**JAVA**

Java is a high-level programming language.

Java runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.

**Java Installation**

Reference Tutorial:

1. <http://javabeginnerstutorial.com/core-java-tutorial/java-basicsgetting-started-with-java/>

2. <https://www.tutorialspoint.com/java/index.htm>

**STEPS:**

**1. Download Java.**

Java SE Development Kit (jdk).

Link: <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

**2. Install Java.**

**3. Setting up the Environment Variables**

CLASSPATH: This environment variable points to the location of **JDK** home directory. It also contains the address of folder from where jars get loaded by ClassLoader.

JAVA\_HOME: This environment variable will point to the location of **Java** home directory.

(Or)

Assuming you have installed Java in c:\Program Files\java\jdk directory −

Right-click on 'My Computer' and select 'Properties'.

Click the 'Environment variables' button under the 'Advanced' tab.

Now, alter the 'Path' variable so that it also contains the path to the Java executable. Example, if the path is currently set to 'C:\WINDOWS\SYSTEM32', then change your path to read 'C:\WINDOWS\SYSTEM32;c:\Program Files\java\jdk\bin'.

**How to check if Java is installed**

Open Command Prompt, type **java -version** and hit enter. If your Java is installed properly and all environment variables are configured correctly it will show the version of Java installed.

**4. First Java Program.**

Sample Program: Refer "FirstProgramme.java" file.

Note: Name of the file should be same as the name given to public class.

**5. To compile a Java application**

Open the command prompt and change the directory where your file is saved.

**Example:**

C:\Users\540719>cd D:\Nithi\Java

D:\Nithi\Java>javac FirstProgramme.java

**Note:**

If the java file is compiled properly the compiler will create a class file for the sourceJava. It will be saved in the same location as the source file. Since no package is declared in the given code the .class file will be created in the same folder location.

Notice the difference for Java files using package declaration.

**Refer:** “FirstProgrammeWithPackage.java" file.

**Compile:** javac -d . FirstProgrammeWithPackage.java

It will create the class file in a corresponding package (com.jbt).

**6. To run a Java application**

**Without Package:**

D:\Nithi\Java>java FirstProgramme

Hello JBT!

**With Package:**

D:\Nithi\Java>java com.jbt.FirstProgrammeWithPackage

**(Or)**

D:\Nithi\Java>java com/jbt/FirstProgrammeWithPackage

Hello JBT!

**Note:**

The “javac” command uses the class file name with its extension (.java).

The “java” command uses the class file name without its extension (.class).

**JDK (Java Development Kit)**

JDK contains everything that will be required to develop and run Java application.

**JRE (Java Run time Environment)**

JRE contains everything required to run Java application which has already been compiled. It doesn’t contain the code library required to develop Java application.

**JVM (Java Virtual Machine)**

JVM is a virtual machine which work on top of your operating system to provide a recommended environment for your compiled Java code. JVM only works with bytecode. Hence you need to compile your Java application(.java) so that it can be converted to bytecode format (also known as the .class file). Which then will be used by JVM to run application. JVM only provide the environment it needs the Java code library to run applications.

**Java Portability**

In order to understand portability in Java you need to understand what happens to java code from start to finish.

* Java Source Code (Written by Developer) (Machine Neutral)
* Compiled Code / Byte Code (Compiled by javac) (Machine Neutral)
* Byte Code executed (Executed by JVM) (Machine Specific)

In step 2, javac (Java Compiler) converts Java code to byte code. This can be moved to any machine (Windows / Linux) and executed by JVM. JVM reads the bytecode and generates machine specific code. In order to generate machine specific code JVM needs to be machine specific. So every type of Machine (Windows / Linux / Mac) has a specific JVM. So in this way the coder doesn’t need to bother with generating byte code. JVM takes care of portability. So the final answer is Java is Portable but JVM is Machine Specific.

**Basic Syntax:**

About Java programs, it is very important to keep in mind the following points.

* **Case Sensitivity** − Java is case sensitive, which means identifier Hello and hello would have different meaning in Java.
* **Class Names** − For all class names the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.

