# **Core Java Notes**

## **What is Java?**

Java is a **programming language** and a **platform**. Java is a high level, robust, object-oriented and secure programming language.

**Types of Java Applications:**

There are mainly 4 types of applications that can be created using Java programming:

**1) Standalone Application**

Standalone applications are also known as desktop applications or window-based applications. AWT and Swing are used in Java for creating standalone applications.

**2) Web Application**

An application that runs on the server side and creates a dynamic page is called a web application.

Currently, [Servlet](https://www.javatpoint.com/servlet-tutorial), [JSP](https://www.javatpoint.com/jsp-tutorial), [Struts](https://www.javatpoint.com/struts-2-tutorial), [Spring](https://www.javatpoint.com/spring-tutorial), [Hibernate](https://www.javatpoint.com/hibernate-tutorial), [JSF](https://www.javatpoint.com/jsf-tutorial), etc. technologies are used for creating web applications in Java.

**3) Enterprise Application**

An application that is distributed in nature, such as banking applications, etc. is called an enterprise application.In Java, [EJB](https://www.javatpoint.com/ejb-tutorial) is used for creating enterprise applications.

**4) Mobile Application**

An application which is created for mobile devices is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

## **Java Platforms / Editions**

There are **4 platforms** or editions of Java:

#### **1) Java SE (Java Standard Edition)**

#### **2) Java EE (Java Enterprise Edition)**

#### **3) Java ME (Java Micro Edition)**

#### **4) JavaFX**

# **Features of Java**

1. [Simple](https://www.javatpoint.com/features-of-java#Simple)
2. [Object-Oriented](https://www.javatpoint.com/features-of-java#Object-Oriented)
3. [Portable](https://www.javatpoint.com/features-of-java#Portable)
4. [Platform independent](https://www.javatpoint.com/features-of-java#Platform-independent)
5. [Secured](https://www.javatpoint.com/features-of-java#Secured)
6. [Interpreted](https://www.javatpoint.com/features-of-java#Interpreted)
7. [High Performance](https://www.javatpoint.com/features-of-java#High-Performance)
8. [Multithreaded](https://www.javatpoint.com/features-of-java#Multithreaded)
9. [Dynamic](https://www.javatpoint.com/features-of-java#Dynamic)

**Q:Java is complied or interpreted language?**

The Java source code first compiled into a binary byte code using Java compiler, then this byte code runs on the JVM (Java Virtual Machine), which is a software based interpreter. So Java is considered as both interpreted and compiled.

Hello.java ----- javac(compiler is fast) ----->Hello.class(understand by jvm)

Hello.class-----java(interpreted (JIT Compiler) ) ---->Runtime output

**Q:Why java is platform independent?**

Platform independent language means once compiled you can execute the program on any platform (OS).

Java is platform independent. Because the Java compiler converts the source code to bytecode, which is Intermidiate Language.

Bytecode can be executed on any platform (OS) using JVM( Java Virtual Machine).

(C : In windows int take 2 bytes and linux it takes 4 bytes)

(Java: all os int consume 4 bytes of memory)

# **Q:What is JVM, Java virtual machine?**

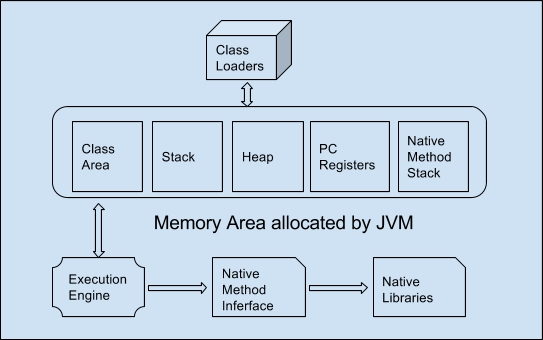
Java uses javac (compiler) to convert the java code to byte code (.class file).

When we run this code using JVM, it internally converts the byte code to system understandable code using an interpreter.

Instead of executing a piece of code, again and again, JVM identifies them as “hot spots” and compiles them using Just in time compiler and, later reuses the same when required.

A compiler compiles (translates) the given program to executable code (whole code at a time). A JIT compiler performs a similar task but it is used by JVM internally, to translate the hotspots in the bytecode.

# **JVM (Java Virtual Machine) Architecture**



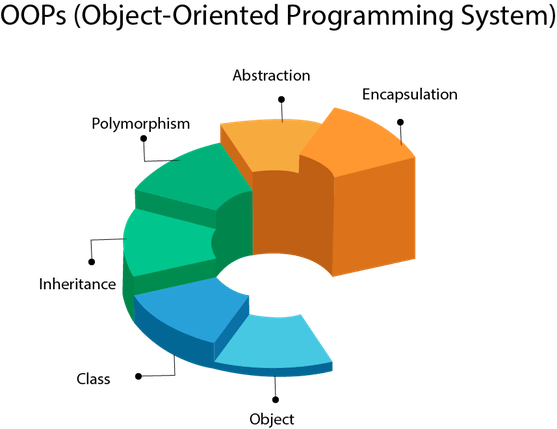
* **Classloader** − Loads the class file into the JVM.
* **Class Area** − Storage areas for a class elements structure like fields, method data, code of method etc.
* **Heap** − Runtime storage allocation for objects.
* **Stack** − Storage for local variables and partial results. A stack contains frames and allocates one for each thread. Once a thread gets completed, this frame also gets destroyed. It also plays roles in method invocation and returns.
* **PC Registers** − Program Counter Registers contains the address of an instruction that JVM is currently executing.
* **Execution Engine** − It has a virtual processor, interpreter to interpret bytecode instructions one by one and a JIT, just in time compiler.
* **Native method stack** − It contains all the native methods used by the application.

# **Q:Differences between JDK, JRE and JVM**

| **JDK** | **JRE** | **JVM** |
| --- | --- | --- |
| JDK (Java Development Kit) is a software development kit to  develop applications in Java | JRE (Java Runtime Environment) is  the implementation of JVM and is  defined as a software package that provides Java class libraries, along  with Java Virtual Machine (JVM). | JVM (Java Virtual Machine) is an abstract machine that  is platform-dependent and has three notions as a specification, a document that describes  requirement of JVM implementation. |
| JDK is primarily used for code execution and has prime functionality of development. | On other hand JRE is majorly responsible for creating environment  for code execution. | JVM on other hand specifies all the implementations and responsible to provide these implementations to JRE. |
| JDK is platform dependent i.e for different platforms different JDK required. | Like of JDK JRE is also platform dependent. | JVM is platform independent. |
| JDK = Java Runtime Environment (JRE) + Development tools | JRE = Java Virtual Machine (JVM) + Libraries to run the application | JVM = Only Runtime environment for executing the Java byte code. |

**Q:What is OOPS?**

OOPS is abbreviated as Object Oriented Programming system in which programs are considered as a collection of objects. Each objecti s nothing but an instance of a class.



1. **Class**

* Class is collection of data member(instance variable,instance properties) & data functions(behaviour and methods in class).
* Class is a blue print or class is a template. Class doesn't have existence.(not runtime)
* e.g.: Car, Cat, Cricketer, Teacher, Product, BankAccount
* Syntax to declare a class:

**class** <class\_name>{

    field;

    method;

}

1. **Object**

* Instance of a class or basic runtime entity or real world entity.
* It has runtime existence.
* Syntax to declare a object:

ClassName **objectName** = **new** ClassName();

**Keywords rule:**

\* true, false, and null might seem like keywords,

\* but they are actually literals; you cannot use them as identifiers in your

Programs.

