|  |  |
| --- | --- |
| ***James Gosling*** | Java is a versatile and powerful programming language known for its platform independence, object-oriented design, robustness, and security. Its extensive ecosystem and large community of developers make it a popular choice for building a wide range of applications across different domains.  *Prepared By*  **PARANOIA TECHNOLOGIES** |

**Java**

**What is Java?**

Java is a popular programming language, created in 1995.

It is owned by Oracle, and more than **3 billion** devices run Java.

It is **used for**:

* Mobile applications (especially Android apps)
* Desktop applications
* Web applications
* Smart Card
* Robotics
* Games

**Types of Java Applications**

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

* Standalone Application/Desktop Applications/Window-based Applications
* Web Application
* Enterprise Application
* Mobile Application

**Java Platforms / Editions**

* Java SE (Java Standard Edition)
* Java EE (Java Enterprise Edition)
* Java ME (Java Micro Edition)
* JavaFX

**Why Use Java?**

* Java works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.)
* It is one of the most popular programming language in the world
* It is easy to learn and simple to use
* It is open-source and free
* It is secure, fast and powerful
* Java is an object oriented language which gives a clear structure to programs and allows code to be reused.
* Reduce the development costs.
* As Java is close to languages like C++ and C#

**History of Java**

* The history of Java is very interesting. Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time.
* [**James Gosling**](https://www.javatpoint.com/james-gosling-father-of-java)**, Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.

**Father of JAVA:** [**James Gosling**](https://www.javatpoint.com/james-gosling-father-of-java)



* Firstly, it was called **"Greentalk"** by James Gosling, and the file extension was **.gt.**
* After that, it was called **Oak** and was developed as a part of the Green project.
* **Oak** is a symbol of strength and chosen as a national tree of many countries like the U.S.A., France, Germany, Romania, etc.



* In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.
* **Java** is an island in Indonesia where the first coffee was produced (called Java coffee). It is a kind of espresso bean. Java name was chosen by James Gosling while having a cup of coffee nearby his office.

**Features of Java**

Features of java is also known as **Java buzzwords**

**Simple**

* Java syntax is based on C++ (so easier for programmers to learn it after C++).
* Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
* There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

**Object-oriented**

* Java is an [**object-oriented**](https://www.javatpoint.com/java-oops-concepts) programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporate both data and behavior.

**Platform Independent**

* Java is a write once, run anywhere language.

**Secured**

* Java is best known for its security. With Java, we can develop virus-free systems.
* Java programs runs on JRE(Java Runtime Environment) and no interaction with our system OS

**Robust**

Meaning of Robust is strong. Java is robust because:

* Strong memory management.
* Lack of pointers that avoids security problems.
* Java provides automatic garbage collection which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
* Exception handling and the type checking mechanism in Java.

**Architecture-neutral**

* Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

**Portable**

* Java is portable because it facilitates you to carry the Java byte code to any platform. It doesn't require any implementation.

**High-performance**

* Java is faster than other traditional interpreted programming languages because Java byte code is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++).
* Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

**Distributed**

* Java is distributed because it facilitates users to create distributed applications in Java.
* RMI and `EJB are used for creating distributed applications. Compared to other languages it is easy to create network connection in Java.

**Multi-threaded**

* A thread is like a separate program, executing concurrently.
* Write Java programs that deal with many tasks at once by defining multiple threads.
* The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area.
* Threads are important for multi-media, Web applications, etc.

**Dynamic**

* Java is a dynamic language.
* It supports the dynamic loading of classes.
* It means classes are loaded on demand.

**Source File**

**File name: Hello.java**

public class Hello{

      public static void main(String[] args) {

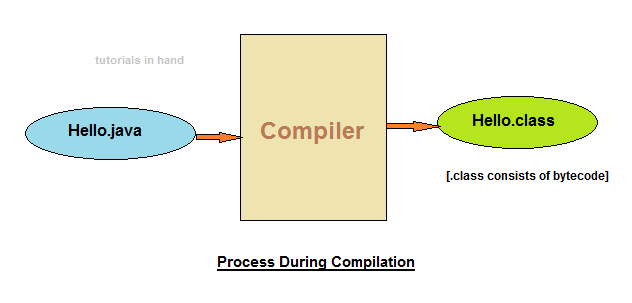
            System.out.print("My first program in java");

      }

}

**Internal working during compile time**

* When you compile the **Hello.java** file then the compiler creates a **.class file**
* **.class file** is created with the name **Hello.class**
* **Hello.class** contains **byte codes**



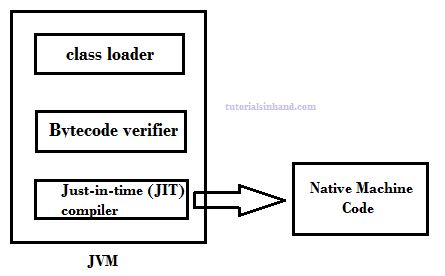
**Byte code (.class file)**is specialized set of instruction that **JVM** can read. Thus this .class file consisting of byte code can be transported to other system, irrespective of the operating system it has, and could be executed comfortably without any issue.

* This is why java a **portable language**.
* Because of the same reason java is also popularly known as **Write once, run anywhere (WORA) language**.

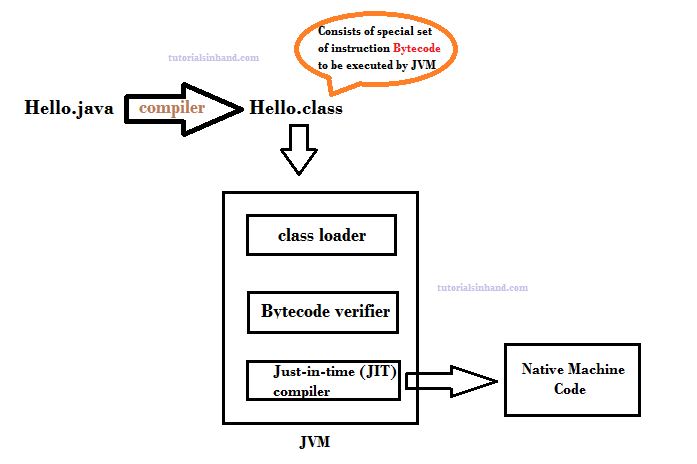
**JVM** is capable of reading, verifying the bytecode.

