**Java Programming**

**What is java?**

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

**History**

develop by James Gosling in 1995

sun microsystem/oracle

previous name oracle

**It is used for:**

* Mobile applications (specially Android apps)
* Desktop applications
* Web applications
* Web servers and application servers
* Games
* Database connection
* And much, much more!

**Features of java**



**Types of java applications**

Mainly 4 types of applications are created using java

1 standalone application

-desktop or windows based applications created using, AWT, swing

-media player, antivirus

2 web application

-runs o server side to create dynamic web pages

-servlet, jsp, strusts, spring, hibernate, jfs etc

3 enterprise application

-applications that are distributed in nature(banking applications or EA).

-high level security

-load balancing

-clustering

-EJB used for creating enterprise applications

4 mobile applications

-android and java ME use to create mobile applications.

**Java Platform/Editions**

There are 4 platforms or editions of Java:

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

It is a Java programming platform. It includes Java programming APIs such as java.lang, java.io, java.net, java.util, java.sql, java.math etc. It includes core topics like OOPs, [String](https://www.javatpoint.com/java-string), Regex, Exception, Inner classes, Multithreading, I/O Stream, Networking, AWT, Swing, Reflection, Collection, etc.

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

It is an enterprise platform that is mainly used to develop web and enterprise applications. It is built on top of the Java SE platform. It includes topics like Servlet, JSP, Web Services, EJB, [JPA](https://www.javatpoint.com/jpa-tutorial), etc.

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

It is a micro platform that is dedicated to mobile applications.

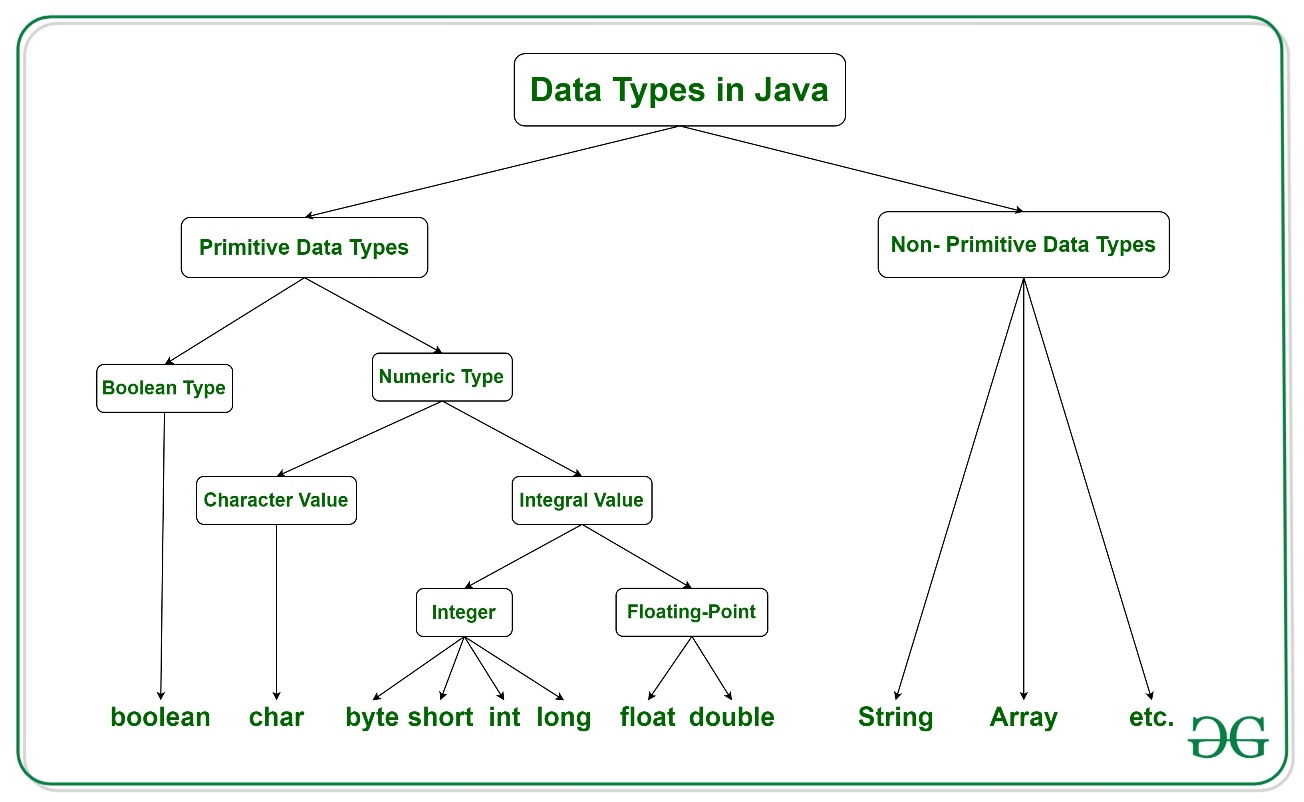
**4) JavaFX**

It is used to develop rich internet applications. It uses a lightweight user interface API.

**Data Types**

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include [Classes](https://www.javatpoint.com/object-and-class-in-java), [Interfaces](https://www.javatpoint.com/interface-in-java), and [Arrays](https://www.javatpoint.com/array-in-java).



**statically** type: here we declare data type of defined variable

**dynamically** type: we don’t need to declare data type of variable.

**Primitive data type**

primitive data types are the building blocks of data manipulation. These are the most basic data types



**Non-primitive data type or reference data type**

The Reference Data Types will contain a memory address of variable values because the reference types won’t store the variable value directly in memory. They are strings, objects, arrays, etc.

**Variables**

-it is a container which holds value while program is executed.

-it is name of memory location inside JVM

* The value stored in a variable can be changed during program execution.
* A variable is only a name given to a memory location. All the operations done on the variable affect that memory location.
* In Java, all variables must be declared before use.

**Declaring & initializing variables**

int a =20;

data\_type= int

variable\_name=a

value\_initialise=20

**Rules for declaring variable name**

1 not begin with a digit

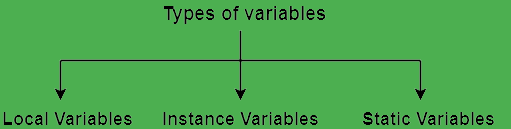
2 case sensitive

3 should not be keyword

4 whitespaces are not allowed

5 no other special character is allow other than ‘$’, \_.

**Types of variables**



**declare inside declare inside class declare with static keyword**

**method outside method inside class outside method**

**1 local variables**

A variable defined within a block or method or constructor is called a local variable.

* These variables are created when the block is entered, or the function is called and destroyed after exiting from the block or when the call returns from the function.
* The scope of these variables exists only within the block in which the variables are declared, i.e., we can access these variables only within that block.
* Initialization of the local variable is mandatory before using it in the defined scope.

**2 instance variables**

Instance variables are non-static variables and are declared in a class outside of any method, constructor, or block.

* As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
* Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier, then the default access specifier will be used.
* Initialization of an instance variable is not mandatory. Its default value is 0.
* Instance variables can be accessed only by creating objects.

**3 static variables**

Static variables are also known as class variables.

* These variables are declared similarly as instance variables. The difference is that static variables are declared using the static keyword within a class outside of any method, constructor or block.
* Unlike instance variables, we can only have one copy of a static variable per class, irrespective of how many objects we create.
* Static variables are created at the start of program execution and destroyed automatically when execution ends.
* Initialization of a static variable is not mandatory. Its default value is 0.
* If we access a static variable like an instance variable (through an object), the compiler will show a warning message, which won’t halt the program. The compiler will replace the object name with the class name automatically.
* If we access a static variable without the class name, the compiler will automatically append the class name.

class A

{

static int a=20;

int b=30;

void func(int c)

{

c=10;

}

}

p s v m(String[] args)

{

int d=40;

sopln(d);

sopln(c);

A ref=new A();

sopln(ref.b);

sopln(A.a);//or sopln(a);

}

}

Note:

-local variable can be easily accessible

-instance variable is accessible by creating object of that class

-static variable can be access with or without class name.

**Scope of variables**

Scope of a variable is the part of the program where the variable is accessible.

-class level scope.

-method level scope.

**Wrapper classes**

-java being object oriented programming it is always expecting object not primitive.

-so wrapper class is mechanism to convert primitive to object and object to primitive.

**Objective of wrapper class**

-to wrap primitive to object form so that we can handle primitive just like object

int Interger

-to wrap int into object some concept is required that concept is known as wrapper class.

-here to wrap int into its object Integer wrapper class in needed.