1. Example of creating class and objects:

**public** **class** ClassExample1 {

//state or field

**public** **int** id;

**private** String name;

//behaviour or methods

**public** **void** dispalyStudentName() {

System.***out***.println("Hello Shilpa");

}

**public** **void** dispalyStudentID() {

System.***out***.println(121);

}

**public** **static** **void** main(String ... args) {

ClassExample1 obj1 = **new** ClassExample1();

obj1.dispalyStudentName();

ClassExample1 obj2 = **new** ClassExample1();

obj2.dispalyStudentID();

}

}

# **Q: Difference between object and class**

|  |  |
| --- | --- |
| **Class** | **Object** |
| A class is a template for creating objects in program. | The object is an instance of a class. |
| A class is a logical entity | Object is a physical entity |
| A class does not allocate memory space when it is created. | Object allocates memory space whenever they are created. |
| You can declare class only once. | You can create more than one object using a class. |
| Example: Car. | Example: Jaguar, BMW, Tesla, etc. |
| Class generates objects | Objects provide life to the class. |
| Classes can’t be manipulated as they are not available in memory. | They can be manipulated. |
| It doesn’t have any values which are associated with the fields. | Each and every object has its own values, which are associated with the fields. |
| You can create class using “class” keyword. | You can create object using “new” keyword in Java |

# **Q:How many Ways to Create an Object in Java ?**

## **Java new Operator:**

Syntax:

class\_name object\_name = **new** class\_name()

## **Java Class.newInstance() method:**

Syntax:

**public** T newInstance() **throws** IllegalAcccessException,

InstantiationException

## **Java newInstance() method of Constructor class:**

Syntax:

**public** T newInstance(Objects...initargs)

## **Java Object.clone() method:**

Syntax:

**protected** Object clone() **throws** CloneNotSupportedException

## **Java Object Serialization and Deserialization:**

Syntax:

**public** **final** **void** writeObject(Object obj) **throws** IOException

**public** **final** Object readObject() **throws** IOException

# **Object class in Java:**

* The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.
* Object class defined in java.lang package is the superclass of all other classes defined in Java programming language.
* Every class extends from the Object class either directly or indirectly
* Each class in java is derived from object class.
* It is also known as cosmic class.
* The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.

### **Methods of Object class:**

* clone() - Creates and returns a copy of this object.
* equals() - Indicates whether some other object is "equal to" this one.
* finalize() - Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
* getClass() - Returns the runtime class of an object.
* hashCode() - Returns a hash code value for the object.
* notify() - Wakes up a single thread that is waiting on this object's monitor.
* notifyAll() - Wakes up all threads that are waiting on this object's monitor.
* toString() - Returns a string representation of the object.
* wait() - Causes current thread to wait until another thread invokes the notify() or notifyAll() method for this object.

**Object Class Ex:**

**class** Student1 {

**int** id;

String name;

**public** Student1(**int** id, String name) {

**this**.name = name;

**this**.id = id;

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **boolean** equals(Object s1) {

System.***out***.println("In equals method of Student class");

**if**(s1==**null**)

**return** **false**;

**if**(!(s1 **instanceof** Student1))

**return** **false**;

Student1 s = (Student1)s1;

**if**(**this**.id==s.getId() && **this**.name==s.getName())

**return** **true**;

**else**

**return** **false**;

}

/\* If we don't implement hash code then even if Objects are equal in nature

\* their hashcode will not be same(not satisfying the contract)

\*/

**public** **int** hashCode() {

**return** **this**.id \* 25 + **this**.name.length();

}

/\* Remember

1) If two objects are equal, then they must have the same hash code.

2) If two objects have the same hash code, they may or may not be equal.

\*/

}

**public** **class** ObjectTest {

**public** **static** **void** main(String[] args) {

Student1 obj1=**new** Student1(1,"Ramesh");

Student1 obj2=**new** Student1(2,"Suresh");

Student1 obj3=**new** Student1(1,"Ramesh");

/\*

-If we don't override equals() method in Student1 class then the default equals method from the

Object class check the reference of the object e.g return (obj1 == obj2);

-Try commenting equals method. If do so then the below equality will return false as both bj1 and

obj2 have different reference (as they are created by new)

\*/

System.***out***.println("obj1.equals(obj3) : "+obj1.equals(obj3));

System.***out***.println("Obj1 hashCode:"+obj1.hashCode());

System.***out***.println("Obj2 hashCode:"+obj2.hashCode());

System.***out***.println("Obj2 hashCode:"+obj3.hashCode()+"\n");

Student1 s1=**new** Student1(5,"Kedar");

Student1 s2=s1;

System.***out***.println("Hash Code of S1:"+s1.hashCode());

System.***out***.println("Hash Code of S1:"+s2.hashCode());

}

}

**equals() and hashcode()**

* equals() and hashcode() are one of the two method of cosmic class java.lang.Object
* Used to compare the objects.

**equals() :**

* Signature : public boolean equals() { }
* The default implementation provided by the JDK is based on memory location — two objects are equal if and only if they are stored or referred in the same memory address (override equals() method to check the equality as per your requirement.)

**hashcode () :**

* Signature : public boolean equals() { }
* This method returns a random integer that is unique for each instance. This integer might change between several executions of the application and won't stay the same.

**Ex-Contract of equals() & hashcode():**

* If two objects are equal according to the equals(Object) method, then calling the hashcode() method on each of the two objects must produce the same integer result.

**public** **class** Student {

**private** **int** id;

**private** String name;

**public** Student(**int** id, String name) {

**this**.name = name;

**this**.id = id;

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

}

**public** **class** HashcodeEqualsTest {

**public** **static** **void** main(String[] args) {

Student s1 = **new** Student(1,"Kunal");

Student s2 = **new** Student(1,"Kunal");

System.***out***.println("S1 hashcode = "+ s1.hashCode());

System.***out***.println("S2hashcode = "+ s2.hashCode());

System.***out***.println("Equality between s1 and s2= "+ s1.equals(s2));

}

}

Output

S1 hashcode = 1993134103

S2 hashcode = 604107971

Equality between s1 and s2= false

***comment*** : If you want to get the equals and hashCode() contract true then we need to implement equals and hascode method.

## **Variable**

1. A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location. It is a combination of "vary + able" which means its value can be changed.
2. Ex: int data=50; // Here data is variable

### **Types of Variables**

There are three types of variables in [Java](https://www.javatpoint.com/java-tutorial):

1. local variable
2. instance variable
3. static variable

#### 1) Local Variable

* A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.
* A local variable cannot be defined with "static" keyword.
* Its scope is local.

#### 2) Instance Variable

* A variable declared inside the class but outside the body of the method, is called an instance variable.
* It is called an instance variable because its value is instance-specific and is not shared among instances.

#### 3) Static variable

* A variable that is declared as static is called a static variable.
* Static is a keyword, its class property.
* it can access using class name.
* Static variable in Java is variable which belongs to the class and initialized only once at the start of the execution.
* it is a variable which belongs to the class and not to object(instance ).

**Ex:**

**public** **class** A

{

**static** **int** m=100;//static variable

**void** method()

    {

**int** n=90;//local variable

    }

**public** **static** **void** main(String args[])

    {

**int** data=50;//instance variable

    }

}//end of class

## **Static Keyword**

* + The static keyword in java is mostly used for memory management.
  + Static keyword can be used for variables,methods,blocks and nested classes.