**Internal working during run time**

During run or execution time following tasks are performed within JVM:

* Class loader loads the .class files containing bytecode to the memory.
* Next the bytecode is verified and checked for any errors or bugs which may result in program exhibiting anomalous behaviour.
* At last, the just-in-time compiler converts the bytecode into machine code.

**Compile + Run diagram of java code**



**Difference between JVM JRE and JDK**

**Java Virtual Machine - JVM**

* JVM is a virtual machine which is capable of reading the ***.class file*** that contains bytecode.
* In java, compiler produces bytecode during compilation which can be run on any system that has JVM installed on it. This results in making java a **portable programming language**. It can be written on any system and run-on different system easily irrespective of operating system. Thus, java is also referred to as **write once, run anywhere**.

 JVM performs three major tasks:

* Loads code
* Verifies code
* Executes code
* Provides runtime environment

**Java Runtime Environment - JRE**

* JRE provides environment to run java applications.
* JRE contains supporting libraries, core classes and other components that JVM uses during the runtime.
* JRE is part of JDK. It can also be downloaded separately to just only run java application and applet.



**Java Development Kit – JDK**

JDK comprises of JRE and other tools that helps in developing, debugging & monitoring the java application.

A JDK always comprises of:

* JRE
* Compiler (javac)
* Debbugger
* Java document



**What is a class?**

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

* **Fields**
* **Methods**
* **Constructors**
* **Blocks**
* **Nested class and interface**

**Syntax to declare a class:**

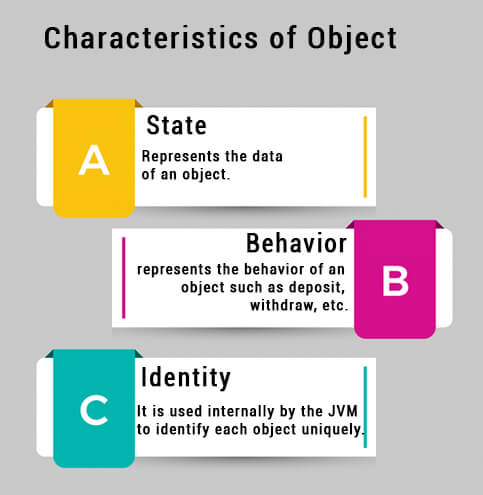
**class** <class name>

{    field;

    method;  }

**What is an object?**

* An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc.
* **An object is an instance of a class.**
* A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.



**How to create an object?**

**Syntax:**

ClassName object = **new** Constructor();

**Using new Keyword**

Using the **new** keyword is the most popular way to create an object or instance of the class. When we create an instance of the class by using the new keyword, it allocates memory (heap) for the newly created **object** and also returns the **reference** of that object to that memory.

**Constructors**

* Constructor is a block of codes similar to the method.
* It is called when an instance of the [class](https://www.javatpoint.com/object-and-class-in-java) is created. At the time of calling constructor, memory for the object is allocated in the memory.
* It is a special type of method which is used to initialize the object.
* Every time an object is created using the new () keyword, at least one constructor is called.
* 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.

* Constructor name must be the same as its class name
* A Constructor must have no explicit return type
* A Java constructor cannot be abstract, static, final, and synchronized

**Types of Java constructors**

There are two types of constructors in Java:

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

**Default Constructor**

A constructor is called "Default Constructor" when it doesn't have any parameter.

**Purpose:**

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

**Syntax:**

class name(){}

**Example**

**class** Fruit

{

//creating a default constructor

Fruit ()

{

System.out.println("Fruit is created");

}

//main method

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

{

//calling a default constructor

Fruit  f = **new**  Fruit ();

Constructor Name

Object Name

Class Name

New Keyword

}

}

**Parameterized Constructor**

A constructor which has a specific number of parameters is called a parameterized constructor.

**Purpose:**

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

**Example**

**class** Fruit

{

//creating a parameterized constructor

Fruit (String str)

{

System.out.println("Fruit is created as" +str);

}

//main method

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

{

//calling a parameterized constructor

Fruit f=**new** Fruit(“Apple”);

