Java

Java is a programming language and computing platform first released by Sun Microsystems in 1995

Compiler - A compiler translates a computer program written in a human-readable computer language (like Java) into a form that a computer can **execute**. You have probably seen EXE files on your computer. These EXE files are the output of compilers. They contain **executables** -- machine-readable programs translated from human-readable programs.

How Java works

The java compiler takes a .java file and generates a .class file

- The .class file contains Java bytecodes, the assembler language for Java programs
- Bytecodes are executed in a JVM (java virtual machine),the valid bytecodes are specified by Sun

The JVM interprets the bytecodes

- JVM is platform/OS specific, must ultimately run the code

First Java Program:

```
class Hello
{
  public static void main ( String[] args )
  {
    System.out.println("Hello World!");
  }
}
```

shows where the program starts running. The word main means that this is the **main method** — where the Java virtual machine starts running the program. The main method must start with this line, and all of its parts must be present. Wherever there is one space it is OK to have any number of spaces. The spaces surrounding the parentheses are not required. The fifth line shows parentheses not surrounded by spaces (but you could put them in).

The main method of this program consists of a single statement:

```
System.out.println("Hello World!");
```

This statement writes the characters inside the quotation marks on one line of the monitor and then moves on to the next line. (The quotation marks are not written.)

statement

A **statement** in a programming language is a command for the computer to do something. It is like a sentence of the language. A statement in Java is followed by a semicolon.

Methods are built out of statements. The statements in a method are placed between braces { and } as in this example.

Comments

```
// Write first line comments
```

```
/* Program 1
Write out one than one lines of a comments
The comments describes definition of code
*/
```

Data Types

Integer Primitive Data Types		
Type	Size	Range
byte	8 bits	-128 to +127
short	16 bits	-32,768 to +32,767
int	32 bits	-2 billion to +2 billion (approximately)
long	64 bits	-9E18 to +9E18 (approximately)

boolean

Another of the primitive data types is the type boolean. It is used to represent a single true/false value.
A boolean value can have only one of two values:

true false

Character Literals

A character literal is a single character with an apostrophe on each side:

'm' 'y' 'A'

Each of these represents a *single* control character.

"Hello"

This is a **String,** which is not primitive data. It is, in fact, an object. Strings are surrounded by double quote marks ", not by apostrophes.

Objects

An object can be a variable, function, or data structure. In the object-oriented programming paradigm, "object" refers to a particular instance of a class where the object can be a combination of variables, functions, and data structures.

- An object is a big block of data. An object may use many bytes of memory.
- An object usually consists of many internal pieces.

Variables and Assignment Statements

variable — a named location in main memory which uses a particular data type to hold a value.

Declaration of a Variable

```
class Example
{
  public static void main ( String[] args )
  {
    int age;
    age = 20;    //the declaration of the variable

    System.out.println("The variable contains: " + age );
  }
}
```

Syntax of Variable Declaration

There are several ways to declare variables:

```
dataType   variableName;
dataType   variableName  =  initialValue ;
dataType   variableNameOne, variableNameTwo ;
```

Rules of Variables

The programmer picks a name for each variable in a program. Various things in a program are given names. A name chosen by a programmer is called an **identifier**. Here are the rules for identifiers:

- Use only the characters 'a' through 'z', 'A' through 'Z', '0' through '9', character '_', and character '$'.
- An identifier can not contain the space character.
- Do not start with a digit.
- An identifier can be any length.
- Upper and lower case count as different characters.
- An identifier can not be a reserved word.
- An identifier must not already be in use in this part of the program.

- Local variables
- Instance variables
- Class/static variables

Local variables:

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.
- Local variables are visible only within the declared method, constructor or block.
- There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.

Example:

Here, *age* is a local variable. This is defined inside *myage()* method and its scope is limited to this method only.

```
public class Test{
  public void myage(){
    int age = 21;
    age = age + 7;
    System.out.println(" my age is : " + age);
  }
 public static void main(String args[]){
    Test testmyage= new Test();
    testmyage.myage();
  }
}
```

Instance variables:

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
- The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However visibility for subclasses can be given for these variables with the use of access modifiers.
- Instance variables have default values. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor.