Example: class MyFirstJavaClass

* **Method Names** − All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.

Example: public void myMethodName()

* **Program File Name** − Name of the program file should exactly match the class name.

When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and the class name do not match, your program will not compile).

Example: Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as 'MyFirstJavaProgram.java'

* **public static void main(String args[])** − Java program processing starts from the main() method which is a mandatory part of every Java program.

**Public:** is an Access Modifier, which defines who can access this Method. Public means that this Method will be accessible by any Class (If other Classes are able to access this Class.).

**Static:** is a keyword which identifies the class related thing. This means the given Method or variable is not instance related but Class related. It can be accessed without creating the instance of a Class.

**Void:** is used to define the Return Type of the Method. It defines what the method can return. Void means the Method will not return any value.

**main:** is the name of the Method. This Method name is searched by JVM as a starting point for an application with a particular signature only.

**String args[]:** is the parameter to the main Method.

**Java Identifiers:**

* All Java components require names. Names used for classes, variables, and methods are called identifiers.
* In Java, there are several points to remember about identifiers. They are as follows −
* All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (\_).
* After the first character, identifiers can have any combination of characters.
* A key word cannot be used as an identifier.
* Most importantly, identifiers are case sensitive.
* Examples of legal identifiers: age, $salary, \_value, \_\_1\_value.
* Examples of illegal identifiers: 123abc, -salary.

**Java Modifiers:**

Like other languages, it is possible to modify classes, methods, etc., by using modifiers. There are two categories of modifiers −

* Access Modifiers − default, public, protected, private
* Non-access Modifiers − final, abstract, strictfp

**JAVA VARIABLES:**

In Java, objects store their states in variables. Variables are used as containers to hold values (int, long, string…) during the life cycle of an application.

Example: public final int var;

Java supports the below mentioned primitive data types:

byte, short, int, long, float, double, char, boolean

**Variables Types in Java**

Variables in Java can be defined anywhere in the code (inside a class, inside a method or as a method argument) and can have different modifiers. Depending on these conditions variables in Java can be divided into four categories.

**I. Local Variables (Method Local)**

Variables defined inside methods, constructors or blocks are called local variables. The scope of local variables is only inside the Method, which means local variables cannot be accessed outside that Method.

Rule for Local Variable

* Local variables cannot use any of the access level (**Access Modifier**) since its scope is only inside the method.
* **Final** is the Only Non Access Modifier that can be applied to a local variable.
* Local variables don’t get a default value, hence local variables need to be initiated before they can be used.

**II. Class Variables (Static Variables)**

Class variables are variables declared within a class, outside any method, with the **static** keyword. They are known as Class level variables because values of these variables are not specific to any instance but are common to all instances of a class. Such variables will be shared by all instances of an Object.

**III. Instance Variables (Non-static Variables)**

Instance variables are used by Objects to store their states. Instance variables are variables declared within a class but outside any method, without the **static** keyword. Such variables are called instance variables because their values are instance specific and values of these variables are not shared among instances.

These variables are initialized when the class is instantiated. (i.e.) 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 can be accessed from inside any method, constructor or blocks of that particular class.

Rules for Instance variable

* Instance variables can use any of the four access levels. (i.e.) **Public**, **Private**, **Protected** all 3 access modifiers can be applied to Instance Variable(**Default** also).
* Instance variable will get default value means instance variable can be used without initializing it. Same is not true for Local Variable.
* Instance Variable can be marked **final**, **transient**.
* Instance Variable can not be **abstract**.
* Instance Variable can not have **synchroonized** modifier.
* Instance Variable can not have **strictfp** modifier.
* Instance Variable can not have **native** modifier.
* Instance Variable can not have **Static** modifier as it will becomes Class level variable.

**IV. Parameters**

Parameters are variables that are passed in Methods. For example, String args[] variables in the main Method is a parameter.

**Example:** public static void main(String args[]) { System.out.println("Hello"); }

**JAVA - OBJECT AND CLASSES**

**1. Class**

Class is a template for creating objects which defines its state and behavior. A class contains field and method to define the state and behavior of its object.