Static variable:

* A variable that is declared as static is called a static variable.
* Static is a keyword, its class property.
* it can access using class name.
* Static variable in Java is variable which belongs to the class and initialized only once at the start of the execution.
* it is a variable which belongs to the class and not to object(instance ).
* Ex:

**static** int buildYear=2020;

Static Block:

* The static block is a block of statement inside a Java class that will be executed when a class is first loaded into the JVM.
* A static block helps to initialize the static data members.
* Ex:

**static** {

System.out.println("We are in static block");

}

Static Method:

* Static method in Java is a method which belongs to the class and not to the object.
* A static method can access only static data.
* It is a method which belongs to the class and not to the object(instance).
* A static method can access only static data.
* It cannot access non-static data (instance variables).
* A static method can call only other static methods and can not call a non-static method from it.
* A static method can be accessed directly by the class name and doesn’t need any object
* A static method cannot refer to “this” or “super” keywords in anyway.
* Ex:

public **static** void display() {

System.out.println("We are in static methods");

}

**Ex:**

**public class** Carvariable {

**static** {

System.***out***.println("We are in static block"); // static block

}

String model; // object instance property

**static int** *buildYear*=2020; // static variable

**public static void** display() {

System.***out***.println("We are in static methods"); // static methods

}

**public int** avg() {

**int** temp=10; // local variable

fun1();

**return** 0;

}

**private void** fun1() {

// **TODO** Auto-generated method stub

}

**public static void** main(String[] args) {

**int** a,b; // instance variable

Carvariable cv = **new** Carvariable();

cv.model="Audi";

cv.*buildYear*=2020;

System.***out***.println(cv.*buildYear*);

System.***out***.println(Carvariable.*buildYear*)

}

}

# **Constructors**

1. It is a special type of method which is used to initialize the object. It invoke at the time of object creation.
2. Every time an object is created using the new() keyword, at least one constructor is called.
3. It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

### **Rules for creating Java constructor**

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no return type
3. A Java constructor cannot be abstract, static, final, and synchronize

Note: We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

## **Types of Java constructors**

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

# **Default Constructors:**

1. A constructor is called "Default Constructor" when it doesn't have any parameter.
2. Syntax of default constructor:

<class\_name>(){}

**Example of default constructor:**

In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

//Java Program to create and call a default constructor

**public class** defaultConstructor {

//Creating a default constructor

**public** defaultConstructor() {

System.***out***.println("Creating default constructor");

}

**public static void** main (String[] args) {

//Calling default constructor

defaultConstructor dc = **new** defaultConstructor();

}

}

Rule: If there is no constructor in a class, compiler automatically creates a default constructor.

### **Q) What is the purpose of a default constructor?**

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

### **Parameterized Constructor:**

* A constructor which has a specific number of parameters is called a parameterized constructor.
* When parameterize constructor is implemmeted then you should add default constructor also.

### **Why use the parameterized constructor?**

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

### **Example of parameterized constructor**

### In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

**public class** parameterizedConstructor {

**public** String name;

**public** int id;

**public** String address;

//Creating parameterized constructor

parameterizedConstructor(String n,int i, String a){

name= n;

id=i;

address=a;

}

**public** void display() {

System.out.println(name+ " " +id+" "+address);

}

**public static void**(String [] args) {

//Creating object for parameterized constructor and passing values

parameterizedConstructor pc = new parameterizedConstructor("shilpa", 23, "pune");

pc.display();

 }

}

## **Constructor Overloading in Java**

* In Java, a constructor is just like a method but without return type.
* It can also be overloaded like Java methods.
* Constructor [overloading in Java](https://www.javatpoint.com/method-overloading-in-java) is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

**public class** constructorOverloading {

**public** String name;

**public int** id;

**public** String address;

//Creating default constructor

constructorOverloading(){

System.***out***.println(name+ " " +id+" "+address);

}

//Creating two arg constructor

constructorOverloading(String n,**int** i){

name= n;

id=i;

}

//Creating three arg constructor

constructorOverloading(String n,**int** i, String a){

name= n;

id=i;

address=a;

}

**void** display() {

System.***out***.println(name+ " " +id+" "+address);

}

**public static void** main(String [] args) {

constructorOverloading cv1 = **new** constructorOverloading();

constructorOverloading cv2 = **new** constructorOverloading("Kunal",123);

cv2.display();

constructorOverloading cv3 = **new** constructorOverloading("shilpa", 23, "pune");

cv3.display();

}

}

## **Java Copy Constructor**

* There is no copy constructor in Java. However, we can copy the values from one object to another like copy constructor in C++.
* There are many ways to copy the values of one object into another in Java. They are:

1. By constructor
2. By assigning the values of one object into another
3. By clone() method of Object class

* In this example, we are going to copy the values of one object into another using Java constructor.

**public class** copyConstructor{

**int** id;

String name;

  //constructor to initialize integer and string

copyConstructor(**int** i,String n){

id = i;

name = n;

}

//constructor to initialize another object

//We can copy the values from one object to another

copyConstructor(copyConstructor cc){

id = cc.id;

name =cc.name;

}

**void** display(){

System.***out***.println(id+" "+name);

}

**public static void** main(String args[]){

copyConstructor cc1 = **new** copyConstructor(111,"Karan");

copyConstructor cc2 = **new** copyConstructor(cc1);

cc1.display();

cc1.display();

}

}

### **Q: Does constructor return any value?**

Yes, it is the current class instance (You cannot use return type yet it returns a value).

### **Q: Can constructor perform other tasks instead of initialization?**

Yes, like object creation, starting a thread, calling a method, etc. You can perform any operation in the constructor as you perform in the method.

### **Q: Is there Constructor class in Java?**

Yes.

### **Q: What is the purpose of Constructor class?**

Java provides a Constructor class which can be used to get the internal information of a constructor in the class. It is found in the java.lang.reflect package.

**Q: Can we create constructor in abstract class ?**

We can create constructor in abstract class , it does’nt give any compilation error. But when we cannot

instantiate class there is no use in creating a constructor for abstract class.

**Q: Will the compiler creates a default constructor if I have a parameterized constructor in the class?**

No compiler won’t create default constructor if there is parameterized constructor in the class. For

example if I have a class with no constructors, then compiler will create default constructor.

For Example :

public class Car {}

In the above Car class there are no constructors so compiler creates a default constructor.

public class Car {Car(String name) { }

}

In this example compiler won’t create any default constructor because already there is one constructor in the Car class

**Q: How to call one constructor from the other constructor ?**

With in the same class if we want to call one constructor from other we use this() method. Based on the

number of parameters we pass appropriate this() method is called.

Restrictions for using this method :

1) this must be the first statement in the constructor

2)we cannot use two this() methods in the constructor

**Java Methods:**

* A method is a block of code that performs a specific task.
* In Java, there are **two** types of methods:
* User-defined Methods: We can create our own method based on our requirements.
* Standard Library Methods: These are built-in methods in Java that are available to use.

**The syntax to declare a method is:**

returnType methodName() {

// method body

}

**returnType** –

* It specifies what type of value a method returns For example if a method has an int return type then it returns an integer value.
* If the method does not return a value, its return type is void.

**methodName** –

* It is an identifier that is used to refer to the particular method in a program.

**methodbody** –

* It includes the programming statements that are used to perform some tasks. The method body is enclosed inside the curly braces { }.
* For example:

int addNumbers() {

// code

}

This is the simple syntax of declaring a method. However, the complete syntax of declaring a method is

*modifier static returnType nameOfMethod (parameter1, parameter2, ...) {*

*// method body*

*}*

**modifier –**

* It defines access types whether the method is public, private, and so on. To learn more, visit Java Access Specifier.

**static** **–**

* If we use the static keyword, it can be accessed without creating objects.
* For example, the sqrt() method of standard Math class is static. Hence, we can directly call Math.sqrt() without creating an instance of Math class.

**parameter1/parameter2 –**

* These are values passed to a method. We can pass any number of arguments to a method.

**Note: The method is not static. Hence, we are calling the method using the object of the class.**

**EX:**

**class** Method {

// create a method

**public** **int** addNumbers(**int** a, **int** b) {

**int** sum = a + b;

// return value

**return** sum;

}

**public** **static** **void** main(String[] args) {

**int** num1 = 25;

**int** num2 = 15;

// create an object of Main

Method obj = **new** Method();

// calling method

**int** result = obj.addNumbers(num1, num2);

System.***out***.println("Sum is: " + result);

}

}

**Java Method Return Type:**

* A Java method may or may not return a value to the function call. We use the return statement to return any value.
* If the method does not return any value, we use the void keyword as the return type of the method.
* For Ex:

int addNumbers() {

return sum;

}

*we are returning the variable sum. Since the return type of the function is int. The sum variable should be of int type.*

**EX:**

**class** MethodReturnType {

**public** **static** **int** square(**int** num) {

// return statement

**return** num \* num;

}

**public** **static** **void** main(String[] args) {

**int** result;

// store returned value to result

result = *square*(10);

System.***out***.println("Squared value of 10 is: " + result);

}

}

**Standard Library Methods**

* The standard library methods are built-in methods in Java that are readily available for use. These standard libraries come along with
* the Java Class Library (JCL) in a Java archive (\*.jar) file with JVM and JRE.
* For example,
* print() is a method of java.io.PrintSteam. The print("...") method prints the string inside quotation marks.
* sqrt() is a method of Math class. It returns the square root of a number.