-to define several utility methods for primitive

eg

convert int to String

String s=Integer.toString(10);

**Need of wrapper class**

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

* **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
* **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
* **Synchronization:** Java synchronization works with objects in Multithreading.
* **java.util package:** The java.util package provides the utility classes to deal with objects.
* **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

**-also wrapper class contains several utility methods.**

1 valueOf()

2 parseInt()

3 toString()

1 valueOf()

The **java string valueOf()** method converts different types of values into string. By the help of string valueOf() method, you can convert int to string, long to string, boolean to string, character to string, float to string, double to string, object to string and char array to string.

**int** value=30;

String s1=String.valueOf(value);

System.out.println(s1+10);//concatenating string with 10

2 parseInt()

The method **parseInt()** belongs to the Integer class which is under **java.lang package**. It is used to parse the string value as a signed decimal value. It is used in Java for converting a string value to an integer by using the method parseInt().

public class ParseIntEx {

public static void main(String[] args) {

int decimalExample = Integer.parseInt("3");

int signedPositiveExample = Integer.parseInt("+3");

int signedNegativeExample = Integer.parseInt("-3");

System.out.println("Integer.parseInt(\"3\") = " + decimalExample);

System.out.println("Integer.parseInt(\"+3\") = " + signedPositiveExample);

System.out.println("Integer.parseInt(\"-3\") = " + signedNegativeExample);

}

}

3 toString()

it is in build method which is use to convert object into string.

The **java.lang.Integer.toString()** is an inbuilt method in Java which is used to return the String object representing this Integer’s value.

1 public static String toString()

2 public static String toString(int *a*)

3 public static String toString(int a, int base)

import java.lang.\*;

public class Geeks{

public static void main(String[] args) {

    Integer obj = new Integer(8);

    //It will return a string representation

    String stringvalue1 = obj.toString();

    System.out.println("String Value= " +

                            stringvalue1);

    String stringvalue2 = obj2.toString(6787);

    System.out.println("String Value = " +

                            stringvalue2);

String returnvalue = a.toString(5254, 2);

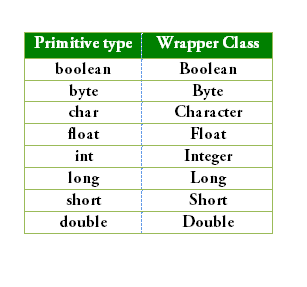
    System.out.println("String Value = " +

                            returnvalue);

}

}

**Primitive data types & their corresponding wrapper classes.**



**Autoboxing & Autounboxing**

**Autoboxing**

Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of int to Integer, long to Long, double to Double etc.

import java.util.ArrayList;

class Autoboxing

{

    public static void main(String[] args)

    {

        char ch = 'a';

        // Autoboxing- primitive to Character object conversion

        Character a = ch;

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

        // Autoboxing because ArrayList stores only objects

        arrayList.add(25);

        // printing the values from object

        System.out.println(arrayList.get(0));

    }

}

**Note**:

before java 1.5 v, autoboxing concept was not there we use valueOf() method to convert primitive to wrapper class object.

public class WrapperExample1{

public static void main(String args[]){

//Converting int into Integer

int a=20;

Integer i=Integer.valueOf(a);//converting int into Integer explicitly

Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally

System.out.println(a+" "+i+" "+j);

}

}

**Autounboxing**

It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For example – conversion of Integer to int, Long to long, Double to double, etc.

import java.util.ArrayList;

class Unboxing

{

    public static void main(String[] args)

    {

        Character ch = 'a';

        // unboxing - Character object to primitive conversion

        char a = ch;

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

        arrayList.add(24);

        // unboxing because get method returns an Integer object

        int num = arrayList.get(0);

        // printing the values from primitive data types

        System.out.println(num);

    }

}

**Note**:

before 1.5 v autounboxing concept was not there then we use xxxValue() method to convert wrapper class into its corresponding primitive datatype.

xxx=int, float etc

// Java program to demonstrate Wrapping and UnWrapping

// in Java Classes

class WrappingUnwrapping

{

public static void main(String args[])

{

// byte data type

byte a = 1;

// wrapping around Byte object

Byte byteobj = new Byte(a);

// int data type

int b = 10;

//wrapping around Integer object

Integer intobj = new Integer(b);

// float data type

float c = 18.6f;

// wrapping around Float object

Float floatobj = new Float(c);

// double data type

double d = 250.5;

// Wrapping around Double object

Double doubleobj = new Double(d);

// char data type

char e='a';

// wrapping around Character object

Character charobj=e;

// printing the values from objects

System.out.println("Values of Wrapper objects (printing as objects)");

System.out.println("Byte object byteobj: " + byteobj);

System.out.println("Integer object intobj: " + intobj);

System.out.println("Float object floatobj: " + floatobj);

System.out.println("Double object doubleobj: " + doubleobj);

System.out.println("Character object charobj: " + charobj);

// objects to data types (retrieving data types from objects)

// unwrapping objects to primitive data types

byte bv = byteobj;

int iv = intobj;

float fv = floatobj;

double dv = doubleobj;

char cv = charobj;

// printing the values from data types

System.out.println("Unwrapped values (printing as data types)");

System.out.println("byte value, bv: " + bv);

System.out.println("int value, iv: " + iv);

System.out.println("float value, fv: " + fv);

System.out.println("double value, dv: " + dv);

System.out.println("char value, cv: " + cv);

} }

**Type casting**

**-type casting** is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer.

In Java, there are two types of casting:

* **Widening Casting** (automatically) - converting a smaller type to a larger type size  
  byte -> short -> char -> int -> long -> float -> double
* **Narrowing Casting** (manually) - converting a larger type to a smaller size type  
  double -> float -> long -> int -> char -> short -> byte

**Widening**

public class Main {

public static void main(String[] args) {

int myInt = 9;

double myDouble = myInt; // Automatic casting: int to double

System.out.println(myInt); // Outputs 9

System.out.println(myDouble); // Outputs 9.0

}

}

**Narrowing**

public class Main {

public static void main(String[] args) {

double myDouble = 9.78d;

int myInt = (int) myDouble; // Manual casting: double to int

System.out.println(myDouble); // Outputs 9.78

System.out.println(myInt); // Outputs 9

}

}

**Operators**

**Operator** in [Java](https://www.javatpoint.com/java-tutorial) is a symbol that is used to perform operations. For example: +, -, \*, / etc.

* **Unary Operator,**
* **Arithmetic Operator,**
* **Shift Operator,**
* **Relational Operator,**
* **Bitwise Operator,**
* **Logical Operator,**
* **Ternary Operator and**
* **Assignment Operator.**
* **instance of operator**

|  |  |  |
| --- | --- | --- |
| **Operator Type** | **Category** | **Precedence** |
| Unary | postfix | *expr*++ *expr*-- |
| prefix | ++*expr* --*expr* +*expr* -*expr* ~ ! |
| Arithmetic | multiplicative | \* / % |
| additive | + - |
| Shift | shift | << >> >>> |
| Relational | comparison | < > <= >= instanceof |
| equality | == != |
| Bitwise | bitwise AND | & |
| bitwise exclusive OR | ^ |
| bitwise inclusive OR | | |
| Logical | logical AND | && |
| logical OR | || |
| Ternary | ternary | ? : |
| Assignment | assignment | = += -= \*= /= %= &= ^= |= <<= >>= >>>= |

**-left shift operator**

The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.

Left shift operator shifts the bits of the number towards **left** a specified number of positions. The symbol for this operator is **<<**. When you write **x<<n**, the meaning is to shift the bits of **x** towards left **n specified** positions.

Example

If **x=10**, then calculate **x<<2** value.

Shifting the value of x towards the left two positions will make the leftmost 2 bits to be lost. The value of x is 10. The binary representation of 10 is **00001010**. The procedure to do left shift explained in the following example:

Observe the above example, after shifting the bits to the left the binary number **00001010** (in decimal 10) becomes **00101000** (in decimal 40).

public class OperatorExample

{

public static void main(String args[])

{

System.out.println(10<<2);//10\*2^2=10\*4=40

System.out.println(10<<3);//10\*2^3=10\*8=80

}

}

**-right shift operator**

The Java right shift operator >> is used to move the value of the left operand to right by the number of bits specified by the right operand.

The Right Shift Operator shifts the bits of the number towards **right** a specified n number of positions. Right shift operator represented by the symbol **>>**, read as double greater than. When you write **x>>n**, the meaning is to shift the bits **x** towards the right n specified positions.

**>>** shifts the bits towards the right and also **preserve the sign bit**, which is the leftmost bit. The leftmost bit represents the **sign** of the number. The sign bit **0** represents a **positive number**, and **1** represents a **negative number**. So after performing **>>** on a positive number, we get a **positive value** in the result also. When we perform **>>** on a **negative number**, again we get a **negative value**.

Example

If **x=10**, then calculate **x>>2** value.

