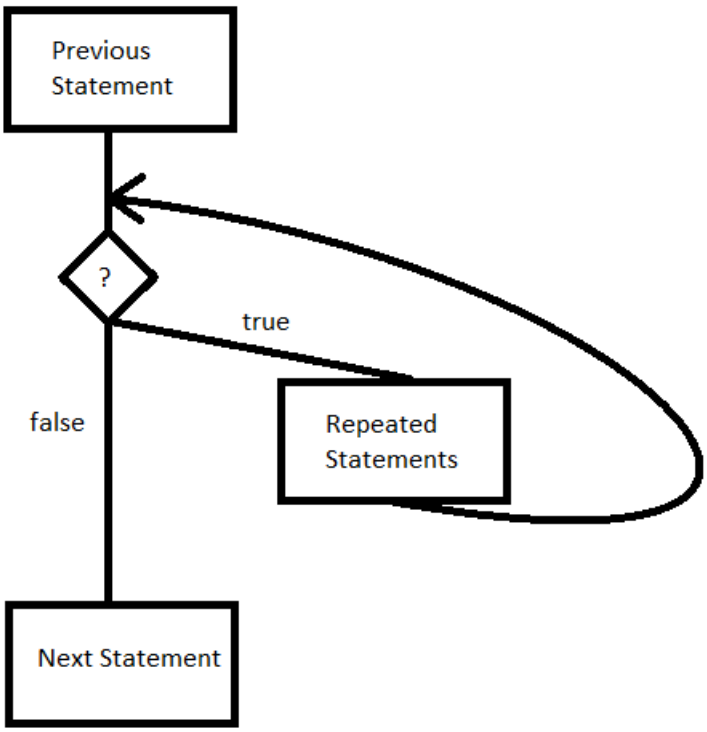
A newer form is used to iterate over a collection i.e. examine every element of the collection in turn:

```
for (loopVariable : collectionName) { ... }
```

### Iteration (while)

A pre-condition (**while**) loop will evaluate a condition before entering the block of statements; since the condition may fail on first evaluation the statement body may not be executed.

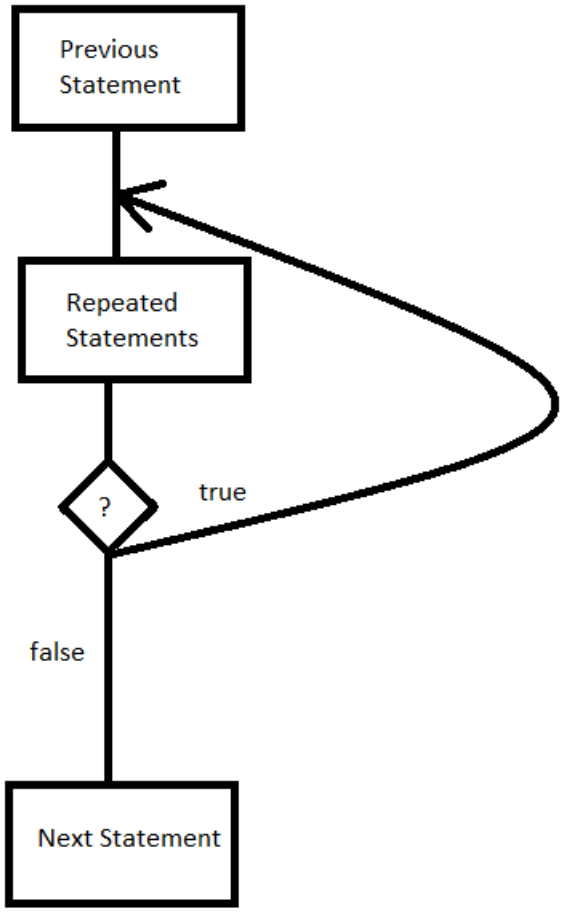
```
while (condition)
{
  // Statements to be repeated
}
```




### Iteration (do-while)

A post-condition (**do-while**) loop is where the statement body is executed at least once.

```
do {
     // Statements to be repeated
} while (condition);
```




All control structures may be nested inside each other provided they are 'opened' and 'closed' properly. However, too much nesting promotes complexity and is indicative that a method is attempting to specify more than a single piece of functionality.

## f. Data Structures

The basic facility available for collecting objects together is an array which is a fixed sized collection of objects – all of the same basic class where each individual element of the array is identified by a subscript or index value starting from 0.

### *Arrays*

A data structure groups together data which has a collective meaning under single variable. Arrays are fixed size collections of the same type of variables: either primitive values or objects.

Individual values are accessed through a subscript (index) value generally starting at 0 for the first element.

e.g. `int[] array_name` declares a variable of integer array type

A **new** operator is used to allocate the necessary memory locations

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

Initial values can be supplied at the time of creation

```
int[] array_name = {1, 2, 3, 4, 5}
```

The array has 5 elements and **`array_name[0]`** refers to the first element which has value 1 while **`array_name[4]`** has value 5.

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| **0** | **1** | **2** | **3** | **4** |

Multi-dimensional arrays have separate indexes for each dimension e.g. a two dimensional array could be declared as:

```
int[][] twoDArray = {{1, 2}, {3, 4}};
```

**`twoDArray[1][0]`** has value 3

| | | |
|---|---|---|
| **0** | 1 | 2 |
| **1** | 3 | 4 |
| | **0** | **1** |

The class **java.util.arrays** has a number of array manipulation methods including copying, sorting and searching.