- Instance variables can be accessed directly by calling the variable name inside the class. However within static methods and different class (when instance variables are given accessibility) should be called using the fully qualified name . ObjectReference.VariableName.

```
import java.io.*;

public class SitsEmployee{
  // this instance variable is visible for any child class.
  public String name;

  // salary  variable is visible in Employee class only.
  private double salary;

  // The name variable is assigned in the constructor.
  public SitsEmployee (String empName){
    name = empName;
  }

  // The salary variable is assigned a value.
  public void setSalary(double empSal){
    salary = empSal;
  }
  // This method prints the employee details.
  public void printEmp(){
    System.out.println("name  : " + name );
    System.out.println("salary :" + salary);
  }
  public static void main(String args[]){
    Employee empOne = new Employee("John");
    empOne.setSalary(1000);
    empOne.printEmp();
  }
```

```
}
```

Class/static variables:

- Class variables also known as static variables are declared with the *static* keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final and static. Constant variables never change from their initial value.
- Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally values can be assigned in special static initializer blocks.
- Static variables can be accessed by calling with the class name . *ClassName.VariableName*.
- When declaring class variables as public static final, then variables names (constants) are all in upper case. If the static variables are not public and final the naming syntax is the same as instance and local variables.

Example:

```
import java.io.*;
public class Employee{
  // salary  variable is a private static variable
  private static double salary;
  // DEPARTMENT is a constant
  public static final String DEPARTMENT = "Development ";
  public static void main(String args[]){
    salary = 1000;
    System.out.println(DEPARTMENT+"average salary:"+salary);
  }
}
```

Assignment Statements

Assignment statements look like this:

```
variableName  =  expression ;
age= 20;
```

Arithmetic Operators

Operator	Meaning	precedence
-	unary minus	highest
+	unary plus	highest
*	multiplication	middle
/	division	middle
%	remainder	middle
+	addition	low
-	subtraction	low

The Relational Operators:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

The Logical Operators:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

Difference between AND and OR

A	B	A && B	A || B
F	F	F	F
F	T	F	T
T	F	F	T
T	T	T	T

```
// check that the weight is within range
  if ( sitsscore >= 80 && sitsscore <= 90 )
    System.out.println(" acceptable range!" );
  else
     System.out.println("Unaccepted range." );
```

Expression

An **expression** is a combination of literals, operators, variable names, and parentheses used to calculate a value.

(32 - y) / (x + 5)

Boolean Expressions

A **boolean expression** is an expression that evaluates to *true* or *false*.

- The operator for "equal" is == (two equal signs in a row). In your web browser it may be hard to see that there are two equal signs.
- The operator for "not equal" is != (exclaimation point equal sign).

It is easy to forget these two details, and easy to overlook these details in a program. You might spend hours debugging a program because a = was used where a == belongs.

Class

Java class is nothing but a template for object you are going to create or it's a blue print by using this we create an object. In simple word we can say it's a specification or a pattern which we define and every object we define will follow that pattern.

Creating an Object

```
class Stringtest1
{
  public static void main ( String[] args )
  {
    String str ;

    str = new String( " Welcome to SITS training!" );
  }
}
```

The declaration

String str;

creates a **reference variable**, but does not create a String object. The variable str is used to refer to a String after one has been created.

method

A method is a set of code which is referred to by name and can be called (invoked) at any point in a program simply by utilizing the method's name

Running a Method

```
class Stringtest2
{
  public static void main ( String[] args )
  {
    String str;
    int    len;
    str = new String( "Welcome to sits training" );
    len = str.length();    // call the length() method of the object
```

```
    System.out.println("The length is: " + len );

  }
}
```

Calling a method means asking a method to run. This program *called* thelength() method. (Sometimes the phrase **invoking a method** is used to mean the same thing.)

Declaring a Reference Variable

There are several ways to declare a reference variable:

ClassName variableName;

- This declares a reference variable and declares the class of the object it will later refer to. No object is created.

ClassName variableName = new ClassName(parameter, parameter, ...) ;

- This declares a reference variable and declares the class of the object. But now, at run time, a new object is constructed and a reference to that object is put in the variable. Sometimes parameters are needed when the object is constructed.

ClassName variableNameOne, variableNameTwo ;

- This declares *two* reference variables, both potentially referring to objects of the same class. No objects are created. You can do this with more than two variables, if you want.