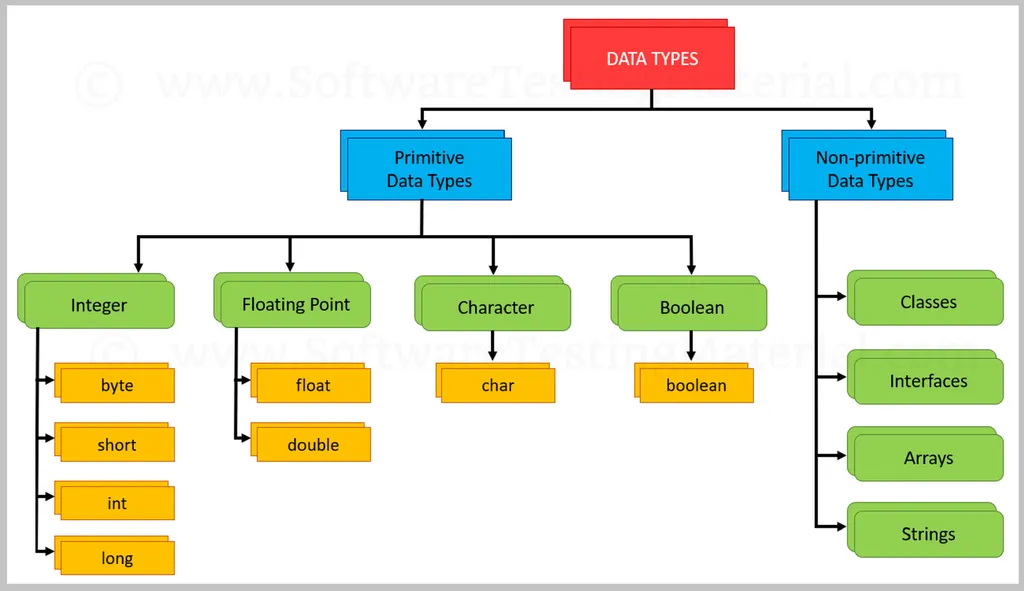
}

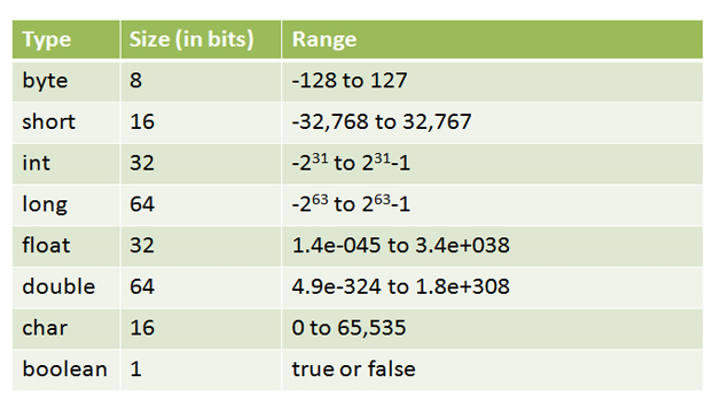
}

**Data Types**

Primarily “**Data types**” in java can be categorized into two types.

* **Primitive data types**
* **Non Primitive or Reference data types.**

****

****

**Variables**

* **Variables** are containers for storing data values.
* **Variables in java**can be thought of as a container that can hold certain values like int, long, byte, etc. during the life time of an application program.

**Syntax**

**type** variablename = value;

**Types of Variables in Java**

* Local Variables
* Instance Variables
* Static Variables

**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.

**Example**

public class A

{

public static void main(String args[])

    {

int a=90;

System.out.println(“value is :”+a);

     }

}//end of class

**2) Instance Variable**

A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as [static](https://www.javatpoint.com/static-keyword-in-java).

It is called an instance variable because its value is instance-specific and is not shared among instances.

**Example**

public class A

{

int d=50; **//instance variable**

public static void main(String args[])

    {

A a=new A();

System.out.println(“Instance Variable:”+a.d);

     }

}//end of class

**3) Static variable**

A variable that is declared as static is called a static variable. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

**Example**

public class A

{

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

   public static void main(String args[])

    {

System.out.println(“Static Variable:”+m);

     }

}//end of class

**Access Modifiers**

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of access modifiers:

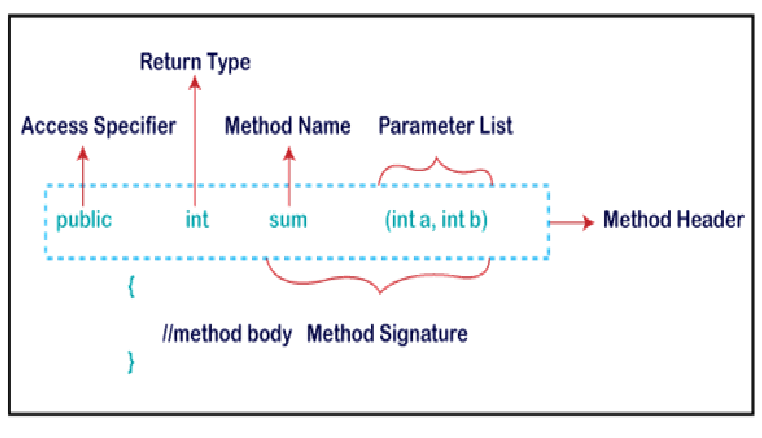
* **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
* **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
* **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
* **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Access Modifier** | **within class** | **within package** | **outside package by subclass only** | **outside package** |
| **Private** | Y | N | N | N |
| **Default** | Y | Y | N | N |
| **Protected** | Y | Y | Y | N |
| **Public** | Y | Y | Y | Y |

**What is a method?**

A **method** is a block of code which only runs when it is called. You can pass data, known as parameters, into a method. Methods are used to perform certain actions, and they are also known as **functions**.



**Types of Method**

There are two types of methods in Java:

* Predefined Method
* User-defined Method

**Predefined Method**

In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the **standard library method** or **built-in method**.

**Example**

**public** **class** Demo

{

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

{

**// using the max() method of Math class which is a predefined method**

System.out.print("The maximum number is: " + Math.max(9,7));

}

}

**User-defined Method**

The method written by the user or programmer is known as **a user-defined** method. These methods are modified according to the requirement.

**Example**

public class A

{

void show()  **//User-defined Method**

    {

        int n=90**;**

System.out.println(“Variable:”+n);

    }

public static void main(String args[])

    {

A a=new A(); **//Object Creation;**

a.show(); **//Method Calling**

     }

}**//end of class**

**Conditional & Control Statements**

A program written in Java programming language is generally executed by [JVM](https://www.scientecheasy.com/2021/03/what-is-jvm.html/) sequentially (one by one) in the order in which they appear.These statements are called sequential statements. The flow of execution takes place from top to bottom.

There are three main types of flow of execution (control) that occur in any computer programming. They are:

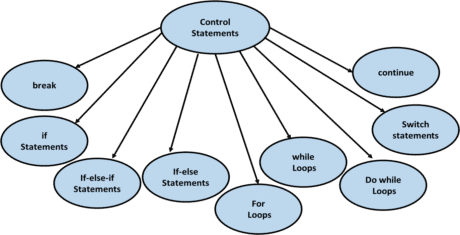
* **Sequential:** Statements execute from top to bottom one by one.
* **Conditional or selection:** Out of two instructions, only one will be executed successfully based on the specified condition. This is because the condition generates the result as either true or false.
* **Repetition or loop:** Group of statements repeats whenever the specified condition is true.