## 3. Object Oriented Concepts

### a. Classes and Objects

Classes define a new 'type' which can then be used to declare variables i.e. object instances. Classes have field (attribute/instance variable) definitions which must include a type either, simple such as **int**, **float** etc., or complex such as a collection or another class.

An object instance uses its instance variables to store the data associated with that object. For example, an employee object can be defined of class **Employee**; the employee object can then store actual values for each of the attributes defined in the **Employee** class: the *employeeNumber* (E001); the *employeeName* (Andres Iniesta), the *departmentName* (Creative) etc.

Objects are created using the **new** operator applied to the Class of the object. While classes have default constructors built in, typically, the new operator runs a constructor (pseudo, class) method defined in the class supplying arguments which are used to set the value of the attributes of the object instance. As different forms of data may be available at the point of creation, a class will often specify more than one constructor to allow some or all of the attribute values to be supplied at time of creation.

The **new** operator creates an object instance variable with appropriate memory allocated for each of the instance variables i.e. attribute values and returns a reference to the object.

When no more references to the object exist in the application execution then a garbage collector reclaims the memory for future reallocation to new objects i.e. either the object moves out of scope or its value is set to null.

### b. Class

A class definition provides a 'template' or blueprint for creating objects with well-defined state & behaviour; defines an object 'type' to allow object variables to be created.

Class definitions specify fields (attributes) and their types; possibly including other objects (composition) and relationships (associations) with other objects.

Also specifies methods that an object instance requires i.e. defines the 'interface' to the rest of the application.

Java allows different types of classes including subclasses, abstract classes and local classes.

The convention is that a class name begins with a capital letter e.g. **`class ClassName {..}`**.

#### *Attributes/Fields/Instance Variables*

Attributes (fields) are defined of a specific data type or structure which specifies permissible values and interactions (operations) e.g.

```
int integerField;
```

```
int[] arrayField;
```

Access modifiers define scope of availability; typically these are **private** – i.e. the attribute can only be accessed/modified through **public** object methods – provides encapsulation.

The convention for attributes names is a lower case initial letter with no spaces and other words capitalised e.g. **`String firstName;`** .

While ideally attributes should be defined as **private** i.e. encapsulated within the object and methods declared as **public** i.e. available for other applications to use to query/update the state of an object; Java allows a number of other access modifiers to clarify the scope and meaning of attributes and methods.

The **this** keyword allows an object to refer to its own features e.g. attributes

```
this.x = x;
```

- it allows us to distinguish between the attribute x and the argument x supplied to a method.

#### *Methods*

The behaviour of a class (object) is defined through methods. Methods have parameters (arguments) which specify type and number of values expected to be supplied when a method is invoked (executed) on an object instance. A method may return a value and the type of that value is specified as the return type – again, simple or complex.

Methods are units of code which can be declared with up to 6 components: access modifier, return value type, method name, parameter list, exception list, and, method body (enclosed between {}) e.g. **`public void printDetails(..) {}`**.

The method signature is its name and parameter types.

Typically a method has a **public** access specifier if it is to be invoked by another object e.g. **`objectVariable.objectMethod();`**.

**private** methods are for internal use by the object itself.

### Constructors

Constructors are (pseudo, class) methods which are used to create object instances using the class definition. Java doesn't treat constructors as methods but we are going to categorize them as such for simplicity purposes. A constructor's name is the class name and it has no return type. Different constructors differ through their parameter list i.e. method signatures e.g. **`ClassName(){}`**, **`ClassName(int x){}`**, **`ClassName(int x, String y){}`**.

One constructor may invoke another using this e.g. **`this(x);`**.

The **new** operator is applied to invoke a constructor and create an object instance. For example, **`ClassName classObject = new ClassName();`**.

Although Java creates a default constructor, it is better to define your own to initialise object fields.

### Accessors (getters)

Accessors (getters) are methods used to access the object's attributes – one method per accessible attribute value e.g. **`public int getFirstField(){}`**.

### Mutators (setters)

Mutators (setters) are methods used to modify an object attribute e.g.

**`public void setFirstField(int firstField){}`**.

Note that not all attributes may be changeable from outside the object e.g. an id value and, therefore, not all attributes will require a setter method – a constructor is used to set the value of the attribute at time of creation and this is never changed.

### Other Methods

Other methods may be used to ask the object to create new values/objects e.g. **`toString()`** is often implemented to provide a String representing the current state of the object's attributes.

In Java classes can be broadly categorized as model classes representing data and application classes which provide the code functionality. Application classes will define methods based on the role the class plays in the architecture of the application while model classes will have constructors, accessors, setters and other methods as determined by the nature of the instance variables and their required manipulation.

## 4. Discussion Questions

1. Identify the issues with the following Java variable declarations and assignments.

```
i.    integer number = -2;
ii.   char letter = "*";
iii.  string name = "jAvA";
iv.   Boolean flag = 'false';
```

2. Trace the following code identifying the values of variables as they change.

```
int x = 5;
int y = x++;
int z = --y;
int t = x/y;
int u = x % z;
int v = x + y * z;
```

3. For each of the following scenarios below identify the most appropriate control flow constructs (if, switch, for, while, do-while).

a. Give all employees a 10% increase.

b. Display a menu, accept and process a choice until the user chooses to exit.

c. Find the largest value in an unsorted list of numbers.

d. Ensure an input number is within a specified range

e. Querying a database produces a list of results which may be empty – display the results.