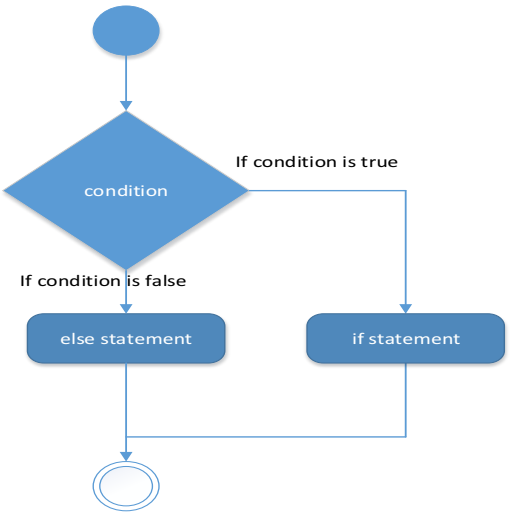
ClassName variableNameOne = new ClassName(parameter, parameter,..),
variableNameTwo = new ClassName(parameter, parameter, ...) ;

- This declares *two* reference variables. At run time, two objects are created and their references are assigned to the variables. Again, you can do this for more than two as long as you follow the pattern.

Packages

A **package** is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of **packages** as being similar to different folders on your computer. You might keep HTML pages in one folder, images in another, and scripts or applications in yet another.

Two-way Decisions




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

       int testscore = 76;

       if (testscore >= 90)
       {
      System.out.println("pass");
       }
      else
       {
      System.out.println(" fail");
       }
       }
```

The while and do-while Statements

The while statement continually executes a block of statements while a particular condition is true. Its syntax can be expressed as:

```
while (expression) {
    statement(s)
}
```

The while statement evaluates *expression*, which must return a boolean value. If the expression evaluates to true, the while statement executes the *statement*(s) in the while block. The while statement continues testing the expression and executing its block until the expression evaluates to false.

```
class Whiletest {
   public static void main(String[] args){
      int count = 1;
      while (count < 11) {
         System.out.println("Count is: " + count);
         count++;
      }
   }
}
```

The Java programming language also provides a do-while statement, which can be expressed as follows:

```
do {
    statement(s)
} while (expression);
```

The difference between do-while and while is that do-while evaluates its expression at the bottom of the loop instead of the top.

```
class DoWhiletest {
   public static void main(String[] args){
      int count = 1;
      do {
         System.out.println("Count is: " + count);
         count++;
      } while (count < 11);
   }
}
```

The for Statement

The for statement provides a compact way to iterate over a range of values. Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied. The general form of the for statement can be expressed as follows:

```
for (initialization; termination;
    increment) {
   statement(s)
}
```

When using this version of the for statement, keep in mind that:

- The *initialization* expression initializes the loop; it's executed once, as the loop begins.
- When the *termination* expression evaluates to false, the loop terminates.
- The *increment* or *decrement expression* is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment *or* decrement a value.

```
class Fortest {
   public static void main(String[] args){
      for(int i=1; i<11; i++){
         System.out.println("Count is: " + i);
      }
   }
}
```

Constructors

When you create a new instance (a new object) of a class using the new keyword, a *constructor* for that class is called. Constructors are used to initialize the instance variables (fields) of an object. Constructors are similar to methods, but with some important differences.

- **Constructor name is class name**. A constructors must have the *same name as the class its in*.
- **Default constructor**. If you don't define a constructor for a class, a *default parameterless constructor* is automatically created by the compiler. The default constructor calls the default parent constructor (super()) and initializes all instance variables to default value (zero for numeric types, null for object references, and false for booleans).
- **Default constructor is created only if there are no constructors**. If you define *any* constructor for your class, no default constructor is automatically created.
- **Differences between methods and constructors**.
    - There is *no return type* given in a constructor signature (header). The value is this object itself so there is no need to indicate a return value.
    - There is *no return statement* in the body of the constructor.
    - The *first line* of a constructor must either be a call on another constructor in the same class (using this), or a call on the superclass constructor (using super). If the first line is neither of these, the compiler automatically inserts a call to the parameterless super class constructor