Here are the topics if you want to jump directly:

* [Conditional Statements](https://intellipaat.com/blog/tutorial/java-tutorial/control-statements-in-java/#_conditional_control)
  + [If statement](https://intellipaat.com/blog/tutorial/java-tutorial/control-statements-in-java/%22#_if_statement)
  + [If-else statement](https://intellipaat.com/blog/tutorial/java-tutorial/control-statements-in-java/#_if_else)
  + [If-else-if statement](https://intellipaat.com/blog/tutorial/java-tutorial/control-statements-in-java/#_if_else_if)
  + Nested if statement
  + [Switch statement](https://intellipaat.com/blog/tutorial/java-tutorial/control-statements-in-java/#_switch_statement)
* [Looping Statements](https://intellipaat.com/blog/tutorial/java-tutorial/control-statements-in-java/#_looping_control)
  + [For loop](https://intellipaat.com/blog/tutorial/java-tutorial/control-statements-in-java/#_for_loop)
  + [While loop](https://intellipaat.com/blog/tutorial/java-tutorial/control-statements-in-java/#_while_loop)
  + [Do-while loop](https://intellipaat.com/blog/tutorial/java-tutorial/control-statements-in-java/#_do_while)
* [Unconditional Statements/Jump Statements](https://intellipaat.com/blog/tutorial/java-tutorial/control-statements-in-java/#_unconditional_control)
  + [break statement](https://intellipaat.com/blog/tutorial/java-tutorial/control-statements-in-java/#_break_statement)
  + [continue statement](https://intellipaat.com/blog/tutorial/java-tutorial/control-statements-in-java/#_continue_statement)

**Control Statements**  
Control Statements are used to control the execution flow of the

program.

[](https://intellipaat.com/mediaFiles/2018/12/ja1.png)

There are three types of control statements:

* **Conditional Control Statements**
* **Looping Control Statements**
* **Unconditional Control Statements/Jump Statements**

**Conditional Statements**

Conditional Statements allows the program to select between the alternatives during the program execution.  
They are also called as **decision-making statements or selection statements.**

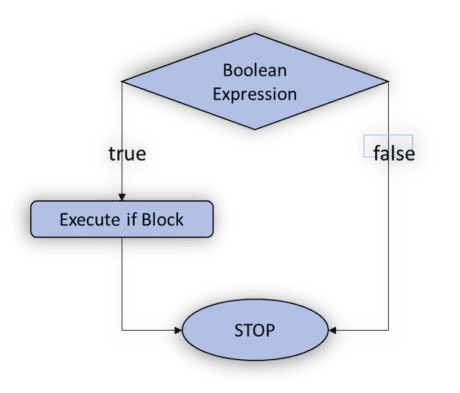
**If statement**

* It will go inside the block only if the condition is true otherwise, it will not execute the block.

**Syntax:**

if (condition)

{  
// statements (if Block)  
}  
//other statements.

**Execution Flow Chart of If Statement**  
[](https://intellipaat.com/mediaFiles/2018/12/ja2.png)

**Example**

class Condif

{ static

{

System.out.println("Simple If Statement");

}

public static void main(String args[])

{

int n=5;//local variable

if(n==5)

{ System.out.println("n=" +n);

}

System.out.println("Done");

}

}

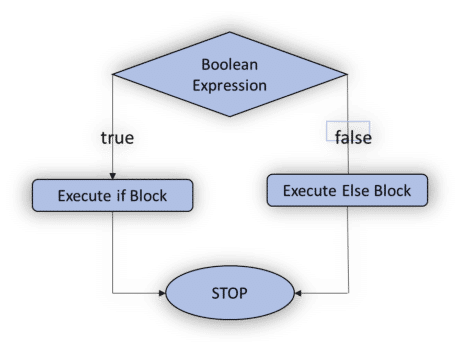
**If-Else Statement**

* If the condition is true then, it will execute the If block. Otherwise, it will execute the Else block.

**Syntax:**

if (condition)

{  
// statements (if Block)  
}  
else{  
// statements (Else block)  
}  
//other statements

**Execution flow chart of If-Else Statement**  
[](https://intellipaat.com/mediaFiles/2018/12/ja3-1.png)

**Example**

class Condifelse

{

static

{

System.out.println("If else Statement");

}

public static void main(String args[])

{

int n=5;

if(n>4)

{

System.out.println("n=" +n);

}

else

{

System.out.println("Wrong number");

}

}

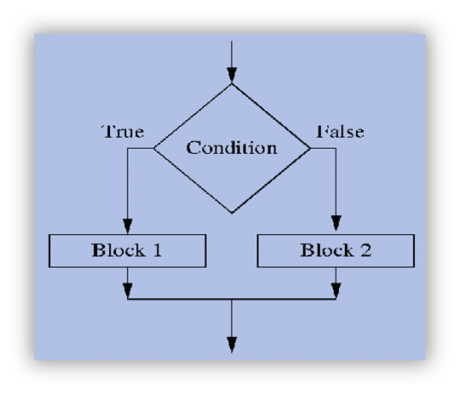
}

**If Else-If statement**

* If the condition is true, then it will execute the If block. Otherwise, it will execute the Else-If block. Again, if the condition is not met, then it will move to the else block.

**Syntax:**

if (condition){  
// statements (if Block)  
}  
else if(condition){  
// statements (Else-If block)  
}  
else{  
//statements(Else Block)  
}//other statements

**Execution flow chart of If Else-If Statement**  
[](https://intellipaat.com/mediaFiles/2018/12/ja4-2.png)

**Example**

class Condifelseif

{

static

{

System.out.println("If else if Statement");

}

public static void main(String args[])

{

int a=10,b=15,c=5;

if((a>b)&&(a>c))

System.out.println("a is greater:" +a);

else if((b>c)&&(b>a))

System.out.println("b is greater:" +b);

else

System.out.println("C is greater:" +c);

}

}

**Nested if statement**

**Example**

class Nestedif

{

static

{

System.out.println("Nested if Statement");

}

public static void main(String args[])

{

int a=70,b=45,c=25,d=50;

if(a>d)

{

System.out.println("a is greater:" +a);

if(b>c) 45>25

{

System.out.println("b is greater:" +b);

}

else

{

System.out.println("C is greater:" +c);

}

}

else

{

System.out.println("D is greater:" +d);

}

}

}

**Switch Statement**

* Switch statement allows program to select one action among multiple actions during the program execution.