**public** **class** StandardLibraryMethod {

**public** **static** **void** main(String[] args) {

// using the sqrt() method

System.***out***.print("Square root of 4 is: " + Math.*sqrt*(4));

}

}

**What are the advantages of using methods?**

1. The main advantage is code reusability. We can write a method once, and use it multiple times. We do not have to rewrite the entire code each time. Think of it as, "write once, reuse multiple times".
2. Methods make code more readable and easier to debug. Here, the getSquare() method keeps the code to compute the square in a block. Hence, makes it more readable.

**Ex: Java Method for Code Reusability**

**public** **class** MethodCodeReusability {

// method defined

**private** **static** **int** getSquare(**int** x){

**return** x \* x;

}

**public** **static** **void** main(String[] args) {

**for** (**int** i = 1; i <= 5; i++) {

// method call

**int** result = *getSquare*(i);

System.***out***.println("Square of " + i + " is: " + result);

}

}

}

# **Access Modifiers in Java:**

* Access specifier is use to decide the access or scope of the variable or function or class

**There are four types of Java access modifiers:**

**Default:**

* declarations are visible only within the package (package private)
* Ex:

package defaultPackage;

class Logger {

void message(){

System.out.println("This is a message");

}

}

* If we do not explicitly specify any access modifier for classes, methods, variables, etc, then by default the default access modifier is considered.

**Private:**

* declarations are visible within the class only
* Ex:

class Data {

private String name;

public String getName() {

return this.name;

}

public void setName(String name) {

this.name= name;

}}

public class Main {

public static void main(String[] main){

Data d = new Data();

// access the private variable using the getter and setter

d.setName("Programiz");

System.out.println(d.getName());

}}

**Note**: We cannot declare classes and interfaces private in Java. However, the nested classes can be declared private. To learn more, visit Java Nested and Inner Class.You cannot set the access modifier of getters methods.

**Protected:**

* declarations are visible within the package or all subclasses
* Ex:

class Animal {

// protected method

protected void display() {

System.out.println("I am an animal");

}

}

class Dog extends Animal {

public static void main(String[] args) {

// create an object of Dog class

Dog dog = new Dog();

// access protected method

dog.display();

}

}

**Note**: We cannot declare classes or interfaces protected in Java.

**Public:**

* declarations are visible everywhere
* Ex:

// Animal.java file

// public class

public class Animal {

// public variable

public int legCount;

// public method

public void display() {

System.out.println("I am an animal.");

System.out.println("I have " + legCount + " legs.");

}

}

// Main.java

public class Main {

public static void main( String[] args ) {

// accessing the public class

Animal animal = new Animal();

// accessing the public variable

animal.legCount = 4;

// accessing the public method

animal.display();

}

}

*The public class Animal is accessed from the Main class.*

*The public variable legCount is accessed from the Main class.*

*The public method display() is accessed from the Main class.*

*Access modifiers are mainly used for encapsulation. It can help us to control what part of a program can access the members of a class. So that misuse of data can be prevented.*

1. **Encapsulation**

* Wrapping/binding of data member & functions together.
* It prevents outer classes from accessing and changing fields and methods of a class. This also helps to achieve data hiding.

**To achieve encapsulation in Java −**

* Declare the variables of a class as private.
* Provide public setter and getter methods to modify and view the variables values.

**EX:**

Following is an example that demonstrates how to achieve Encapsulation in Java

**public** **class** EncapTest {

**private** String name;

**private** String idNum;

**private** **int** age;

**public** **int** getAge() {

**return** age;

}

**public** String getName() {

**return** name;

}

**public** String getIdNum() {

**return** idNum;

}

**public** **void** setAge( **int** newAge) {

age = newAge;

}

**public** **void** setName(String newName) {

name = newName;

}

**public** **void** setIdNum( String newId) {

idNum = newId;

}

}

**public** **class** RunEncap {

**public** **static** **void** main(String args[]) {

EncapTest encap = **new** EncapTest();

encap.setName("James");

encap.setAge(20);

encap.setIdNum("12343ms");

System.***out***.print("Name : " + encap.getName() + " Age : " + encap.getAge());

}

}

## **Benefits of Encapsulation**

* The fields of a class can be made read-only or write-only.
* A class can have total control over what is stored in its fields.

**Note**:

* People often consider encapsulation as data hiding, but that's not entirely true.
* Encapsulation refers to the bundling of related fields and methods together. This can be used to achieve data hiding. Encapsulation in itself is not data hiding.

**Data Hiding**

* Data hiding is a way of restricting the access of our data members by hiding the implementation details. Encapsulation also provides a way for data hiding.
* We can use access modifiers to achieve data hiding. For ex

**public** **class** Datahiding {

**private** **int** age;

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

}

**class** Main {

**public** **static** **void** main(String[] args) {

Datahiding p1 = **new** Datahiding();

p1.setAge(24);

System.***out***.println("My age is " + p1.getAge());

}

1. **Abstraction**

* Hiding the data and expose the necessary functionality. Hide Internal functionality.
* We can achieve abstraction in two ways:
* Using Abstract Class
* Using Interface

## **Abstract Class**

A class which contains the **abstract** keyword in its declaration is known as abstract class.

* Abstract classes may or may not contain *abstract methods*, i.e., methods without body ( public void get(); )
* But, if a class has at least one abstract method, then the class **must** be declared abstract.
* If a class is declared abstract, it cannot be instantiated.
* To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.
* If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

### **Example**

To create an abstract class, just use the **abstract** keyword before the class keyword, in the class declaration.

**public** **abstract** **class** Abstraction {

**private** String name;

**private** String address;

**private** **int** number;

**public** Abstraction(String name, String address, **int** number) {

System.***out***.println("Constructing an Employee");

**this**.name = name;

**this**.address = address;

**this**.number = number;

}

**public** **double** computePay() {

System.***out***.println("Inside Employee computePay");

**return** 0.0;

}

**public** **void** mailCheck() {

System.***out***.println("Mailing a check to " + **this**.name + " " + **this**.address);

}

**public** String toString() {

**return** name + " " + address + " " + number;

}

**public** String getName() {

**return** name;

}

**public** String getAddress() {

**return** address;

}

**public** **void** setAddress(String newAddress) {

address = newAddress;

}

**public** **int** getNumber() {

**return** number;

}

}

**public** **class** AbstractDemo {

**public** **static** **void** main(String [] args) {

/\* Following is not allowed and would raise error \*/

//Abstraction is abstract; cannot be instantiated

Abstraction e = **new** Abstraction("George W.", "Houston, TX", 43);

System.***out***.println("\n Call mailCheck using Employee reference--");

e.mailCheck();

}}

## **Inheriting the Abstract Class**

We can inherit the properties of Employee class just like concrete class in the following way −

**public** **class** SalaryAbstraction **extends** EmployeeAbstraction{

**private** **double** salary; // Annual salary

**public** SalaryAbstraction(String name, String address, **int** number, **double** salary) {

**super**(name, address, number);

setSalary(salary);

}

**public** **void** mailCheck() {

System.***out***.println("Within mailCheck of Salary class ");

System.***out***.println("Mailing check to " + getName() + " with salary " + salary);

}

**public** **double** getSalary() {

**return** salary;

}

**public** **void** setSalary(**double** newSalary) {

**if**(newSalary >= 0.0) {

salary = newSalary;

}

}

**public** **double** computePay() {

System.***out***.println("Computing salary pay for " + getName());

**return** salary/52;

}

}

**public** **class** AbstractDemo {

**public** **static** **void** main(String [] args) {

SalaryAbstraction s = **new** SalaryAbstraction("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);