Shifting the value of x towards the right two positions will make the rightmost 2 bits to be lost. The value of x is 10. The binary representation of **10** is **00001010**. The procedure to do right shift explained in the following example:

Observe the above example, after shifting the bits to the right the binary number **00001010** (in decimal 10) becomes **00000010** (in decimal 2).

public OperatorExample

{

public static void main(String args[])

{

System.out.println(20>>2);//20/2^2=20/4=5

System.out.println(20>>3);//20/2^3=20/8=2

}

}

**Bitwise Zero Fill Right Shift Operator (>>>)**

public class OperatorExample

{

public static void main(String args[])

{

//For positive number, >> and >>> works same

System.out.println(20>>2);

System.out.println(20>>>2);

//For negative number, >>> changes parity bit (MSB) to 0

System.out.println(-20>>2);

System.out.println(-20>>>2);

}

}

**-logical (&&) and bitwise (&) AND operator**

The logical && operator doesn't check the second condition if the first condition is false. It checks the second condition only if the first one is true.

The bitwise & operator always checks both conditions whether first condition is true or false.

public class OperatorExample

{

public static void main(String args[])

{

int a=10;

int b=5;

int c=20;

System.out.println(a<b&&a++<c);//false && true = false

System.out.println(a);//10 because second condition is not checked

System.out.println(a<b&a++<c);//false && true = false

System.out.println(a);//11 because second condition is checked

}

}

**-logical (||) and bitwise (|) OR operator**

The logical || operator doesn't check the second condition if the first condition is true. It checks the second condition only if the first one is false.

The bitwise | operator always checks both conditions whether first condition is true or false.

public class OperatorExample

{

public static void main(String args[])

{

int a=10;

int b=5;

int c=20;

System.out.println(a>b||a<c);//true || true = true

System.out.println(a>b|a<c);//true | true = true

//|| vs |

System.out.println(a>b||a++<c);//true || true = true

System.out.println(a);//10 because second condition is not checked

System.out.println(a>b|a++<c);//true | true = true

System.out.println(a);//11 because second condition is checked

}

}

**-ternary operator**

Ternary operator is used as one line replacement for **if-then-else** statement.

It is the only conditional operator which takes three operands.

public class OperatorExample

{

public static void main(String args[])

{

int a=10;

int b=5;

int min=(a<b)?a:b;

System.out.println(min);

}

}

**-if a>b then print a else b**

**Keywords**

Java keywords are also known as reserved words. Keywords are particular words that act as a key to a code. These are predefined words by Java so they cannot be used as a variable or object name or class name.

-**instanceof**

The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).

The instanceof in java is also known as type *comparison operator* because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.

class Animal{}

class Dog1 extends Animal

{//Dog inherits Animal

public static void main(String args[])

{

Dog1 d=new Dog1();

System.out.println(d instanceof Animal);//true

}

}

-**strictfp**

Java strictfp keyword ensures that you will get the same result on every platform if you perform operations in the floating-point variable. The precision may differ from platform to platform that is why java programming language have provided the strictfp keyword, so that you get same result on every platform. So, now you have better control over the floating-point arithmetic.

The strictfp keyword can be applied on methods, classes and interfaces.

strictfp class A{}//strictfp applied on class

strictfp interface M{}//strictfp applied on interface

class A{

strictfp void m(){}//strictfp applied on method

}

**Control statement**

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, [Java](https://www.javatpoint.com/java-tutorial) provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

**Java provides three types of control flow statements.**

1. Decision Making statements
   * if statements
   * switch statement
2. Loop statements
   * do while loop
   * while loop
   * for loop
   * for-each loop
3. Jump statements
   * break statement
   * continue statement

**1 Decision Making Statements**

decision-making statements decide which statement to execute and when.

The [Java](https://www.javatpoint.com/java-tutorial) *if statement* is used to test the condition. It checks [boolean](https://www.javatpoint.com/boolean-keyword-in-java) condition: *true* or *false*. There are various types of if statement in Java.

* if statement
* if-else statement
* if-else-if ladder
* nested if statement

**if and if-else statement**

public class Example

{

public static void main(String[] args)

{

int age=20;

int number=13;

if(age>18){

System.out.print("Age is greater than 18");

}

if(number%2==0){

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

}else{

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

}

}

}

**if-else using ternary operator**

ternary operator (? :) can be used to perform if-else operation

public class IfElseTernaryExample

{

public static void main(String[] args)

{

int number=13;

String output=(number%2==0)?"even number":"odd number";

System.out.println(output);

}

}

**if-else-if ladder**

if-else-if ladder statement executes one condition from multiple statements.

public class IfElseIfExample {

public static void main(String[] args) {

int marks=65;

if(marks<50){

System.out.println("fail");

}

else if(marks>=50 && marks<60){

System.out.println("D grade");

}

else if(marks>=60 && marks<70){

System.out.println("C grade");

}

else if(marks>=70 && marks<80){

System.out.println("B grade");

}

else if(marks>=80 && marks<90){

System.out.println("A grade");

}else if(marks>=90 && marks<100){

System.out.println("A+ grade");

}else{

System.out.println("Invalid!");

}

}

**nested if**

The nested if statement represents the if block within another if block. Here, the inner if block condition executes only when outer if block condition is true.

if(age>=18){

if(weight>50){

System.out.println("You are eligible to donate blood");

}

}

public class JavaNestedIfExample2 {

public static void main(String[] args) {

//Creating two variables for age and weight

int age=25;

int weight=48;

//applying condition on age and weight

if(age>=18){

if(weight>50){

System.out.println("You are eligible to donate blood");

} else{

System.out.println("You are not eligible to donate blood");

}

} else{

System.out.println("Age must be greater than 18");

}

}

}

**switch statement**

The Java *switch statement* executes one statement from multiple conditions. It is like [if-else-if](https://www.javatpoint.com/java-if-else) ladder statement. The switch statement works with byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long. Since Java 7, you can use [strings](https://www.javatpoint.com/java-string) in the switch statement.

In other words, the switch statement tests the equality of a variable against multiple values.

* here can be one or N number of case values for a switch expression.
* The case value must be of switch expression type only. The case value must be literal or constant. It doesn't allow [variables](https://www.javatpoint.com/java-variables).
* The case values must be unique. In case of duplicate value, it renders compile-time error.
* The Java switch expression must be of byte, short, int, long (with its Wrapper type), [*enums*](https://www.javatpoint.com/java-switch) and string.
* Each case statement can have a break statement which is optional. When control reaches to the [break statement](https://www.javatpoint.com/java-break), it jumps the control after the switch expression. If a break statement is not found, it executes the next case.
* The case value can have a default label which is optional.

public class SwitchExample {

public static void main(String[] args) {

//Declaring a variable for switch expression

int number=20;

//Switch expression

switch(number){

//Case statements

case 10: System.out.println("10");

break;

case 20: System.out.println("20");

break;

case 30: System.out.println("30");

break;

//Default case statement

default:System.out.println("Not in 10, 20 or 30");

}

}

}

**Note**:

Java switch statement is fall-through. It means it executes all statements after the first match if a break statement is not present.

public class SwitchExample2 {

public static void main(String[] args) {

int number=20;

//switch expression with int value

switch(number){

//switch cases without break statements

case 10: System.out.println("10");

case 20: System.out.println("20");

case 30: System.out.println("30");

default:System.out.println("Not in 10, 20 or 30");

}

}

}

O/P

20

30

Not in 10, 20 or 30

**Note**:

Java allows us to use enum in switch statement. Java enum is a class that represent the group of constants. (immutable such as final variables). We use the keyword enum and put the constants in curly braces separated by comma.

Java allows us to use four [wrapper classes](https://www.javatpoint.com/wrapper-class-in-java): Byte, Short, Integer and Long in switch statement.

**2 Loop statement**

sometimes we need to execute the block of code repeatedly while some condition evaluates to true

in this condition we use loop

1. for loop
2. while loop
3. do-while loop

**-for loop**

The Java for loop is used to iterate a part of the program several times.

If the number of iteration is **fixed**, it is recommended to use for loop.

-simple for loop

for(initialization; condition; increment/decrement)

{

//statement or code to be executed

}

**-nested for loop**

If we have a for loop inside the another loop, it is known as nested for loop. The inner loop executes completely whenever outer loop executes.

public class NestedForExample

{

public static void main(String[] args)

{

//loop of i

for(int i=1;i<=3;i++)

{

//loop of j

for(int j=1;j<=3;j++)

{

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

}//end of i

}//end of j

}

}

**-for-each loop**

The for-each loop is used to traverse array or collection in Java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation.