**Syntax:**

switch(variable/value/expression)

{  
case:  
//statements;

break;  
case:  
//statements;

break;  
default:  
//statements;  
}

* Based on the argument in the switch statement suitable case value will be selected and executed.
* If no matching case found, then the default will be executed.
* It is optional to write a break statement at the end of each case statement.

**Eample**

import java.util.\*;

class Switchprogram

{

public static void main(String []args)

{

System.out.println("Enter the choice from 0-2");

Scanner sc=new Scanner(System.in);

int x=sc.nextInt();

switch(0) switch(value/variable/expression)

{

case 0:

System.out.println("Hai");

break;

case 1:

System.out.println("Hello");

break;

case 2:

System.out.println("Bye");

break;

default:

System.out.println("No choice found");

}

}

}

**Looping Control Statements**

These are used to execute a block of statements multiple times. It means it executes the same code multiple times so it saves code. These are also called Iteration statements.

There are three types of looping control statements:

* **For loop**
* **While loop**
* **Do-while loop**

**For loop**

* It executes the code until condition is false.
* It is used when numbers of iterations are known.

**Syntax:**

for(initialization; condition; increment/decrement)

{  
//statements(For Body)

}

**Example**

public class Forloop

{

public static void main(String[] args)

{

int i;

for(i=0;i<=5;i++)

{

System.out.println("Hello");

}

}

}

**While loop**

* While loop executes till the condition becomes false.

**Syntax:**

while(condition)

{  
// statements

}

**Example**

public class Whileloop

{

public static void main(String[] args)

{

int i=1;

while(i<5)

{

System.out.println("While Loop");

i++;

}

}

}

**Do-while loop**

* When you are using for or while, then it will execute the loop body only if the condition if true.
* In do-while loop, it will execute the loop first, then it checks the condition. So, it will execute the loop atleast once.
* It is called **exit controlled loop** while **for & while loop** are called **entry controlled loop**.

**Syntax:**

do{  
//statements  
}while(condition);

**Example**

public class DoWhileloop {

public static void main(String[] args) {

int a=5;

do

{ System.out.println("Hello");

a--;

}while(a!=0);

}

}

**Unconditional Control Statements/Jump Statements**

**break Statement**

* break is a keyword. It is used within any control statements. It is used to terminate the execution of the current loop or switch statements.
* **Syntax:** **break;**

**Example**

class Breakprogram

{

public static void main(String args[])

{

int i;

for(i=0;i<5;i++)

{

if(i==3)

break;

System.out.println("Break Statement" +i);

}

}

}

**continue Statement**

* continue is a keyword. It is used to continue the execution of the current loop with the next iteration.

**Syntax:** **continue;**

**Example**

class Continueprogram

{

public static void main(String args[])

{

for(int i=1;i<=4;i++)

{

System.out.println("i="+i);

if(i==3)

continue;

System.out.println("continue1");

}

}

}

**OOPS CONCEPTS**

**Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

* [Object](https://www.javatpoint.com/object-and-class-in-java)
* Class
* [Inheritance](https://www.javatpoint.com/inheritance-in-java)
* [Polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java)
* [Abstraction](https://www.javatpoint.com/abstract-class-in-java)
* [Encapsulation](https://www.javatpoint.com/encapsulation)

**Object**

* Any entity that has state and behavior is known as an object. It can be physical or logical.
* An Object can be defined as an instance of a class.
* An object contains an address and takes up some space in memory.
* **Example:** A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.



**Class**

* Collection of objects is called class. It is a logical entity.
* A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

**Inheritance**

* **Inheritance** is a mechanism in which one class acquires the property of another class.
* With inheritance, we can reuse the fields and methods of the existing class. Hence, inheritance facilitates Reusability and is an important concept of OOPs.
* As we know, a child inherits the properties from his parents. A similar concept is followed in Java, where we have two classes:
* Parent class (Super or Base class)
* Child class (Subclass or Derived class)
* A class which inherits the properties is known as Child Class whereas a class whose properties are inherited is known as Parent class.

**Inheritance Syntax:**

class subClass extends superClass

{

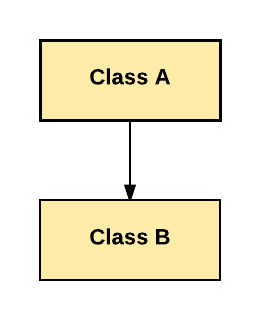
//methods and fields

}

**Types of Inheritance**

### Single Inheritance:

In Single Inheritance one class extends another class (one class only).



In above diagram, Class B extends only Class A. Class A is a super class and Class B is a Sub-class.

**Example**

class Parent{

void Parent\_Details() {

System.out.println("Parent Details...");

}

}

class Child extends Parent {

void Child\_Details() {

System.out.println("Child Detail...");

}

}

public class SingleInheritance {

public static void main(String args[])

{

Child c = new Child();

c.Parent\_Details();

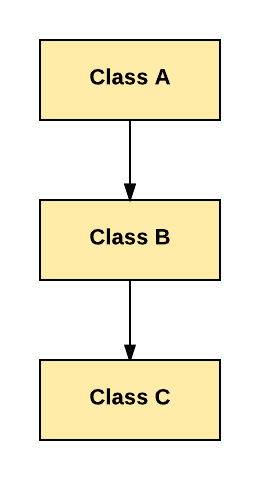
c.Child\_Details();

}

}

### Multilevel Inheritance:

In Multilevel Inheritance, one class can inherit from a derived class. Hence, the derived class becomes the base class for the new class.