SalaryAbstraction e = **new** SalaryAbstraction("John Adams", "Boston, MA", 2, 2400.00);

System.***out***.println("Call mailCheck using Salary reference --");

s.mailCheck();

System.***out***.println("\n Call mailCheck using Employee reference--");

e.mailCheck();

}

}

## **Abstract Methods**

* A method that doesn't have its body is known as an abstract method. We use the same abstract keyword to create abstract methods. For example

abstract void display();

* Here, display() is an abstract method. The body of display() is replaced by ;.
* If a class contains an abstract method, then the class should be declared abstract. Otherwise, it will generate an error. For example,

// error

// class should be abstract

class Language {

// abstract method

abstract void method1();

}

**Implementing Abstract Methods**

* If the abstract class includes any abstract method, then all the child classes inherited from the abstract superclass must provide the implementation of the abstract method. For example,

**abstract** **class** Animal {

**abstract** **void** makeSound();

**public** **void** eat() {

System.***out***.println("I can eat."); } }

**class** Dog **extends** Animal {

// provide implementation of abstract method

**public** **void** makeSound() {

System.***out***.println("Bark bark");

}

}

**class** Main {

**public** **static** **void** main(String[] args) {

// create an object of Dog class

Dog d1 = **new** Dog();

d1.makeSound();

d1.eat();

}

}

* In the above example, we have created an abstract class Animal. The class contains an abstract method makeSound() and a non-abstract method eat().
* We have inherited a subclass Dog from the superclass Animal. Here, the subclass Dog provides the implementation for the abstract method makeSound().

**Interfaces:**

* An interface in Java is a mechanism that is used to achieve complete abstraction. It is basically a kind of class that contains only constants and abstract methods.
* Data members declared in an interface are by default public, static, and final.
* Only abstract and public modifiers are allowed for methods in interfaces
* We cannot create an object of interface using new operator. But we can create a reference of interface type and interface reference refers to objects of its implementation classes
* From Java 8 onwards, we can define static and default methods in an interface. Prior to Java 8, it was not allowed.
* A variable in an interface must be initialized at the time of declaration.
* However, an interface is different from a class in several ways, including −
* You cannot instantiate an interface.
* An interface does not contain any constructors.
* All of the methods in an interface are abstract.
* An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
* we can access static methods of an interface using its references.
* An interface is not extended by a class; it is implemented by a class.
* An interface can extend multiple interfaces.

Interface Segregation (I in SOLID principle) -

* larger interfaces should be split into smaller ones.
* By doing so, we can ensure that implementing classes only need to be concerned about the methodsthat are of interest to them.

## **Declaring Interfaces**

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface −

### **Example**

Following is an example of an interface −

/\* File name : NameOfInterface.java \*/

**import** java.lang.\*;

// Any number of import statements

**public** **interface** NameOfInterface {

// Any number of final, static fields

// Any number of abstract method declarations }

Interfaces have the following properties −

* An interface is implicitly abstract. You do not need to use the **abstract** keyword while declaring an interface.
* Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
* Methods in an interface are implicitly public.

**interface** Animal {

**public** **void** eat();

**public** **void** travel();

}

## **Implementing Interfaces**

* When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.
* A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

**public** **class** MammalInt **implements** Animal {

**public** **void** eat() {

System.***out***.println("Mammal eats");

}

**public** **void** travel() {

System.***out***.println("Mammal travels");

}

**public** **int** noOfLegs() {

**return** 0;

}

**public** **static** **void** main(String args[]) {

MammalInt m = **new** MammalInt();

m.eat();

m.travel();

}

}

When overriding methods defined in interfaces, there are several rules to be followed −

* Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
* The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
* An implementation class itself can be abstract and if so, interface methods need not be implemented.

When implementation interfaces, there are several rules −

* A class can implement more than one interface at a time.
* A class can extend only one class, but implement many interfaces.
* An interface can extend another interface, in a similar way as a class can extend another class.

## **Extending Interfaces**

* An interface can extend another interface in the same way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.
* The following Sports interface is extended by Hockey and Football interfaces.

**EX:**

**public** **interface** Sports {

**public** **void** setHomeTeam(String name);

**public** **void** setVisitingTeam(String name);

}

**public** **interface** Football **extends** Sports {

**public** **void** homeTeamScored(**int** points);

**public** **void** visitingTeamScored(**int** points);

**public** **void** endOfQuarter(**int** quarter);

}

**public** **interface** Hockey **extends** Sports {

**public** **void** homeGoalScored();

**public** **void** visitingGoalScored();

**public** **void** endOfPeriod(**int** period);

**public** **void** overtimePeriod(**int** ot);

}

* The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

## **Extending Multiple Interfaces**

* A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.
* The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.
* For example, if the Hockey interface extended both Sports and Event, it would be declared as –

Ex*: public interface Hockey extends Sports, Event*

## **Tagging Interfaces**

* The most common use of extending interfaces occurs when the parent interface does not contain any methods. For example, the MouseListener interface in the java.awt.event package extended java.util.EventListener, which is defined as –

### **Example**

**package** java.util;

**public** **interface** EventListener

{}

An interface with no methods in it is referred to as a **tagging** interface. There are two basic design purposes of tagging interfaces −

* **Creates a common parent** − As with the EventListener interface, which is extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent among a group of interfaces. For example, when an interface extends EventListener, the JVM knows that this particular interface is going to be used in an event delegation scenario.
* **Adds a data type to a class** − This situation is where the term, tagging comes from. A class that implements a tagging interface does not need to define any methods (since the interface does not have any), but the class becomes an interface type through polymorphism.

|  |  |
| --- | --- |
| **Interface** | **Abstract class** |
| Abstract methods | Abstract & concrete methods |
| A Class can implement multiple interfaces | The class can inherit only one Abstract Class |
| The interface does not have access modifiers. Everything defined inside the interface is assumed public modifier. | Abstract Class can have an access modifier. |
| An interface is abstract so that it can’t provide any code. | An abstract class can give complete, default code which should be overridden. |
| You cannot use access modifiers for the method, properties, etc. | You can use an abstract class which contains access modifiers. |
| An interface can inherit multiple interfaces but cannot inherit a class. | An abstract class can inherit a class and multiple interfaces. |
| An interface cannot declare constructors or destructors. | An abstract class can declare constructors and destructors. |
| It can extend any number of interfaces. | It can extend only one class or one abstract class at a time. |
| In an abstract interface keyword, is optional for declaring a method as an abstract. | In an abstract class, the abstract keyword is compulsory for declaring a method as an abstract. |
| An interface can have only public abstract methods. | An abstract class has protected and public abstract methods. |

1. **Polymorphism**
   * + Polymorphism is simply means more than one form.
     + That is, the same entity (method or operator or object) can
     + perform different operations in different scenarios.

**Types of polymorphism:**

**Compile time Polymorphism or Early Binding**

* The polymorphism in which compiler identifies which polymorphic form it has to execute at compile time it self is called as compile time polymorphism or early binding.
* Advantage of early binding is execution will be fast. Because every thing about the method is known to compiler during compilation it self and disadvantage is lack of flexibility.
* Examples of early binding are overloaded methods, overloaded operators and overridden methods that are called directly by using derived objects.

**Runtime Polymorphism or Late Binding**

* The polymorphism in which compiler identifies which polymorphic form to execute at runtime but not at compile time is called as runtime polymorphism or late binding.
* Advantage of late binding is flexibility and disadvantage is execution will be slow as compiler has to get the information about the method to execute at runtime.
* Example of late binding is overridden methods that are called using base class object.