It works on the basis of elements and not the index. It returns element one by one in the defined variable.

for(data\_type variable : array\_name)

{

//code to be executed

}

int arr[]={12,23,44,56,78};

//Printing array using for-each loop

for(int i:arr)

{

System.out.println(i);

}

**-labeled for loop**

We can have a name of each Java for loop. To do so, we use label before the for loop. It is useful while using the nested for loop as we can break/continue specific for loop.

labelname:

for(initialization; condition; increment/decrement)

{

//code to be executed

}

public class LabeledForExample {

public static void main(String[] args) {

//Using Label for outer and for loop

aa:

for(int i=1;i<=3;i++){

bb:

for(int j=1;j<=3;j++){

if(i==2&&j==2){

break aa;

}

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

}

}

}

}

**-while loop**

The [Java](https://www.javatpoint.com/java-tutorial) *while loop* is used to iterate a part of the [program](https://www.javatpoint.com/programs-list) repeatedly until the specified Boolean condition is true. As soon as the Boolean condition becomes false, the loop automatically stops.

The while loop is considered as a repeating if statement. If the number of iteration is not fixed, it is recommended to use the while [loop](https://www.javatpoint.com/java-for-loop).

while (condition)

{

//code to be executed

I ncrement / decrement statement

}

public class WhileExample {

public static void main(String[] args) {

int i=1;

while(i<=10){

System.out.println(i);

i++;

}

}

}

**-do-while loop**

The Java *do-while loop* is used to iterate a part of the program repeatedly, until the specified condition is true. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use a do-while loop.

Java do-while loop is called an **exit control loop**. Therefore, unlike while loop and for loop, the do-while check the condition at the end of loop body. The Java *do-while loop* is executed at least once because condition is checked after loop body.

do

{

//code to be executed / loop body

//update statement

}

while (condition);

public class DoWhileExample {

public static void main(String[] args) {

int i=1;

do{

System.out.println(i);

i++;

}while(i<=10);

}

}

**-Difference between 3 loops**

|  |  |  |  |
| --- | --- | --- | --- |
| **Comparison** | **for loop** | **while loop** | **do-while loop** |
| Introduction | The Java for loop is a control flow statement that iterates a part of the [programs](https://www.javatpoint.com/java-programs) multiple times. | The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given boolean condition. | The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends upon the given boolean condition. |
| When to use | If the number of iteration is fixed, it is recommended to use for loop. | If the number of iteration is not fixed, it is recommended to use while loop. | If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use the do-while loop. |
| Syntax | for(init;condition;incr/decr){ // code to be executed } | while(condition){ //code to be executed } | do{ //code to be executed }while(condition); |
| Example | //for loop for(int i=1;i<=10;i++){ System.out.println(i); } | //while loop int i=1; while(i<=10){ System.out.println(i); i++; } | //do-while loop int i=1; do{ System.out.println(i); i++; }while(i<=10); |
| Syntax for infinitive loop | for(;;){ //code to be executed } | while(true){ //code to be executed } | do{ //code to be executed }while(true); |

-**break**

Break Statement is a loop control statement that is used to terminate the loop. As soon as the break statement is encountered from within a loop, the loop iterations stop there, and control returns from the loop immediately to the first statement after the loop.

We can use Java break statement in all types of loops such as [for loop](https://www.javatpoint.com/java-for-loop), [while loop](https://www.javatpoint.com/java-while-loop) and [do-while loop](https://www.javatpoint.com/java-do-while-loop).

break;

for(int i=1;i<=10;i++){

if(i==5){

//breaking the loop

break;

}

System.out.println(i);

}

-**continue**

The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately. It can be used with for loop or while loop.

for(int i=1;i<=10;i++){

if(i==5){

//using continue statement

continue;//it will skip the rest statement

}

System.out.println(i);

}

**Keywords**

| .No | Keyword | Usage |
| --- | --- | --- |
| 1. | **abstract** | Specifies that a class or method will be implemented later, in a subclass |
| 2. | **assert** | Assert describes a predicate placed in a java program to indicate that the developer thinks that the predicate is always true at that place. |
| 3. | **boolean** | A data type that can hold True and False values only |
| 4. | **break** | A control statement for breaking out of loops. |
| 5. | **byte** | A data type that can hold 8-bit data values |
| 6. | **case** | Used in switch statements to mark blocks of text |
| 7. | **catch** | Catches exceptions generated by try statements |
| 8. | **char** | A data type that can hold unsigned 16-bit Unicode characters |
| 9. | **class** | Declares a new class |
| 10. | **continue** | Sends control back outside a loop |
| 11. | **default** | Specifies the default block of code in a switch statement |
| 12. | **do** | Starts a do-while loop |
| 13. | **double** | A data type that can hold 64-bit floating-point numbers |
| 14. | **else** | Indicates alternative branches in an if statement |
| 15. | **enum** | A Java keyword is used to declare an enumerated type. Enumerations extend the base class. |
| 16. | **extends** | Indicates that a class is derived from another class or interface |
| 17. | **final** | Indicates that a variable holds a constant value or that a method will not be overridden |
| 18. | **finally** | Indicates a block of code in a try-catch structure that will always be executed |
| 19. | **float** | A data type that holds a 32-bit floating-point number |
| 20. | **for** | Used to start a for loop |
| 21. | **if** | Tests a true/false expression and branches accordingly |
| 22. | **implements** | Specifies that a class implements an interface |
| 23. | **import** | References other classes |
| 24. | **instanceof** | Indicates whether an object is an instance of a specific class or implements an interface |
| 25. | **int** | A data type that can hold a 32-bit signed integer |
| 26. | **interface** | Declares an interface |
| 27. | **long** | A data type that holds a 64-bit integer |
| 28. | **native** | Specifies that a method is implemented with native (platform-specific) code |
| 29. | **new** | Creates new objects |
| 30. | **null** | This indicates that a reference does not refer to anything |
| 31. | **package** | Declares a Java package |
| 32. | **private** | An access specifier indicating that a method or variable may be accessed only in the class it’s declared in |
| 33. | **protected** | An access specifier indicating that a method or variable may only be accessed in the class it’s declared in (or a subclass of the class it’s declared in or other classes in the same package) |
| 34. | **public** | An access specifier used for classes, interfaces, methods, and variables indicating that an item is accessible throughout the application (or where the class that defines it is accessible) |
| 35. | **return** | Sends control and possibly a return value back from a called method |
| 36. | **short** | A data type that can hold a 16-bit integer |
| 37 | **static** | Indicates that a variable or method is a class method (rather than being limited to one particular object) |
| 38. | **strictfp** | A Java keyword is used to restrict the precision and rounding of floating-point calculations to ensure portability. |
| 39. | **super** | Refers to a class’s base class (used in a method or class constructor) |
| 40. | **switch** | A statement that executes code based on a test value |
| 41. | **synchronized** | Specifies critical sections or methods in multithreaded code |
| 42. | **this** | Refers to the current object in a method or constructor |
| 43. | **throw** | Creates an exception |
| 44. | **throws** | Indicates what exceptions may be thrown by a method |
| 45. | **transient** | Specifies that a variable is not part of an object’s persistent state |
| 46. | **try** | Starts a block of code that will be tested for exceptions |
| 47. | **void** | Specifies that a method does not have a return value |
| 48. | **volatile** | This indicates that a variable may change asynchronously |
| 49. | **while** | Starts a while loop |

**-super keyword**

The**super** keyword in java is a reference variable that is used to refer to parent class objects.

* **Use of super with variables**
* **Use of super with methods**
* **Use of super with constructors**

1

This scenario occurs when a derived class and base class has the same data members. In that case, there is a possibility of ambiguity.

class Vehicle {

int maxSpeed = 120;

}

// sub class Car extending vehicle

class Car extends Vehicle {

int maxSpeed = 180;

void display()

{

// print maxSpeed of base class (vehicle)

System.out.println("Maximum Speed: "

+ super.maxSpeed);

}

}

// Driver Program

class Test {

public static void main(String[] args)

{

Car small = new Car();

small.display();

}

}

O/P 120

2

This is used when we want to call the parent class [method](https://www.geeksforgeeks.org/methods-in-java/). So whenever a parent and child class have the same-named methods then to resolve ambiguity we use the super keyword.

class Person {

void message()

{

System.out.println("This is person class\n");

}

}

// Subclass Student

class Student extends Person {

void message()

{

System.out.println("This is student class");

}

// Note that display() is

// only in Student class

void display()

{

// will invoke or call current

// class message() method

message();

// will invoke or call parent

// class message() method

super.message();

}

}

// Driver Program

class Test {

public static void main(String args[])

{

Student s = new Student();

// calling display() of Student

s.display();

}

}

3

The super [keyword](https://www.geeksforgeeks.org/list-of-all-java-keywords/)can also be used to access the parent class constructor. One more important thing is that ‘super’ can call both parametric as well as non-parametric constructors depending upon the situation.

class Person {

Person()

{

System.out.println("Person class Constructor");

}

}

// subclass Student extending the Person class

class Student extends Person {

Student()

{

// invoke or call parent class constructor

super();

System.out.println("Student class Constructor");

}

}

// Driver Program

class Test {

public static void main(String[] args)

{

Student s = new Student();

}

}

-for constructor super keyword in child class must be first statement

**Note: super keyword is use to call parent from child.**

**-this keyword**

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

-**enum**

java Enums are classes that have a fixed set of constants or variables that do not tend to change.

is achieved using keyword enum.

java enum constants are static and final implicitly.