As per shown in diagram Class C is subclass of B and B is a of subclass Class A

**Example**

class Parent{

void Parent\_Details() {

System.out.println("Parent Details...");

}

}

class Child1 extends Parent {

void Child1\_Details()

{

System.out.println("Child1 Detail...");

}

}

class Child2 extends Child1 {

void Child2\_Details() {

System.out.println("Child2 Detail...");

}

}

public class MultilevelInheritance

{

public static void main(String args[]) {

Child2 c = new Child2();

c.Parent\_Details();

c.Child1\_Details();

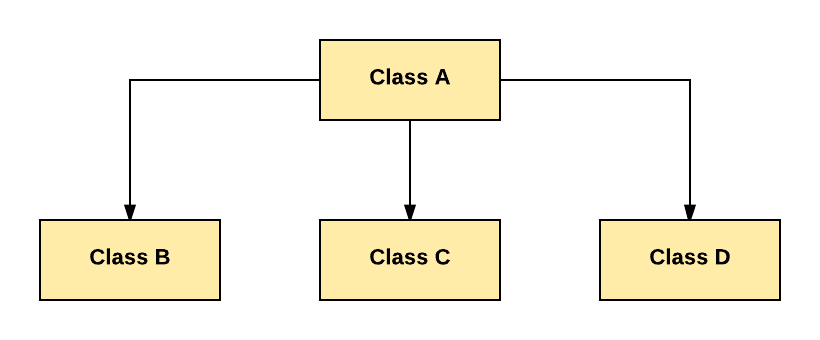
c.Child2\_Details();

}

}

### Hierarchical Inheritance:

In Hierarchical Inheritance, one class is inherited by many sub classes.



As per above example, Class B, C, and D inherit the same class A.

**Example**

class Parent{

void Parent\_Details() {

System.out.println("Parent Details...");

}

}

class Child1 extends Parent {

void Child1\_Details() {

System.out.println("Child1 Detail...");

}

}

class Child2 extends Parent {

void Child2\_Details() {

System.out.println("Child2 Detail...");

}

}

public class HierarichialInheritance

{

public static void main(String args[])

{

Child1 c = new Child1();

Child2 c1 = new Child2();

c.Parent\_Details();

c.Child1\_Details();

c1.Parent\_Details();

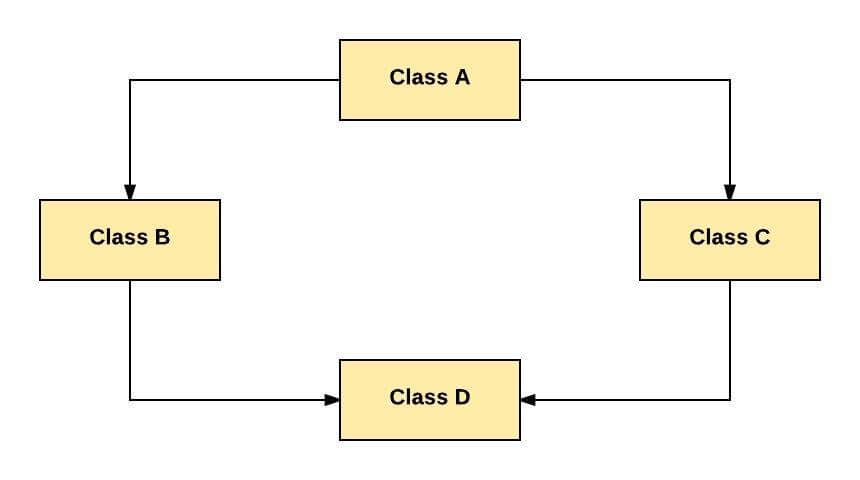
c1.Child2\_Details();

}

}

### Hybrid Inheritance:

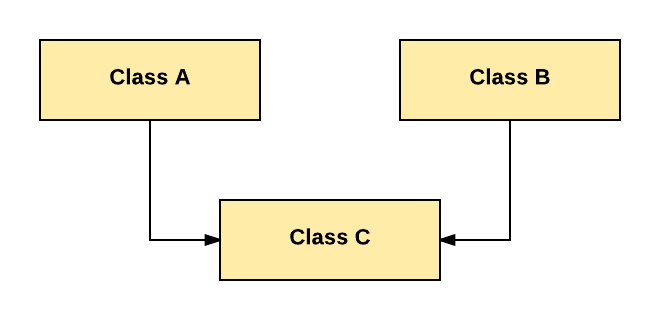
Hybrid inheritance is one of the inheritance types in Java which is a combination of Hierarchical and Multiple inheritance.



As per above example, all the public and protected members of Class A are inherited into Class D, first via Class B and secondly via Class C.

### Multiple Inheritance:

Multiple Inheritance is one of the inheritances in Java types where one class extending more than one class. Java does not support multiple inheritance.



As per above diagram, Class C extends Class A and Class B both.

**Note:** Java doesn’t support hybrid/Multiple inheritance because it results in data ambiguity problem.

### Abstraction

* **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.
* Another way, it shows only essential things to the user and hides the internal details.
* for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

### Ways to achieve Abstraction

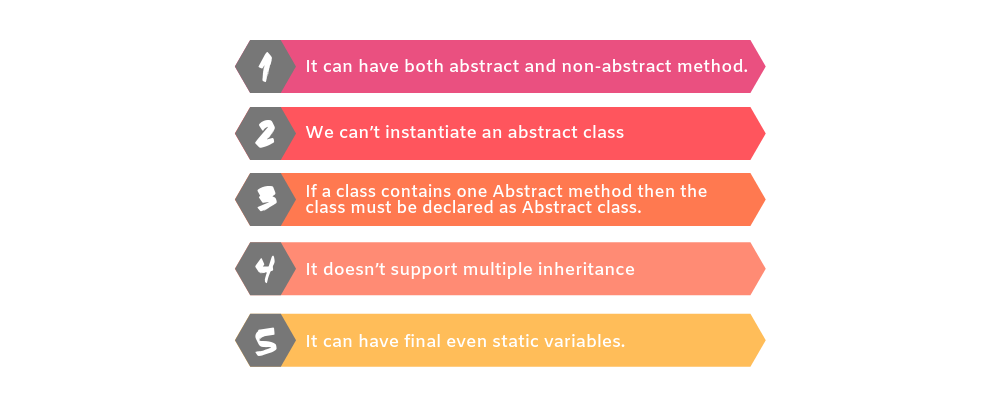
There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

### Abstract class in Java

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

**Rules for abstract class**



**Example**

**abstract** **class** A{}

### Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an abstract method.