**Syntax:**

<Access Modifier> class <Class\_Name> extends <Super\_Class\_Name> implements <Interface\_Name>

**2. Object**

An object is an instance of a class. Objects have states and behaviors.

Object implements the state and behavior in the form of variables and methods and requires some memory allocated.

Example: A dog has states - color, name, breed as well as behavior such as wagging their tail, barking, eating.

To create an Object of a Class <**new**> **Keyword** can be used.

**Syntax:**

<Class\_Name> ClassObjectReference = new <Class\_Name>();

**Example:**

HelloWorld obj = new HelloWorld ();

**2.1 How to Access the Member of a Class**

You will call the Method of an Object by naming the Object followed by a period (dot), followed by the name of the Method and its argument list.

**Example:** objectName.methodName(arg1, arg2, arg3)

**BULLET POINTS**

* Class can have only public and default access.
* The public class needs to be in same name java file.
* A single java file can contain more than one non-public class but can have only one public class.
* Import and package statements should be applied to all the classes in the same source code file.
* A **public class** can be **seen** by all **classes** from **all packages**.
* A class with **default access** can be **seen** only by **classes** within the **same package**.
* A Java file with no public class have no naming restriction.
* The Class can also have final, abstract and strictfp non access modifiers.
* An Abstract Class cannot be instantiated.
* A Final Class cannot be sub-classed.
* A Class cannot be both final and abstract.
* Class visibility can be seen in 3 parameter

If a Class can extend another Class?

If a Class can create instance of another Class?

If a Class can access methods and variable of another Class?

**CONSTRUCTORS**

Constructors in Java can be seen as Methods in a Class. But there is a big difference between Constructors and Methods. These differences are defined in terms of purpose, syntax and invocation.

The main rule of constructors is that they should have the same name as the class. A Constructor’s syntax does not include a return type, since Constructors never return a value. A class can have more than one constructor.

Constructors have only one purpose, to create an Instance of a Class. This instantiation includes memory allocation and member initialization (Optional).

**Invocation of Constructor**

There is a difference between how Constructors and Methods are called. Constructors cannot be called explicitly, the Constructor will be invoked implicitly when the instance of the Class is generated

**Example:** ConstructorExample.java

**Constructor in Java Rules**

* A Constructor cannot have a return type.
* Constructor must have the same name as that of the Class.
* Constructors cannot be marked static
* Constructor cannot be marked abstract
* A Constructor cannot be overridden.
* A Constructor cannot be Final.

**Constructor Overloading:**

Like Methods, Constructors can also be overloaded. Since the Constructors in a Class, all have the same name as the Class, their signatures are differentiated by their parameter lists.

It is possible to use this() construct, to implement local chaining of Constructors in a class. The this() call in a Constructor invokes the other Constructor with the corresponding parameter list within the same Class. Java requires that any this() call must occur as the first statement in a constructor.

**Cheatsheet**

* Constructor is invoked when a new **object** is **created**.
* Constructor’s **can** also be **overloaded** but it **can not** be **overridden**.
* Every class has at least one constructor. If user doesn’t provide any JVM will provide a default no arg constructor.
* Abstract class also **has** constructor.
* Constructor must have the same name as class.
* Constructor can’t have a return type.
* If a method with the same name as class has return type will be treated as normal member method and not constructor.
* Constructor can have **any** **access modifier(All)**.
* Default constructor is a no arg constructor which calls the no arg constructor of super class. In case super class doesn’t have any no arg constructor then it will throw a run time exception.
* In case where a class has default constructor, its super class needs to have a no arg constructor.
* First statement of a constructor can be either **this** or **super** but can not be both at the same time.
* If coder doesn’t write any **this** or **super** call then compiler will add a call to super.
* Instance member can be accessible only after the super constructor runs.
* Interfaces do not have constructors.
* Constructor are **not inherited**. Hence can not be overridden.
* Constructor can not be directly invoked. It will be invoked(**Implicitly**) when a new object is created or a call by other constructor.