**difference between class and enum.**

|  |  |
| --- | --- |
| class | enum |
| class constants can be overridden | enum constants cannot be overridden |
| classes support creation of objects | enum cannot support creation of objects |
| classes can extends others classes | enum cannot extends other classes |
| classes can implements interface | enum can implements interface |

**syntax**

enum variable\_name{constants}

eg

enum days{Monday, Sunday}

**define enum**

-we can define enum within or outside class.

enum bike

{

BAJAJ, KTM, YAMAHA

}

public class Edureka

{

p s v m(s[] args)

{

bike c1=bike.YAMAHA;

sopl(c1);

}

}

enum with switch

enum Day

{

SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THRISDAY, FRIDAY, SATURDAY;

}

public class Edureka

{

Day day;

public class Edureka

{

Day day;

public Edureka(Day day)

{

this.day=day;

}

public void dayIsLike()

{

switsh(day)

{

case MONDAY;

sopln(“Hi, Today is Monday”);

break;

case TUESDAY;

sopln(“Hi, Today is Tuesday”);

break;

case WEDNESDAY;

sopln(“Hi Today is Wednesday”);

break;

case TUESDAY;

sopln(“Hi Today is Tuesday)’;

break;

case FRIDAY’

sopln(“Hi Today is Friday);

break;

case SATURDAY;

sopln(“Hi Today is Saturday);

break;

case Default;

sopln(“Enter valid Day”);

break;

}

}

**Inheritance using ENUM.**

public class Edureka{

p s v m(s[] a){

List<HTTPMethodConvertible> inherit=new ArrayList<>();

inherit.add(LibraryEnum.FIRST);

inherit.add(ClientEnum.google);

for(HTTPMethodConvertible element: inherit){

sopln(element.getHTTPMethodType());

}

}

static interface HTTPMethodConvertible {

public String getHTTPMethodType();

}

static enum LibraryEnum implements HTTPMethodConvertible {

FIRST(“Goggle Pixel”), SECOND(“Huawei”), THIRD(“Apple 11 Pro”);

String httpMethodType;

LibraryEnum(String phone) {

httpMethodType=phone;

}

public String getHTTPMethodType(){

return httpMethodType;

}

}

static enum ClientEnum implements HTTPMethodConvertible {

huawei(“Hoonging OS”), apple(“IOS”), google(“Android One”);

String httpMethodType;

ClientEnum(String os){

httpMethodType=os;

}

public String getHTTPMethodType(){

return httpMethodType;

}

}

}

enum TrafficSingle{

RED(“Stop”), GREEN(“Go”), ORANGE(“Wait”);

private String action;

public String getAction() {

return this.action=action;

}

private TrafficSignal(String action) {

this.action=action;

}

}

public class Edureka{

p s v m(s[] a){

TrafficSignal[] signal=TrafficSignal.values();

for(TrafficSignal signal:signal){

sopln(“name:” +signal.name()+ “action:”+ signal.getAction());

}

}

}

**if else with ENUM**

enum Directions{

EAST, WEST, NORTH, SOUTH

}

public class Edureka{

p s v m(String[] args) {

Directions dir=Directions.NORTH;

if(dir==Directions.EAST){

sopln(“Direction EAST”);

}

else if(dir==Direction.WEST){

sopln(“direction west”)

}

--//--

else{

sopln(“direction south”);

}

}

**#Methods of ENUM.**

Java compiler internally adds values(), valueOf() and ordinal() methods within the enum at compile time. It internally creates a static and final class for the enum.

**What is the purpose of the values() method in the enum?**

The Java compiler internally adds the values() method when it creates an enum. The values() method returns an array containing all the values of the enum.

**What is the purpose of the valueOf() method in the enum?**

The Java compiler internally adds the valueOf() method when it creates an enum. The valueOf() method returns the value of given constant enum.

**What is the purpose of the ordinal() method in the enum?**

The Java compiler internally adds the ordinal() method when it creates an enum. The ordinal() method returns the index of the enum value.

enum Color {

    RED,

    GREEN,

    BLUE;

}

public class Test {

    public static void main(String[] args)

    {

        // Calling values()

        Color arr[] = Color.values();

        // enum with loop

        for (Color col : arr) {

            // Calling ordinal() to find index

            // of color.

            System.out.println(col + " at index "

                               + col.ordinal());

        }

        // Using valueOf(). Returns an object of

        // Color with given constant.

        // Uncommenting second line causes exception

        // IllegalArgumentException

        System.out.println(Color.valueOf("RED"));

        // System.out.println(Color.valueOf("WHITE"));

    }

}

**enum and constructor:**

* enum can contain a constructor and it is executed separately for each enum constant at the time of enum class loading.
* We can’t create enum objects explicitly and hence we can’t invoke enum constructor directly.

**enum and methods:**

* enum can contain both **concrete** methods and **abstract** methods. If an enum class has an abstract method, then each instance of the enum class must implement it

enum Color {

    RED,

    GREEN,

    BLUE;

    // enum constructor called separately for each

    // constant

    private Color()

    {

        System.out.println("Constructor called for : "

                           + this.toString());

    }

    public void colorInfo()

    {

        System.out.println("Universal Color");

    }

}

public class Test {

    // Driver method

    public static void main(String[] args)

    {

        Color c1 = Color.RED;

        System.out.println(c1);

        c1.colorInfo();

    }

}

**#Advantages of enum.**

-improve type safety.

-easily usable in switch cases.

-can be traversed.

-has fields, methods, constructors.

-can implements interfaces.

**Object Oriented Programming**

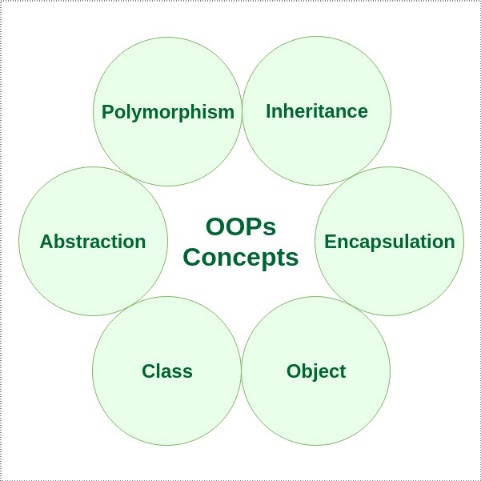
[Object-Oriented Programming](https://www.geeksforgeeks.org/object-oriented-programming-oops-concept-in-java/) or OOPs refers to languages that use objects in programming, they use objects as a primary source to implement what is to happen in the code. Objects are seen by the viewer or user, performing tasks assigned by you. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism etc. in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

**OOPs contains**

1. **class**
2. **object**
3. **method and method passing**
4. **4 pillars of OOPs**
5. **abstraction**
6. **encapsulation**
7. **inheritance**
8. **polymorphism**

**compile time polymorphism**

**run time polymorphism**



**Naming conventions in java**

Java follows **camel-case syntax** for naming the class, interface, method, and variable.

If the name is combined with two words, the second word will start with uppercase letter always such as actionPerformed(), firstName, ActionEvent, ActionListener, etc.

|  |  |  |
| --- | --- | --- |
| **Identifiers Type** | **Naming Rules** | **Examples** |
| Class | It should start with the uppercase letter. It should be a noun such as Color, Button, System, Thread, etc. Use appropriate words, instead of acronyms. | public class **Employee** { //code snippet } |
| Interface | It should start with the uppercase letter. It should be an adjective such as Runnable, Remote, ActionListener. Use appropriate words, instead of acronyms. | interface **Printable** { //code snippet } |
| Method | It should start with lowercase letter. It should be a verb such as main(), print(), println(). If the name contains multiple words, start it with a lowercase letter followed by an uppercase letter such as actionPerformed(). | class Employee { // method void **draw()** { //code snippet } } |
| Variable | It should start with a lowercase letter such as id, name. It should not start with the special characters like & (ampersand), $ (dollar), \_ (underscore). If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as firstName, lastName. Avoid using one-character variables such as x, y, z. | class Employee { // variable int **id**; //code snippet } |
| Package | It should be a lowercase letter such as java, lang. If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang. | //package package **com.javatpoint;** class Employee { //code snippet } |
| Constant | It should be in uppercase letters such as RED, YELLOW. If the name contains multiple words, it should be separated by an underscore(\_) such as MAX\_PRIORITY. It may contain digits but not as the first letter. | class Employee { //constant static final int **MIN\_AGE** = 18; //code snippet } |

**Object and classes**

An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

**object**

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

1. **State**: It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behavior**: It is represented by methods of an object. It also reflects the response of an object with other objects.
3. **Identity**: It gives a unique name to an object and enables one object to interact with other objects.