**EX:**

**class** Polygon {

// method to render a shape

**public** **void** render() {

System.***out***.println("Rendering Polygon...");

}

}

**class** Square **extends** Polygon {

// renders Square

**public** **void** render() {

System.***out***.println("Rendering Square...");

}

}

**class** Circle **extends** Polygon {

// renders circle

**public** **void** render() {

System.***out***.println("Rendering Circle...");

} }

**public** **class** polygonMain {

**public** **static** **void** main(String[] args) {

// create an object of Square

Square s1 = **new** Square();

s1.render();

// create an object of Circle

Circle c1 = **new** Circle();

c1.render();

}

}

**Why Polymorphism?**

* Polymorphism allows us to create consistent code. In the previous example, we can also create different methods: renderSquare() and renderCircle() to render Square and Circle, respectively.
* This will work perfectly. However, for every shape, we need to create different methods. It will make our code inconsistent.
* To solve this, polymorphism in Java allows us to create a single method render() that will behave differently for different shapes.Note: The print() method is also an example of polymorphism. It is used to print values of different types like char, int, string, etc.

**We can achieve polymorphism in Java using the following ways:**

1. Method Overriding
2. Method Overloading
3. Operator Overloading

**Method Overriding:**

* During inheritance in Java, if the same method is present in both the superclass and the subclass. Then, the method in the subclass overrides the same method in the superclass. This is called method overriding.
* In this case, the same method will perform one operation in the superclass and another operation in the subclass. For example,

**EX:**

**class** Language {

**public** **void** displayInfo() {

System.***out***.println("Common English Language");

}

}

**class** Java **extends** Language {

@Override

**public** **void** displayInfo() {

System.***out***.println("Java Programming Language");

}

}

**class** Main {

**public** **static** **void** main(String[] args) {

// create an object of Java class

Java j1 = **new** Java();

j1.displayInfo();

// create an object of Language class

Language l1 = **new** Language();

l1.displayInfo();

}

}

* **Note**: The method that is called is determined during the execution of the program. Hence, method overriding is a run-time polymorphism.

**Method Overloading:**

* we can create methods with the same name if they differ in parameters. This is known as method overloading in Java. Here, the same method will perform different operations based on the parameter.
* For ex

void func(int a) { ... }

float func(double a) { ... }

float func(int a, float b) { ... }

**EX:**

**class** Pattern {

// method without parameter

**public** **void** display() {

**for** (**int** i = 0; i < 10; i++) {

System.***out***.print("\*");

}}

// method with single parameter

**public** **void** display(**char** symbol) {

**for** (**int** i = 0; i < 10; i++) {

System.***out***.print(symbol);

}}

}

**class** Main {

**public** **static** **void** main(String[] args) {

Pattern d1 = **new** Pattern();

// call method without any argument

d1.display();

System.***out***.println("\n");

// call method with a single argument

d1.display('#');

}

}

* **Note**: The method that is called is determined by the compiler. Hence, it is also known as compile-time polymorphism.

**Polymorphic Variables**

* A variable is called polymorphic if it refers to different values under different conditions.
* Object variables (instance variables) represent the behavior of polymorphic variables in Java. It is because object variables of a class can refer to objects of its class as well as objects of its subclasses.

**Operator Overloading:**

* The + operator is used to add two entities. However, in Java, the + operator performs two operations.
* When + is used with numbers (integers and floating-point numbers), it performs mathematical addition.
* For example,

int a = 5;

int b = 6;

// + with numbers

int sum = a + b; // Output = 11

* When we use the + operator with strings, it will perform string concatenation (join two strings).
* For example,

String first = "Java ";

String second = "Programming";

// + with strings

name = first + second; // Output = Java Programming

* Here, we can see that the + operator is overloaded in Java to perform two operations: addition and concatenation.
* Note: In languages like C++, we can define operators to work differently for different operands. However, Java doesn't support user-defined operator overloading.

|  |
| --- |
|  |
| When a class have same method name with different argument, than it is called method overloading. | | Method of superclass is **overridden** in subclass **to provide more specific implementation**., than it is called method overriding. |
| Method overloading concept is also known as **compile time polymorphism** in java.  **Method name - same method name.**  **Access modifier -** Does not matter.  **Return type -** Does not matter.  **Number of parameters in java -** Have **different number of** [**parameters**](http://www.javamadesoeasy.com/2015/06/difference-between-arguments-and.html)  **Exception thrown -** Does not matter. | | Method overriding concept is also known as **runtime time polymorphism** in java.  **Method name - same name as of superclass method,**  [**Access modifier**](http://www.javamadesoeasy.com/2015/06/access-modifier-access-specifier-in.html) **-** Must not have more restrictive modifier. Example - public method cannot be overridden by private method.  **Return type -** Java **allow overriding by changing the return type**, but only **Covariant return type** are allowed in java.  **Number of parameters in java -** Have **same number of parameters in java.**  [**Exception thrown**](http://www.javamadesoeasy.com/2015/05/throwdeclare-checked-and-unchecked.html) **-** Overriding method must **not throw new or broader** |
| Method overloading is **generally done in same class** but **can also be done in SubClass** | | Method overriding is always done in **subClass** in java. |
| Both [**Static**](http://www.javamadesoeasy.com/2015/05/static-keyword-in-java-variable-method.html) **and instance method can be overloaded** in java. | | **Only instance methods can be overridden** in java. |
| **Main method can also be overloaded** in java | | **Main method can’t be overridden** in java, because main is static method and static methods can’t be overridden in java |
| **private methods can be overloaded** in java. | | **private methods can’t be overridden** in java, because private methods are not inherited in subClass in java. |
| [**final**](http://www.javamadesoeasy.com/2015/05/final-keyword-in-java-20-salient.html) **methods can be overloaded** in java. | | **final methods can’t be overridden** in java, because final methods are not inherited in subClass in java. |

1. **Inheritance**

* Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance or to create a new class from an existing class.
* The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

## **extends Keyword**

* The extends keyword is used to perform inheritance in Java
* extends is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.
* Syntax**:**

**class** Super {

} **class** Sub **extends** Super {

}

**EX**

**class** Animal {

// field and method of the parent class

String name;

**public** **void** eat() {

System.***out***.println("I can eat");

}

}

// inherit from Animal

**class** Dog **extends** Animal {

// new method in subclass

**public** **void** display() {

System.***out***.println("My name is " + name);

}

}

**class** Main {

**public** **static** **void** main(String[] args) {

// create an object of the subclass

Dog labrador = **new** Dog();

// access field of superclass

labrador.name = "Rohu";

labrador.display();

// call method of superclass

// using object of subclass

labrador.eat();

}

}

## **IS-A (Inheritance) :**

* In Object oriented programming, IS-A relationship denotes “one object is type of another”. IS-A relation denotes Inheritance methodology.
* A simple example of IS-A relation : Dell IS-A laptop.
* Note that Object class will be there always on top in every inheritance Hierarchy, So every class holds IS-A relationship with Object class and instance of  Object test is always true for all class.

## **Has-A (Association) :**

* In Object orientation design, We can say “class one is in Has-A relationship with class B if class A holds reference of Claas B”.
* By this reference of class B, A can access all properties of class B which are allowed.

**Why use inheritance?**

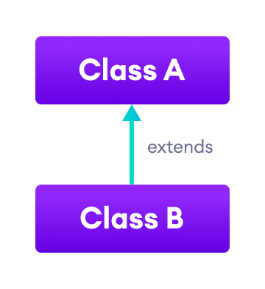
* The most important use of inheritance in Java is code reusability. The code that is present in the parent class can be directly used by the child class.
* Method overriding is also known as runtime polymorphism. Hence, we can achieve Polymorphism in Java with the help of inheritance.

**Types of inheritance**

* There are five types of inheritance.