```
public class Person {
  private String firstName;
  private String lastName;
  private String address;
```

```
  private String username;
  // The constructor method
  public Person(String personFirstname, String personLastName, String personAddress, String
personUsername)
  {
   firstName = personFirstName;
   lastName = personLastName;
   address = personAddress;
   username = personUsername;
  }
  // A method to display the state of the object to the screen
  public void displayPersonDetails()
  {
   System.out.println("Name: " + firstName + " " + lastName);
   System.out.println("Address: " + address);
   System.out.println("Username: " + username);
  }
}
```

The toString() Method

```
public String toString();   // returns character data that can be printed
  |     |       |
  |     |       +--- this is the method name.  It takes no parameters.
  |     |
  |     +--- this says that the method returns a String object
  |
  +--- anywhere you have a Point object, you can use this method
```

Syntax of a Class Definition

Class definitions look like this:

```
class ClassName
{
 Descriptions of the instance variables and methods each object
 will have and the constructors that initialize a new object.
}
```

Often programmers separate the definition into three sections:

```
class ClassName
{
```

```
 // Description of the variables.

 // Description of the constructors.

 // Description of the methods.
}
```

Method Definition

Method definitions look like this:

```
returnType  methodName( parameterList )
{
 // Java statements
 return returnValue;
}
```

The returnType is the type of value that the method hands back to the caller of the method. Methods in classes you define can return values just as do methods from library classes. The **return** statement is used to hand back a value to the caller.

If you want a method that does something, but does not return a value to the caller, use a return type of **void** and do not use a return value with the **return** statement. The**return** statement can be omitted; the method will automatically return to the caller after it executes. Here is the method from the example program:

```
// method definition
void speak()
{
  System.out.println("Hello from an object!");
}
```

Array

An **array** is an object that is used to store a list of values. It is made out of a contiguous block of memory that is divided into a number of **cells**. Each cell holds a value, and all the values are of the same type. Sometimes the cells of an array are called *slots*.

Arrays are Objects

An array is an object, and like any other object in Java, it is constructed out of main storage as the program is running. The array constructor uses different syntax than other object constructors:

```
new type[ length ]
```

This names the type of data in each cell and the number of cells.

Once an array has been constructed, the number of cells it has does not change. Here is an example:

```
int[] data = new int[10];
```

Inheritance

Inheritance enables you to define a new class based upon an existing class. The new class is similar to the existing class, but has additional member variables and methods. This makes programming easier because you can build upon an existing class instead of starting out from scratch

The class that is used to define a new class is called a **parent** class (or superclass or base class.) The class based on the parent class is called a **child**class (or subclass or derived class.)

```
class Car
{
        public void start()
        {
                System.out.println("start the car");
        }
        public void stop()
        {
                System.out.println("stop the car");
        }
}
class porsche extends Car //class new class name extends base class name
{
        public void special_features()
        {
                System.out.println("special features of porsche");
        }
}
class ford extends porsche
{
        public void special()
        {
                System.out.println("spl features of ford");
        }
}
public class Inheritance1 {
        public static void main(String[] args) {
        porsche m = new porsche();
        m.start();
        m.stop();
        m.special_features();
        System.out.println("*****************************************");
        ford f = new ford();
        f.start();
        f.stop();
        f.special_features();
        f.special();

        } }
```

Polymorphism

Polymorphism means "having many forms." In Java, it means that a single variable might be used with several objects of related classes at different times in a program.

Polymorphism is the capability of a method to do different things based on the object that it is acting upon. In other words, polymorphism allows you define one interface and have multiple implementations.

- It is a feature that allows one interface to be used for a general class of actions.
- An operation may exhibit different behavior in different instances.
- The behavior depends on the types of data used in the operation.
- It plays an important role in allowing objects having different internal structures to share the same external interface.
- Polymorphism is extensively used in implementing inheritance.

```
public class automationtool
{
 public static void main ( String[] args )
 {
  autotool autot = new qtpautomation( "QTP" );
  autot.automation();  //Invoke qtp automation()
  autotool autot = new seleniumautomation( " slenium" );
  autot.automation();  //Invoke selenium automation()
  autotool autot = new almautomation( "alm" );
  autot.automation();  //Invoke ALM automation
 }
}
```