**Example of an object: dog**

For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

* **An object is *a real-world entity*.**
* **An object is *a runtime entity*.**
* **The object is *an entity which has state and behavior*.**
* **The object is *an instance of a class*.**

**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.

**Object creation**

**Using new keyword:** It is the most common and general way to create an object.

Test t = new Test();

**also there are 3 others way to create objects but they all work internally.**

**initialization of object**

The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the class constructor.

public class Dog

{

    // Instance Variables

    String name;

    String breed;

    int age;

    String color;

    // Constructor Declaration of Class

    public Dog(String name, String breed,

                   int age, String color)

    {

        this.name = name;

        this.breed = breed;

        this.age = age;

        this.color = color;

    }

    // method 1

    public String getName()

    {

        return name;

    }

    // method 2

    public String getBreed()

    {

        return breed;

    }

    // method 3

    public int getAge()

    {

        return age;

    }

    // method 4

    public String getColor()

    {

        return color;

    }

    @Override

    public String toString()

    {

        return("Hi my name is "+ this.getName()+

               ".\nMy breed,age and color are " +

               this.getBreed()+"," + this.getAge()+

               ","+ this.getColor());

    }

    public static void main(String[] args)

    {

        Dog tuffy = new Dog("tuffy","papillon", 5, "white"); //object

        System.out.println(tuffy.toString());

    }

}

**Creating multiple by one type only.**

In real-time, we need different objects of a class in different methods. Creating a number of references for storing them is not a good practice and therefore we declare a static reference variable and use it whenever required. In this case, the wastage of memory is less. The objects that are not referenced anymore will be destroyed by [Garbage Collector](https://www.geeksforgeeks.org/garbage-collection-java/) of java.

class Animal {}

class Dog extends Animal {}

class Cat extends Animal {}

public class Test

{

// using Dog object

Animal obj = new Dog();

// using Cat object

obj = new Cat();

}

**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**

1. **class** <class\_name>{
2. field;
3. method;
4. }
6. //Java Program to demonstrate the working of a banking-system
7. //where we deposit and withdraw amount from our account.
8. //Creating an Account class which has deposit() and withdraw() methods
9. **class** Account{
10. **int** acc\_no;
11. String name;
12. **float** amount;
13. //Method to initialize object
14. **void** insert(**int** a,String n,**float** amt){
15. acc\_no=a;
16. name=n;
17. amount=amt;
18. }
19. //deposit method
20. **void** deposit(**float** amt){
21. amount=amount+amt;
22. System.out.println(amt+" deposited");
23. }
24. //withdraw method
25. **void** withdraw(**float** amt){
26. **if**(amount<amt){
27. System.out.println("Insufficient Balance");
28. }**else**{
29. amount=amount-amt;
30. System.out.println(amt+" withdrawn");
31. }
32. }
33. //method to check the balance of the account
34. **void** checkBalance(){System.out.println("Balance is: "+amount);}
35. //method to display the values of an object
36. **void** display(){System.out.println(acc\_no+" "+name+" "+amount);}
37. }
38. //Creating a test class to deposit and withdraw amount
39. **class** TestAccount{
40. **public** **static** **void** main(String[] args){
41. Account a1=**new** Account();
42. a1.insert(832345,"Ankit",1000);
43. a1.display();
44. a1.checkBalance();
45. a1.deposit(40000);
46. a1.checkBalance();
47. a1.withdraw(15000);
48. a1.checkBalance();
49. }}

**Methods**

**method** is a way to perform some task. Similarly, the **method in Java** is a collection of instructions that performs a specific task. It provides the reusability of code. We can also easily modify code using **methods**.

**what is method in java**

A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the **reusability** of code. We write a method once and use it many times. We do not require to write code again and again. It also provides the **easy modification** and **readability** of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it.



**Method Signature:** Every method has a method signature. It is a part of the method declaration. It includes the **method name** and **parameter list**.

**Access Specifier/modifiers:** Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides **four** types of access specifier:

* **Public:** The method is accessible by all classes when we use public specifier in our application.
* **Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.
* **Protected:** When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
* **Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

**Return Type:** Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword

**Naming a method**

method name should start with lowercase

Single-word method name: sum(), area()

Multi-word method name: areaOfCircle(), stringComparision()

**Types of method**

* **Predefined Method**
* **User-defined Method**

**predefined/built-in methods**

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**. We can directly use these methods just by calling them in the program at any point. Some pre-defined methods are **length(), equals(), compareTo(), sqrt(),** etc.

**When we call any of the predefined methods in our program, a series of codes related to the corresponding method runs in the background that is already stored in the library**.

**user-defined methods**

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

Method Calling

Once we have defined a method, it should be called. The calling of a method in a program is simple. When we call or invoke a user-defined method, the program control transfer to the called method.

A method returns to the code that invoked it when:

* It completes all the statements in the method
* It reaches a return statement
* Throws an exception

import java.io.\*;

// Class 1

// Helper class

class Addition {

    // Initially taking sum as 0

    // as we have not started computation

    int sum = 0;

    // Method

    // To add two numbers

    public int addTwoInt(int a, int b)

    {

        // Adding two integer value

        sum = a + b;

        // Returning summation of two values

        return sum;

    }

}

// Class 2

// Helper class

class GFG {

    // Main driver method

    public static void main(String[] args)

    {

        // Creating object of class 1 inside main() method

        Addition add = new Addition();

        // Calling method of above class

        // to add two integer

        // using instance created

        int s = add.addTwoInt(1, 2);

        // Printing the sum of two numbers

        System.out.println("Sum of two integer values :"

                           + s);

    }

}

**static and instance method**

**static method**

A method that has static keyword is known as static method. In other words, a method that belongs to a class rather than an instance of a class is known as a static method. We can also create a static method by using the keyword **static** before the method name.

The main advantage of a static method is that we can call it without creating an object.

mport java.io.\*;

class Geek {

    public static String geekName = "";

    public static void geek(String name)

    {

        geekName = name;

    }

}

class GFG {

    public static void main(String[] args)

    {

        // Accessing the static method geek()

        // and field by class name itself.

        Geek.geek("vaibhav");

        System.out.println(Geek.geekName);

        // Accessing the static method geek()

        // by using Object's reference.

        Geek obj = new Geek();

        obj.geek("mohit");

        System.out.println(obj.geekName);

    }

}

**instance method**

The method of the class is known as an **instance method**. It is a **non-static** method defined in the class. Before calling or invoking the instance method, it is necessary to create an object of its class.

import java.io.\*;

class Foo {

    String name = "";

    // Instance method to be called within the

    // same class or from a another class defined

    // in the same package or in different package.

    public void geek(String name) { this.name = name; }

}

class GFG {

    public static void main(String[] args)

    {

        // create an instance of the class.

        Foo ob = new Foo();

        // calling an instance method in the class 'Foo'.

        ob.geek("GeeksforGeeks");

        System.out.println(ob.name);

    }

}

**when to use static method**

* When you have code that can be shared across all instances of the same class, put that portion of code into static method.
* They are basically used to access static field(s) of the class

**Abstract method**

Sometimes, we require just method declaration in super-classes.

This can be achieve by specifying the [**abstract**](https://www.geeksforgeeks.org/abstract-keyword-in-java/) type modifier.

These methods are sometimes referred to as *subclasser responsibility* because they have no implementation specified in the super-class. Thus, a subclass must [override](https://www.geeksforgeeks.org/overriding-in-java/) them to provide method definition.

**rules for abstract method**

* Any class that contains one or more abstract methods must also be declared abstract
* The following are various **illegal combinations** of other modifiers for methods with respect to *abstract* modifier:
  1. final
  2. abstract native
  3. abstract synchronized
  4. abstract static
  5. abstract private
  6. abstract strictfp

abstract class A {

    // abstract method

    // it has no body

    abstract void m1();

    // concrete methods are still

    // allowed in abstract classes

    void m2()

    {

        System.out.println("This is "

                           + "a concrete method.");

    }

}

// concrete class B

class B extends A {

    // class B must override m1() method

    // otherwise, compile-time

    // exception will be thrown

    void m1()

    {

        System.out.println("B's "

                           + "implementation of m1.");

    }

}

// Driver class

public class AbstractDemo {

    public static void main(String args[])

    {

        B b = new B();

        b.m1();

        b.m2();

    }

}

**Constructors**

constructor is a block of code(similar to method) having same name as class name.

A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes.

constructor does not have any return type not even ‘**void’**

because it has nothing to return anything it just initialize the object of class.