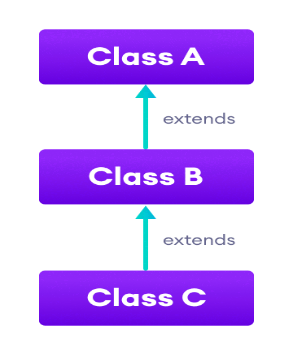
**1. Single Inheritance**

* In single inheritance, a single subclass extends from a single superclass. For example,



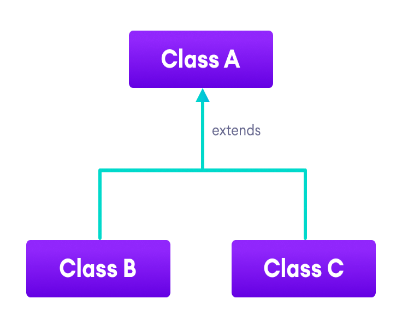
**2. Multilevel Inheritance**

* In multilevel inheritance, a subclass extends from a superclass and then the same subclass acts as a superclass for another class. For example,



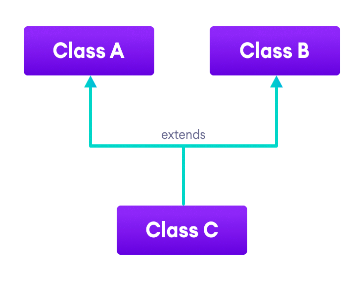
**3. Hierarchical Inheritance**

* In hierarchical inheritance, multiple subclasses extend from a single superclass. For example,



**4. Multiple Inheritance**

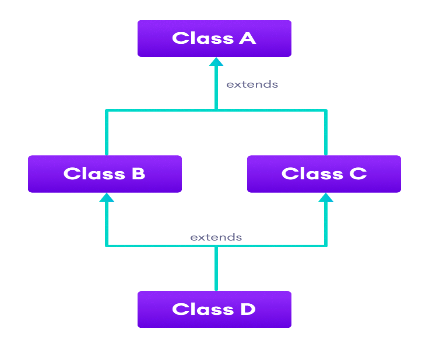
* In multiple inheritance, a single subclass extends from multiple superclasses. For example,



**Note**: Java doesn't support multiple inheritance. However, we can achieve multiple inheritance using interfaces. To learn more, visit Java implements multiple inheritance.

**5. Hybrid Inheritance**

* Hybrid inheritance is a combination of two or more types of inheritance. For example,



**instanceof Operator**

* The instanceof operator in Java is used to check whether an object is an instance of a particular class or not.
* Its syntax is

objectName instanceOf className;

Here, if objectName is an instance of className, the operator returns true. Otherwise, it returns false.

**EX**

**class** Main {

**public** **static** **void** main(String[] args) {

// create a variable of string type

String name = "Programiz";

// checks if name is instance of String

**boolean** result1 = name **instanceof** String;

System.***out***.println("name is an instance of String: " + result1);

// create an object of Main

Main obj = **new** Main();

// checks if obj is an instance of Main

**boolean** result2 = obj **instanceof** Main;

System.***out***.println("obj is an instance of Main: " + result2);

}

}

# **Java – Packages**

* A **java package** is a group of similar types of classes, interfaces and sub-packages.
* The **package keyword** is used to create a package in java.
* Package in java can be categorized in two form, built-in package and user-defined package.
* There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

## **Advantage of Java Package**

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.

## **How to compile java package**

javac -d directory javafilename

## **How to run java package program**

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

|  |
| --- |
| **To Compile:** javac -d . Simple.java |
| **To Run:** java mypack.Simple |
| The -d is a switch that tells the compiler where to put the class file i.e. it represents destination.  The represents the current folder. |
|  |

## **How to access package from another package?**

There are three ways to access the package from outside the package.

1. import package.\*;
2. import package.classname;
3. fully qualified name.

## **Subpackage in java**

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

If you import a package, subpackages will not be imported.

# **Package class**

The package class provides methods to get information about the specification and implementation of a package. It provides methods such as getName(), getImplementationTitle(),getImplementationVendor(),getImplementationVersion() etc.

**Super keyword in java**

* The super keyword in java is a reference variable that is used to refer immediate parent class object.
* Whenever you create the instance of subclass, an instance of parent class is created implicitly i.e. referred by super reference variable.
* Usage of java super Keyword

1. super is used to refer immediate parent class instance variable.

2. super() is used to invoke immediate parent class constructor.

3. super is used to invoke immediate parent class method.

**Problem without super keyword**

**1) super is used to refer immediate parent class instance variable.**

**class** Vehicle{

**int** speed=50;

}

**class** Bike3 **extends** Vehicle{

**int** speed=100;

**void** display(){

System.***out***.println(speed);//will print speed of Bike

}

**public** **static** **void** main(String args[]){

Bike3 b=**new** Bike3();

b.display();

} }

Solution by super keyword

//example of super keyword

**class** Vehicle{

**int** speed=50;

}

**class** Bike4 **extends** Vehicle{

**int** speed=100;

**void** display(){

System.***out***.println(**super**.speed);//will print speed of Vehicle now

}

**public** **static** **void** main(String args[]){

Bike4 b=**new** Bike4();

b.display();

} }

**2) super is used to invoke parent class constructor.**

The super keyword can also be used to invoke the parent class constructor as given below:

**class** Vehicle{

Vehicle(){System.***out***.println("Vehicle is created");}

}

**class** Bike5 **extends** Vehicle{

Bike5(){

**super**();//will invoke parent class constructor

System.***out***.println("Bike is created");

}

**public** **static** **void** main(String args[]){

Bike5 b=**new** Bike5();

} }

Note: super() is added in each class constructor automatically by compiler.

As we know well that default constructor is provided by compiler automatically but it also adds super() for the first statement.If you are creating your own constructor and you don't have either this() or super() as the first statement, compiler will provide super() as the first statement of the constructor.

**3) super can be used to invoke parent class method**

The super keyword can also be used to invoke parent class method. It should be used in case subclass contains the same method as parent class as in the example given below:

**class** Person{

**void** message(){System.***out***.println("welcome");}

}

**class** Student16 **extends** Person{

**void** message(){System.***out***.println("welcome to java");}

**void** display(){

message();//will invoke current class message() method

**super**.message();//will invoke parent class message() method

}

**public** **static** **void** main(String args[]){

Student16 s=**new** Student16();

s.display();

}

}

In case there is no method in subclass as parent, there is no need to use super. In the example given below message() method is invoked from Student class but Student class does not have message() method, so you can directly call message() method.

**Program in case super is not required**

**class** Person{

**void** message(){System.***out***.println("welcome");}

}

**class** Student17 **extends** Person{

**void** display(){

message();//will invoke parent class message() method

}

**public** **static** **void** main(String args[]){

Student17 s=**new** Student17();

s.display();

}

}

**this keyword in java**

* There can be a lot of usage of java this keyword. In java, this is a reference variable that refers to the current object.
* Usage of java this keyword

1. this keyword can be used to refer current class instance variable.

2. this() can be used to invoke current class constructor.

3. this keyword can be used to invoke current class method (implicitly)

4. this can be passed as an argument in the method call.

5. this can be passed as argument in the constructor call.

6. this keyword can also be used to return the current class instance.

Suggestion: If you are beginner to java, lookup only two usage of this keyword.

**1) The this keyword can be used to refer current class instance variable.**

* If there is ambiguity between the instance variable and parameter, this keyword resolves the problem of ambiguity.
* Understanding the problem without this keywordLet's understand the problem if we don't use this keyword by the example given below:

**class** Student10{

**int** id;

String name;

Student10(**int** id,String name){

id = id;

name = name;

}

**void** display(){System.***out***.println(id+" "+name);}

**public** **static** **void** main(String args[]){

Student10 s1 = **new** Student10(111,"Karan");

Student10 s2 = **new** Student10(321,"Aryan");

s1.display();

s2.display();

}

}

In the above example, parameter (formal arguments) and instance variables are same that is why we are using this keyword to distinguish between local variable and instance variable.