**ACCESS MODIFIERS:**

Access modifiers help you set the level of access you want for your Class, variables as well as Methods.

There are three access modifiers. Not including default access modifier. Default is an access control which will be set when one does not specify any access modifier.

**Access Control:**

* Public - Visible to the world.
* Private - Visible to the class only.
* Protected - Visible to the package and all subclasses.
* Default - Visible to the package, the default. No modifiers are needed.

Access modifiers (Some or All) can be applied on **Class, Variable** and **Methods**.

**I. Access Modifiers for Class**

Classes in Java can use only **public** and **default** access modifiers.

**Public:** When set to public, the given Class will be accessible to all the classes available in the Java world.

**Default:** When set to default, the given Class will be accessible to the classes which are defined in the same package.

**Java Access Modifiers Table for Class**

|  |  |  |
| --- | --- | --- |
| **Visibility** | **Public Access Modifier** | **Default Access Modifier** |
| Within Same Package | Yes | Yes |
| From Outside the Same Package | Yes | No |

**II. Access Modifiers for Instance & Static Variables**

Variables are eligible for all of the above mentioned modifiers.

**Note\*:** Visibility of the Class should be checked before checking the visibility of the variable defined inside that Class. If the Class is visible then the variables defined inside that Class will be visible. If the Class is not visible then no variable will be accessible, even if it is set to public.

**Default**

If a variable is set to default, it will be accessible to the classes which are defined in the same package. Any method in any Class which is defined in the same package can access the variable via Inheritance or Direct access.

**Public**

If a variable is set to public it can be accessible from any class available in the Java world. Any Method in any Class can access the given variable via Inheritance or Direct access.

**Protected**

If a variable is set to protected inside a Class, it will be accessible from its sub classes defined in the same or different package only via Inheritance.

**Note:** The only difference between protected and default is that protected access modifiers respect class subclass relation while default does not.

**Private**

A variable if defined private will be accessible only from within the Class in which it is defined. Such variables are not accessible from outside the defined Class, not even in its subclass.

**Java Access Modifiers Table for Variable**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Visibility** | **Public** | **Private** | **Protected** | **Default** |
| Within Same Class | Yes | Yes | Yes | Yes |
| From Any Class in Same Package | Yes | No | Yes | Yes |
| From Any Sub Class in Same Package | Yes | No | Yes | Yes |
| From Any Sub Class from Different Package | Yes | No | Yes(**Only By Inheritance**) | No |
| From Any Non Sub Class in Different Package | Yes | No | No | No |

**III. Access Modifiers for Methods**

Methods are eligible for all of the following modifiers.

**Default**

When a Method is set to default it will be accessible to the classes which are defined in the same package. Any Method in any Class which is defined in the same package can access the given Method via Inheritance or Direct access.

**Public**

When a Method is set to public it will be accessible from any Class available in the Java world. Any Method in any Class can access the given method via Inheritance or Direct access depending on Class level access.

**Protected**

If a Method is set to protected inside a Class, it will be accessible from its sub classes defined in the same or different package.

**Note:\*** The only difference between protected and default is that protected access modifiers respect class subclass relation while default does not.

**Private**

A Method that is defined private will be accessible only from within the Class in which it is defined. Such Methods are not accessible from outside the defined Class, not even its subclass.

**Java Access Modifiers Table for Method**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Visibility** | **Public** | **Private** | **Protected** | **Default** |
| Within Same Class | Yes | Yes | Yes | Yes |
| From Any Class in Same Package | Yes | No | Yes | Yes |
| From Any Sub Class in Same Package | Yes | No | Yes | Yes |
| From Any Sub Class from Different Package | Yes | No | Yes(**Only By Inheritance**) | No |
| From Any Non Sub Class in Different Package | Yes | No | No | No |

**IV. Access Modifier for Local Variable**

No Access Modifiers can be applied to local variables. Only final can be applied to a local variable which is a Non Access Modifer.