Every time an object is created using the new() keyword, at least one constructor is called.

**Note:**

**1) only 4 modifiers are applicable for constructors i.e public, protected, default, & private**

**(static, final, synchronized cannot be used with constructor)**

**2) constructor execute automatically when we create an object of class.**

**use of constructor**

**constructor is used to initialize the object of class.(not to create an object)**

It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

**How Constructor is different from method.**

* Constructors must have the same name as the class within which it is defined it is not necessary for the method in Java.
* Constructors do not return any type while method(s) have the return type or **void** if does not return any value.
* Constructors are called only once at the time of Object creation while method(s) can be called any number of times.

**Why do we need constructor?**

class Employee{

String name;

in emp\_id;

p s v m(String[] args)

{

Employee e1=new Employee();

Employee e2=new Employee();

--

}

}

e1 e2

-there are 3 ways to initialize a object

1) by using reference variable.

2) by using method

3) by using constructor

1)

String name=’sagar’;

int emp\_id=101;

using this way we can only initialize for only one object or for every object same value will allocate, here name=sagar & emp\_id=101 will allocate to both Employee object e1 7& e1.

Employee e1=new Employee();

e1.name=’sagar’

e1.emp\_id=101;

Employee e2=new Employee();

e2.name=’AK’;

e2.emp\_id=’102’;

-this approach can be used resolved problem with reference variable but it increase unnecessary

lines of code.

3)

to resolved this problem we can used **Constructor.**

class Employee{

String name;

in emp\_id;

Employee(String name, int emp\_id){

this.name=name; creating constructor, parameterise constructor

this.emp\_id=emp\_id;

}

p s v m(String[] args)

{

Employee e1=new Employee(‘sagar’,101);

Employee e2=new Employee(‘abc’ 102);

--

}

}

-in this way constructor is used to initialise an object, and reduce no. of lines in code.

So constructors are used to assign values to the class variables at the time of object creation, either explicitly done by the programmer or by Java itself (default constructor).

**When constructor is called?**

Each time an object is created using a **new()** keyword, at least one constructor (it could be the default constructor) is invoked to assign initial values to the **data members**of the same class.

**Rules for creating constructor**

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized can only have publc, protected, private & default<>

**Syntax**

class Geek

{

.......

// A Constructor

Geek() {

}

.......

}

// We can create an object of the above class

// using the below statement. This statement

// calls above constructor.

Geek obj = new Geek();

**Types of Constructors**

* **No-argument constructor**
* **Parameterized Constructor**
* **Default Constructor**

**No-argument constructor**

A constructor that has no parameter is known as the No-argument or Zero argument constructor.

import java.io.\*;

class Geek {

    int num;

    String name;

    // this would be invoked while an object

    // of that class is created.

    Geek() { System.out.println("Constructor called"); }

}

class GFG {

    public static void main(String[] args)

    {

        // this would invoke default constructor.

        Geek geek1 = new Geek();

        // Default constructor provides the default

        // values to the object like 0, null

        System.out.println(geek1.name);

        System.out.println(geek1.num);

    }

}

**Parameterised constructor**

A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with our own values, then use a parameterized constructor.

import java.io.\*;

// Class 1

class Geek {

    // data members of the class.

    String name;

    int id;

    // Constructor would initialize data members

    // With the values of passed arguments while

    // Object of that class created

    Geek(String name, int id)

    {

        this.name = name;

        this.id = id;

    }

}

// Class 2

class GFG {

    // main driver method

    public static void main(String[] args)

    {

        // This would invoke the parameterized constructor.

        Geek geek1 = new Geek("adam", 1);

        System.out.println("GeekName :" + geek1.name

                           + " and GeekId :" + geek1.id);

    }

}

**Default constructor**

If we do not create any constructor, the Java compiler automatically create a no-arg constructor during the execution of the program. This constructor is called default constructor.

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

class Main {

int a;

boolean b;

public static void main(String[] args) {

// A default constructor is called

Main obj = new Main();

System.out.println("Default Value:");

System.out.println("a = " + obj.a);

System.out.println("b = " + obj.b);

}

}

O/P

0

false

Note:

Here, we haven't created any constructors. Hence, the Java compiler automatically creates the default constructor.

The default constructor initializes any uninitialized instance variables with default values.

**Constructor Overloading**

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.

class Main {

String language;

// constructor with no parameter

Main() {

this.language = "Java";

}

// constructor with a single parameter

Main(String language) {

this.language = language;

}

public void getName() {

System.out.println("Programming Langauage: " + this.language);

}

public static void main(String[] args) {

// call constructor with no parameter

Main obj1 = new Main();

// call constructor with a single parameter

Main obj2 = new Main("Python");

obj1.getName();

obj2.getName();

}

}

**O/P**

Programming Language: Java

Programming Language: Python

**Note:**

In the above example, we have two constructors: Main() and Main(String language). Here, both the constructor initialize the value of the variable language with different values.

Based on the parameter passed during object creation, different constructors are called and different values are assigned

**Call one constructor from another constructor/Constructor chaining**

Constructor chaining is the process of calling one constructor from another constructor with respect to current object.

One of the main use of constructor chaining is to avoid duplicate codes while having multiple constructor (by means of constructor overloading) and make code more readable.

Channing can be done in two ways

* **Within same class**: It can be done using **this()** keyword for constructors in the same class
* **From base class:**by using **super()** keyword to call the constructor from the base class.

Constructor chaining occurs through **inheritance**. A sub-class constructor’s task is to call super class’s constructor first. This ensures that the creation of sub class’s object starts with the initialization of the data members of the superclass. There could be any number of classes in the inheritance chain. Every constructor calls up the chain till the class at the top is reached.

**why do we need constructor chaining?**

This process is used when we want to perform multiple tasks in a single constructor rather than creating a code for each task in a single constructor we create a separate constructor for each task and make their chain which makes the program more readable.

**Within same class**

if we want to make communication between constructor of same class we need to use **this()** method

this() method is use to call another constructor of same class.

public class ConstructorChain

{

//default constructor

ConstructorChain()

{

this("Javatpoint");

System.out.println("Default constructor called.");

}

//parameterized constructor

ConstructorChain(String str)

{

System.out.println("Parameterized constructor called");

}

//main method

public static void main(String args[])

{

//initializes the instance of example class

ConstructorChain cc = new ConstructorChain();

}

}

O/P

Parameterized constructor called

Default constructor called

**Note**:

in the above example, we have created an instance of the class without passing any parameter. It first calls the default constructor and the default constructor redirects the call to the parameterized one because of this(). The statements inside the parameterized constructor are executed and return back to the default constructor. After that, the rest of the statements in the default constructor is executed and the object is successfully initialized.

ConstructorChain cc = **new** ConstructorChain(); -> ConstructorChain() -> ConstructorChain(String str) -> System.out.println() -> ConstructorChain() -> System.out.println()

**Calling super class constructor**

Sometimes, we need to call the superclass (parent class) constructor from the child class (derived class) in such cases, we use the super() keyword in the derived class constructor. It is optional to write super() because JVM automatically puts it. It should always write in the first line. We get a syntax error if we try to call a superclass constructor in the child class.

class Demo

{

//base class default constructor

Demo()

{

this(80, 90);

System.out.println("Base class default constructor called");

}

//base class parameterized constructor

Demo(int x, int y)

{

System.out.println("Base class parameterized constructor called");

}

}

//derived class or child class

class Prototype extends Demo

{

//derived class default constructor

Prototype()

{

this("Java", "Python");

System.out.println("Derived class default constructor called");

}

//derived class parameterized constructor

Prototype(String str1, String str2)

{

super();

System.out.println("Derived class parameterized constructor called");

}

}

public class ConstructorChaining

{

//main method

public static void main(String args[])

{

//initializes the instance of example class

Prototype my\_example = new Prototype();

}

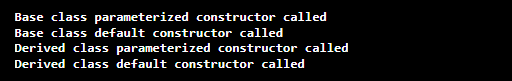
}

in above example, Prototype() call no-arg prototype constructor after that no-arg prototype constructor call this() method due to which arg type constructor got call no-arg default constructor & because of this() method it will call arg type default constructor and print

then control go back to default constructor and print

after control again go back to prototype() constructor and back to default prototype constructor.

O/P



**Note:**

1)Constructor can call other constructors of the same or superclass

2)Constructors call from a constructor must be the first line(this() or super() method must be at first line).

3)such series of invocation of constructors is known as constructor chaining

**super() or this()**

-first line constructor is either super() or this() –by default super().

-Constructor never contain super() or this() both.