Solution of the above problem by this keyword

//example of this keyword

**class** Student11{

**int** id;

String name;

Student11(**int** id,String name){

**this**.id = id;

**this**.name = name;

}

**void** display(){System.***out***.println(id+" "+name);}

**public** **static** **void** main(String args[]){

Student11 s1 = **new** Student11(111,"Karan");

Student11 s2 = **new** Student11(222,"Aryan");

s1.display();

s2.display();

}

}

If local variables(formal arguments) and instance variables are different, there is no need to use this keyword like in the following program:

Program where this keyword is not required

**class** Student12{

**int** id;

String name;

Student12(**int** i,String n){

id = i;

name = n;

}

**void** display(){System.***out***.println(id+" "+name);}

**public** **static** **void** main(String args[]){

Student12 e1 = **new** Student12(111,"karan");

Student12 e2 = **new** Student12(222,"Aryan");

e1.display();

e2.display();

} }

**2) this() can be used to invoked current class constructor.**

The this() constructor call can be used to invoke the current class constructor (constructor chaining). This approach is better if you have many constructors in the class and want to reuse that constructor.

//Program of this() constructor call (constructor chaining)

**class** Student13{

**int** id;

String name;

Student13(){System.***out***.println("default constructor is invoked");}

Student13(**int** id,String name){

**this** ();//it is used to invoked current class constructor.

**this**.id = id;

**this**.name = name;

}

**void** display(){System.***out***.println(id+" "+name);}

**public** **static** **void** main(String args[]){

Student13 e1 = **new** Student13(111,"karan");

Student13 e2 = **new** Student13(222,"Aryan");

e1.display();

e2.display();

}

}

**Where to use this() constructor call?**

* The this() constructor call should be used to reuse the constructor in the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining.
* Let's see the example given below that displays the actual use of this keyword.

**class** Student14{

**int** id;

String name;

String city;

Student14(**int** id,String name){

**this**.id = id;

**this**.name = name;

}

Student14(**int** id,String name,String city){

**this**(id,name);//now no need to initialize id and name

**this**.city=city;

}

**void** display(){System.***out***.println(id+" "+name+" "+city);}

**public** **static** **void** main(String args[]){

Student14 e1 = **new** Student14(111,"karan");

Student14 e2 = **new** Student14(222,"Aryan","delhi");

e1.display();

e2.display(); } }

*Rule: Call to this() must be the first statement in constructor*

**class** Student15{

**int** id;

String name;

Student15(){System.***out***.println("default constructor is invoked");}

Student15(**int** id,String name){

id = id;

name = name;

**this** ();//must be the first statement

}

**void** display(){System.***out***.println(id+" "+name);}

**public** **static** **void** main(String args[]){

Student15 e1 = **new** Student15(111,"karan");

Student15 e2 = **new** Student15(222,"Aryan");

e1.display();

e2.display();

}

}

**Wrapper Classes**

* A Wrapper class is a class whose object wraps or contains primitive data types.

int x=10

Integer obj=new Integer(10)

**Need of Wrapper Classes**

• They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).

public boolean equals(Object obj) { // I changed the state of object.}

• The classes in java.util package handles only objects and hence wrapper classes help in this case also.

Arraylist a=new Arraylist()

a.add(10)

• Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.

•An object is needed to support synchronization in multithreading.

{

//Synchronize

}

**Boxing**

**Autoboxing: Automatic**

* conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing.

**char ch = 'a'; // Literal 'a' store as a primitive data type.**

**Character charObj = 'a';**

// Autoboxing- primitive to Character object conversion

ArrayList<Integer> arrayList = new ArrayList<Integer>();

// Below statement shows Autoboxing because ArrayList stores only object. So 25 int converts to object and then store it into ArrayList

arrayList.add(25); //arrayList.add(Integer.valueOf(25))

**Unboxing**

**Unboxing: Automatically**

* converting an object of a wrapper class to its corresponding primitive type is known as unboxing

Character ch = 'a';

// Below is unboxing - Character object to primitive conversion

char a = ch;

ArrayList<Integer> arrayList = new ArrayList<Integer>();

arrayList.add(24);

int num = arrayList.get(0);

**Primitive to Wrapper Class Conversion**

* We can either use constructor or static factory methods to convert a primitive value to an object of a wrapper class.
* As ofJava 9, however, constructors for many boxed primitives such as Integer or Long have been deprecated.it's highly recommended to only use the factory methods on new code.

int i=10;

Integer anotherObject = Integer.valueOf(i); // don’t use new Integer(10)

* The valueOf() method returns an instance representing the specified int value.
* To convert from a wrapper object to a primitive value, we can use thecorresponding method such as intValue(), doubleValue() etc

int val = object.intValue();

**Serialization**

* Serialization is a process of converting an object into a sequence of bytes which can be persisted to a disk or database or can be sent through streams. The reverse process of creating object from sequence of bytes is called deserialization.
* The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform.
* To make a Java object serializable we implement the java.io.Serializable interface.

**Which API is used in serialization ?**

* The ObjectOutputStream class contains writeObject() method for serializing an Object.

public final void writeObject(Object obj) throws IOException

* The ObjectInputStream class contains readObject() method for deserializing an object.

public final Object readObject() throws IOException, ClassNotFoundException

**Marker Interface**

* is a special interface in Java without any field and method. Marker interface is used to inform compiler that the class implementing it has some special behaviour or meaning. Some example of Marker interface are,

java.io.serializable

java.lang.Cloneable

java.rmi.Remote

java.util.RandomAccess

* All these interfaces does not have any method and field. They only add special behavior to the classes implementing them.
* A marker interface is also called a tagging interface
* Example

class Studentinfo implements Serializable

{

String name;

int rollNo;

static String contact;

public Studentinfo(String n, int r, String c) {

this.name = n;

this.rollNo = r;

this.contact = c;

}

}

public class Demo

{

public static void main(String[] args) {

try {

Studentinfo si = new Studentinfo("Kunal",100 , "87978697866");

FileOutputStream fos = new FileOutputStream("student.txt");

ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.writeObject(si);

oos.flush();

oos.close();

}

catch (Exception e) {

System.out.println(e);

}

}

}

**Deserialization**

class DeSerialize

{

public static void main(String[] args) {

Studentinfo si=null ;

try {

FileInputStream fis = new FileInputStream("student.txt");

ObjectInputStream ois = new ObjectInputStream(fis);

si = (Studentinfo)ois.readObject();

}

catch (Exception e) {

e.printStackTrace(); }

System.out.println(si.name);

System.out. println(si.rollNo);

System.out.println(si.contact);

}

}

**Transient keyword in Java**

* If you want to exclude any object field from getting serialized you can mark that field as transient in Java.
* Note that you cannot use transient keyword with methods or local variables it can only be used with member variables.

Student class -> rollNo, name, marks

Serialization is the mechanism by which any object can be converted to byte stream.By default all of the object instance variables are converted to bytes and persisted/send across network.

There may be a scenario where you don’t want a variable to be part of serialization because that variable may have some sensitive information which you don’t want to persist or send across the network.That variable can be marked as transient in Java.

**Serialization and then Interface**

* Object Serialization in Java is a process used to convert Object into a binary format which can be persisted into a disk or sent over the network to any other running Java virtual machine (send object from one machine to another machine); the reverse process of creating object from the binary stream is called deserialization in Java.

**How to make a Java class Serializable?**

* Making a class Serializable in Java is very easy, Your Java class just needs to implements java.io.Serializable interface and JVM will take care of serializing objects in the default format.

**What is serialVersionUID? What would happen if you don't define this?**

* When we make a class Serializable we have to declare a final static long variable with a value named by serialVersionUID. This variable will be serialized and passed during the byte stream transfer. During deserialization, the compiler will compare this serialVersionUID with the saved one.

**Marker interface**

* If an interface does not contain any methods then it is known as Marker Interface. The Serializable interface is a Marker interface. JVM automatically identifies whether the class is Serializable by checking the Serializable interface is implemented or not.

**How can we restrict some variables to be serialized?**

* If we want to restrict a variable from serialized, we should declare this variable as a transient variable. The transient variable can not be converted to a byte of a stream.

**Do static variables participate in serialization?**

* Static variables or members are belonging to a class not the instance of the class. During serialization, we only consider the current state of an object. So during Serialization, static members are ignored. There is a rule of thumb that static fields are not serialized but there is an exception serialVersionUID.