**Difference between Inheritance and Direct Access**

Refer Example: FirstClass.java and SecondClass.java

Super Class: FirstClass, Sub Class: SecondClass

**BULLET POINTS**

* **Public, Private, Protected** are 3 access modifier
* There are 4 access levels in Java. **Public, Private, Protected & Default**.
* A class can have only public and default access level.
* Methods and Instance variable (**non local**) can use all 4 access levels.
* If a class is not visible to other class there is no question of accessing member of that class, even when access level of that member is public(Important).
* Class visibility should be checked before member visibility.
* If a super class has public member then it will be inherited by subclasses even if it is in other package.
* **this** always refers to the current executing object.
* A public member can be accessed by all other class even from other package.
* Private members can be accessed only by the code in the same class.
* Default members are not visible to subclasses outside package while protected members are visible to subclassed even when they are in different package.
* Different between protected and default comes into picture only in the case of subclass outside package.
* Local variables can not have access modifiers.
* Local variables can have only final non access modifiers that can be applied.

**NON-ACCESS MODIFIERS:**

Below are the Non Access Modifiers available in Java.

* Final
* Abstract
* Static
* Strictfp
* Native
* Synchronized
* Transient

**I. Final Non Access Modifier**

Final modifiers are applicable to Class, Method, Instance Variable, Local Variable and Method arguments.

**Final Class:**

A Class when set to final cannot be extended by any other Class.

Example: A String Class in java.lang package

**Final Method:**

A Method when set to final cannot be **overridden** by any subclass.

**Final Variable:**

When a variable is set to final, its value cannot be **changed**. Final variables are like **constants**.

Example: public static final int i = 10;

**II. Abstract Non Access Modifier**

Abstract modifiers are applicable to Class and Method.

**Abstract Class:**

An abstract Class can have abstract Methods. A Class can also be an abstract class without having any abstract Methods in it. If a Class has an abstract Method, the Class becomes an abstract Class.

**Abstract Method:**

Abstract Methods are those Methods which does not have a body but only a signature.

Example: public abstract void method();

**III. Synchronized Non Access Modifier**

Synchronized modifiers are applicable to Method

**Synchronized Method:**

Synchronized Methods can be accessed by only one thread at a time.

**IV. Native Non Access Modifier**

Native modifiers are applicable to Method

**Native Method:**

Naive Methods indicate that the method is implemented on a platform dependent code.

**V. Strictfp Non Access Modifier**

Strictfp modifiers are applicable to Class and Method

**Strictfp Class / Method:**

Strictfp non access modifier forces floating point or floating point operation to adhere to IEEE 754 standard.

**Note\*:** Strictfp non access modifier cannot be applied on a variable.

**BULLET POINTS:**

* **Non Access Modifiers** available in Java are Static, final, abstract, synchronized & Volatile
* Static keyword can be applied to **Variables** & **Methods**.
* **Static Variable** are those variables which are not associated to any instance but it is associated to class means all instances will access the same single copy of variable.
* Local variables can not be declared as Static.
* Static keyword can also be applied to Methods. They will work for all the instances and they will not be dependent on instances created.
* Final modifier can be applied to method and variables.
* Final is the only modifier which will be available to local variable.
* Once declared as final value of the variable can not be changed.
* Final variable don’t get default value opposed to instance variable coz value can’t be changed once assigned.
* Final method can not be overriden.

**OPERATORS IN JAVA**

**Operator Precedence**

Precedence decides which operator will be evaluated first in a case where more than one operator is present in the same calculation.

|  |  |
| --- | --- |
| **OPERATORS** | **PRECEDENCE(HIGH TO LOW)** |
| postfix | expr++ expr-- |
| unary | ++expr --expr +expr -expr ~ ! |
| multiplicative | \* / % |
| additive | + - |
| shift | << >> >>> |
| relational | < > <= >= instanceof |
| equality | == != |
| bitwise AND | & |
| bitwise exclusive OR | ^ |
| bitwise inclusive OR | | |
| logical AND | && |
| logical OR | || |
| ternary | ? : |
| assignment | = += -= \*= /= %= &= ^= |= <<= >>= >>>= |