**Example of abstract method**

**abstract** **void** printStatus(); **//no method body and abstract**

### Example

**abstract** **class** Birds

{

**abstract** **void** fly();

}

**class** Eagles **extends** Birds

{

**void** fly()

{

System.out.println("Eagles are high flyers");

}

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

{

  Birds b = **new** Eagles();  //upcasting in Java, a subclass object can be referred to by a reference variable of its superclass. This is known as "upcasting."

  b.fly();

}

}

### Abstract class having constructor, data member and methods

### Example

**abstract** **class** Birds

{

Birds() **// Default Constructor**

{

System.out.println(“Birds is created”);

}

**abstract** **void** fly(); **//Abstract Method**

**void** birdquality() **//Non-abstract Method**

**{**

System.out.println(“The eagles are symbol of beauty, bravery, courage, honour, pride, determination, and grace. “);

**}**

}

**class** Eagles **extends** Birds

{

**void** fly()

{

System.out.println("Eagles are high flyers");

}

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

{

  Birds b = **new** Eagles();

  b.fly();

b.birdquality();

}

}

# Interface

* An **interface** is a blueprint of a class. It has static constants and abstract methods.
* The interface in Java is a mechanism to achieve [abstraction](https://www.javatpoint.com/abstract-class-in-java). There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple [inheritance in Java](https://www.javatpoint.com/inheritance-in-java).
* Java Interface also **represents the IS-A relationship.**
* It cannot be instantiated just like the abstract class.
* Since Java 8, we can have **default and static methods** in an interface.
* Since Java 9, we can have **private methods** in an interface.

**Reasons to use interface**

* It is used to achieve abstraction.
* By interface, we can support the functionality of multiple inheritance.
* It can be used to achieve loose coupling.

**How to declare an interface?**

* An interface is declared by using the interface keyword.
* It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default.
* A class that implements an interface must implement all the methods declared in the interface.

### Syntax:

**interface** <interface\_name>{

    // declare constant fields

    // declare methods that abstract

    // by default.

}

**Note:**

* Interface fields are public, static and final by default, and the methods are public and abstract.



#### Relationship between classes and interfaces

**Example**

**interface** Shape

{

**void** draw();

}

**class** Square **implements** Shape{

**public** **void** draw()

{

System.out.println("Draw Square");

}

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

{

Square obj = **new** Square();

obj.draw();

 }

}

**Multiple inheritance in Java by interface**

* If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



**Example**

**interface** Printable

{

**void** print();

}

**interface** Showable

{

**void** show();

}

**class** A7 **implements** Printable,Showable

{

**public** **void** print()

{

System.out.println("Hello");

}

**public** **void** show()

{

System.out.println("Welcome");

}

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

{

A7 obj = **new** A7();

obj.print();

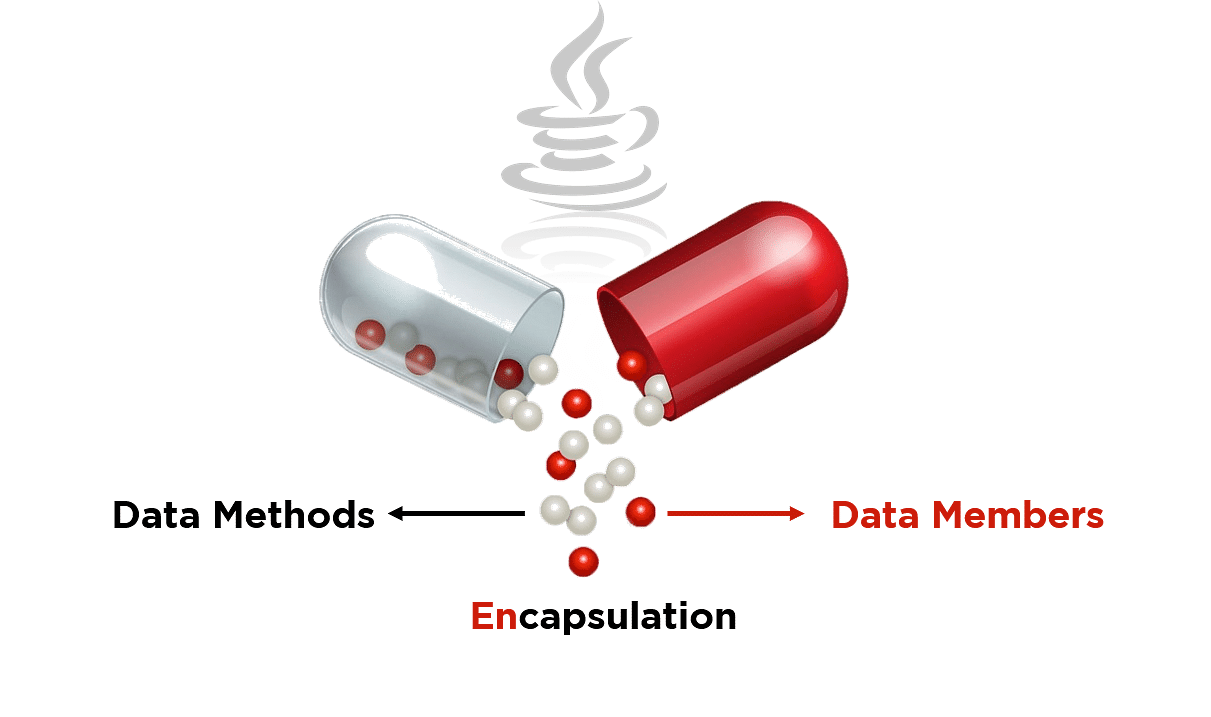
obj.show();

 }

}

**Encapsulation**

* **Encapsulation** is a process of wrapping code and data together into a single unit*,* for example, a capsule which is mixed of several medicines.
* In encapsulation, a class's variables are hidden from other classes and can only be accessed by the methods of the class in which they are found
* We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.
* The **Java Bean** class is the example of a fully encapsulated class.



**Syntax:**

<Access\_Modifier> class <Class\_Name>

{

 private <Data\_Members>;

 private <Data\_Methods>;

}

**Need for Encapsulation**

* Better Control
* Setter and Getter
* Security
* Flexibility

**Example**

**Student.java**

**public** **class** Student

{

**//private data member**

**private** String name;

**//getter method for name**

**public** String getName()

{

**return** name;

}

**//setter method for name**

**public** **void** setName(String name1)

{

Name1=name

}

}

**Test.java**

**class** Test

{

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

{

**//creating instance of the encapsulated class**

Student s=**new** Student();

**//setting value in the name member**

s.setName("vijay");

**//getting value of the name member**

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

}

}

# Polymorphism

**Polymorphism** is a concept by which we can perform a **single action in different ways**. Polymorphism is derived from 2 Greek words: **poly and morphs**. The word "poly" means many and "morphs" means forms. So, polymorphism means many forms.

**Two types of polymorphism**

* Compile-time polymorphism
* Runtime polymorphism.

We can perform polymorphism in java by method overloading and method overriding.

**Runtime Polymorphism (Dynamic Polymorphism)**

* **Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.
* In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

### Upcasting

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting.



**Syntax**

**class** A{}

**class** B **extends** A{}

A a=**new** B ();//upcasting

**Runtime Polymorphism with Method**

**Example**

**class** Fruits

{

**void** flavour()

{

System.out.println("Sweet/Sour/Bitter");}

}

**class** Apple **extends** Fruits{

**void** flavour()

{

System.out.println("Flavour is sweet");

}

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

{

    Fruits f = **new** Apple(); //upcasting

    f.flavour();

  }

}

**Compile time Polymorphism(or Static Polymorphism)**

* Polymorphism that is resolved during compiler time is known as static polymorphism.
* Method overloading is an example of compile time polymorphism.

**Method Overloading**: This allows us to have more than one method having the same name, if the parameters of methods are different in number, sequence and data types of parameters.

**Example**

class SampleAdd

{

int add(int a, int b)

{

return a+b;

}

int add(int a, int b,int c)

{

return a+b+c;

}

}

class Demo

{

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

{

  SampleAdd obj=**new** SampleAdd();

  System.out.println(obj.add(10,20));

System.out.println(obj.add(10,20,30));

}

}

**Keywords**

# this keyword

* this is a **reference variable** that refers to the current object.



**Uses of ‘this’ keyword**

* It can be used to refer instance variable of current class
* It can be used to invoke or initiate current class constructor
* It can be passed as an argument in the method call
* It can be used to return the current class instance

**Example**

class Student

{

int rollno;

String name;

Student(int rollno,String name)

{

this.rollno=rollno;

this.name=name;

}

void display()

{

System.out.println(rollno+" "+name+" ");

}

}

class Test

{

public static void main(String args[])

{

Student s1=new Student(1,”James”);

Student s2=new Student(2,”Alex”);

s1.display();

s2.display();

}

}

# Super Keyword in Java

* The **super** keyword is a reference variable which is used to refer immediate parent class object.
* Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

## Usage of Java super Keyword

* super can be used to refer immediate parent class instance variable.
* super can be used to invoke immediate parent class method.
* super() can be used to invoke immediate parent class constructor.

**Example to invoke super class variables,methods and constructor**

**class** Animal

{

String color="white";

Animal()

{

System.out.println("animal is created");

}

**void** eat()

{

System.out.println("eating...");

}

}

**class** Dog **extends** Animal

{

String color="black";

Dog()

{

**super**();

System.out.println("dog is created");

}

**void** printColor()

{

System.out.println(color);//prints color of Dog class

System.out.println(**super**.color);//prints color of Animal class

}

**void** eat()

{

System.out.println("Dog eating bread...");

**super.**eat();

}

}

**class** TestSuper1

{

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

{

Dog d=**new** Dog();

d.printColor();

d.eat();

}

}

# Final Keyword

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

**Final variable**

If you make any variable as final, you cannot change the value of final variable (It will be constant).

### Example

**class** Car{

**final** **int** speedlimit=100;//final variable

**void** run(){

  speedlimit=400;

 }

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

 Car obj=**new**  Car();

 obj.run();

 }

}

**Final method**

If you make any method as final, you cannot override it.

### Example

**class** Car

{

**final** **void** run()

{

System.out.println("running");

}

}

**class** Hundai **extends** Car

{

**void** run()

{

System.out.println("running safely with 100kmph");

}

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

{

    Hundai h= **new** Hundai();

    h.run();

   }

}

**Final class**

If you make any class as final, you cannot extend it.

### Example

**final** **class** Car{}

**class** Hundai **extends** Car

{

**void** run()

{

System.out.println("running safely with 100kmph");

}

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

{

   Hundai h= **new** Hundai();

   h.run();

  }

}

**Typecasting**

* In Java, type casting, also known as type conversion, is the process of changing the data type of a value or an object from one type to another.
* Type casting is essential when you need to perform operations on values or objects of different data types or when you want to assign a value of one data type to a variable of another data type.
* Two main types of type casting in Java:
  1. **Primitive Type Casting:**
  + This involves converting primitive data types from one type to another.
  + Primitive data types include **int**, **double**, **char**, **boolean**, etc.
  + Casting can be implicit (automatic) or explicit (manual).

**Example of implicit casting (widening):**

int myInt = 5;

double myDouble = myInt; // Implicit casting (widening)

**Example of explicit casting (narrowing):**

double myDouble = 3.14;

int myInt = (int) myDouble; // Explicit casting (narrowing)

**Reference Type Casting:**

* This involves converting references to objects from one class type to another.
* It is used in situations where there's an inheritance hierarchy (subclasses and superclasses) or when working with interfaces.
* Casting can be upcasting (implicit) or downcasting (explicit).

**Example of upcasting (implicit):**

class Vehicle { }

class Car extends Vehicle { }

Car car = new Car();

Vehicle vehicle = car; // Upcasting (implicit)

**Example of downcasting (explicit):**

class Vehicle { }

class Car extends Vehicle { }

Vehicle vehicle = new Car();

Car car = (Car) vehicle; // Downcasting (explicit)