**Copy Constructors**

**Inheritance**

**Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object.

The idea behind inheritance in Java is that you can create new [classes](https://www.javatpoint.com/object-and-class-in-java) that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

-it is-A-relationship

-code reusability

-extends keyword (implement inheritance)

* **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
* **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

1. **class** Subclass-name **extends** Superclass-name
2. {
3. //methods and fields
4. }

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

eg

class P

{

public void m1()

{

sopln(“parent”);

}

}

class C extends P

{

public void m2()

{

sopln(“”child”);

}

}

-**one method is available for parent class**

**-two methods are available for child class**

**Note**

members (method, variables) of parent class by default available to child class

**creating object and calling objects**

class Test

{

p s v m(String[] args)

{

P p=new P();

p.m1(); //possible

p.m2(); //not possible

}

}

-**on parent reference(p) we can call only methods which are present in parent class but not methods which are present in child class**

**-on parent reference we can’t call child specific members**

C c=new C();

c.m1(); //possible

c.m2(); //possible

-**on child reference we can call both parent and child specific members.**

P p=new C();

p.m1();

p.m2();

-**parent reference can be used to call child class object, but using that reference we cant call child specific methods.**

C c new P(); //not possible

-**child reference can’t be use to call parent object.**

**Importance of Inheritance**

-**code reusability**

**-reduce development time**

**-all java API classes are implemented using inheritance.**

class Bicycle {

    // the Bicycle class has two fields

    public int gear;

    public int speed;

    // the Bicycle class has one constructor

    public Bicycle(int gear, int speed)

    {

        this.gear = gear;

        this.speed = speed;

    }

    // the Bicycle class has three methods

    public void applyBrake(int decrement)

    {

        speed -= decrement;

    }

    public void speedUp(int increment)

    {

        speed += increment;

    }

    // toString() method to print info of Bicycle

    public String toString()

    {

        return ("No of gears are " + gear + "\n"

                + "speed of bicycle is " + speed);

    }

}

// derived class

class MountainBike extends Bicycle {

    // the MountainBike subclass adds one more field

    public int seatHeight;

    // the MountainBike subclass has one constructor

    public MountainBike(int gear, int speed,

                        int startHeight)

    {

        // invoking base-class(Bicycle) constructor

        super(gear, speed);

        seatHeight = startHeight;

    }

    // the MountainBike subclass adds one more method

    public void setHeight(int newValue)

    {

        seatHeight = newValue;

    }

    // overriding toString() method

    // of Bicycle to print more info

    @Override public String toString()

    {

        return (super.toString() + "\nseat height is "

                + seatHeight);

    }

}

// driver class

public class Test {

    public static void main(String args[])

    {

        MountainBike mb = new MountainBike(3, 100, 25);

        System.out.println(mb.toString());

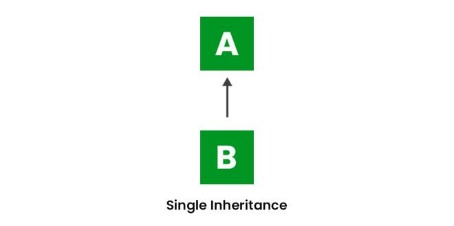
    }

}

**Types of Inheritance**

**1 Single inheritance**

1. single Inheritance



class B extends A

class one {

    public void print\_geek()

    {

        System.out.println("Geeks");

    }

}

class two extends one {

    public void print\_for() { System.out.println("for"); }

}

// Driver class

public class Main {

    public static void main(String[] args)

    {

        two g = new two();

        g.print\_geek();

        g.print\_for();

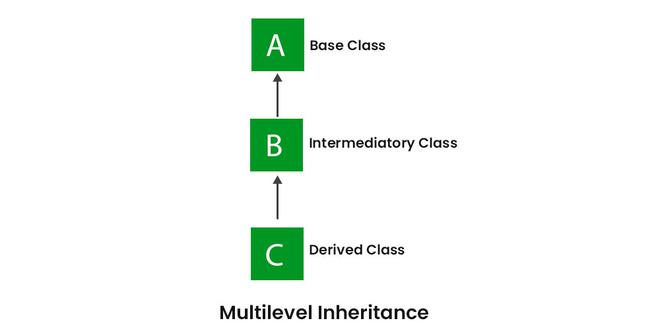
        g.print\_geek();

    }

}

**2 Multilevel inheritance**

In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.



class C extends B

class B extends A

class one {

    public void print\_geek()

    {

        System.out.println("Geeks");

    }

}

class two extends one {

    public void print\_for() { System.out.println("for"); }

}

class three extends two {

    public void print\_geek()

    {

        System.out.println("Geeks");

    }

}

// Drived class

public class Main {

    public static void main(String[] args)

    {

        three g = new three();

        g.print\_geek();

        g.print\_for();

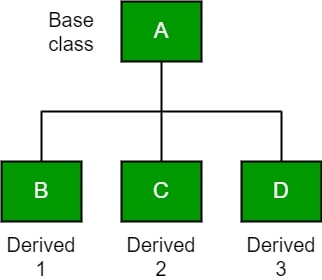
        g.print\_geek();

    }

}

**3 Hierarchal Inheritance**

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived classes B, C, and D.



class A {

    public void print\_A() { System.out.println("Class A"); }

}

class B extends A {

    public void print\_B() { System.out.println("Class B"); }

}

class C extends A {

    public void print\_C() { System.out.println("Class C"); }

}

class D extends A {

    public void print\_D() { System.out.println("Class D"); }

}

// Driver Class

public class Test {

    public static void main(String[] args)

    {

        B obj\_B = new B();

        obj\_B.print\_A();

        obj\_B.print\_B();

        C obj\_C = new C();

        obj\_C.print\_A();

        obj\_C.print\_C();

        D obj\_D = new D();

        obj\_D.print\_A();

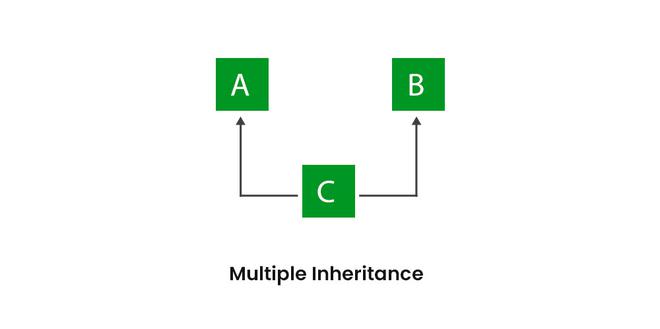
        obj\_D.print\_D();

    }

}

**Note:**

1 Multiple Inheritance



-here parent classes are multiple, single child class inherit multiple parent class at a time.

-java doesn’t support multiple inheritance.

**2 Hybrid Inheritance**

-it is a combination of inheritance.

-java doesn’t support hybrid inheritance

**4 Multiple Inheritance (Through inheritance)**

In java, we can achieve multiple inheritances only through [Interfaces](https://www.geeksforgeeks.org/interfaces-in-java/).

interface two {

    public void print\_for();

}

interface three extends one, two {

    public void print\_geek();

}

class child implements three {

    @Override public void print\_geek()

    {

        System.out.println("Geeks");

    }

    public void print\_for() { System.out.println("for"); }

}

// Drived class

public class Main {

    public static void main(String[] args)

    {

        child c = new child();

        c.print\_geek();

        c.print\_for();

        c.print\_geek();

    }

}

**5 Hybrid Inheritance (through Interface)**

-it is the combination of two or more types of inheritance, it can only support through interface.

**Important facts about inheritance in Java**

* **Default superclass**: Except [Object](https://www.geeksforgeeks.org/object-class-in-java/) class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of the [Object](https://www.geeksforgeeks.org/object-class-in-java/) class.
* **Superclass can only be one:** A superclass can have any number of subclasses. But a subclass can have only **one** superclass. This is because Java does not support [multiple inheritances](https://www.geeksforgeeks.org/java-and-multiple-inheritance/) with classes. Although with interfaces, multiple inheritances are supported by java.
* **Inheriting Constructors:**A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.
* **Private member inheritance:** A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods(like getters and setters) for accessing its private fields, these can also be used by the subclass.

**Why java doesn’t support multiple inheritance.**

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

class A{

void msg(){System.out.println("Hello");}

}

class B{

void msg(){System.out.println("Welcome");}

}

class C extends A,B{//suppose if it were

public static void main(String args[]){

C obj=new C();

obj.msg();//Now which msg() method would be invoked?

}

}

O/P compile time error.

**Aggregation**

**When an object A contains a reference to another object B or we can say Object A has a HAS-A relationship with Object B, then it is termed as Aggregation**. Aggregation helps in reusing the code. Object B can have utility methods and which can be utilized by multiple objects.

public class Address {

String city,state,country;

public Address(String city, String state, String country) {

this.city = city;

this.state = state;

this.country = country;

}

}

Address.java

public class Emp {

int id;

String name;

Address address;

public Emp(int id, String name,Address address) {

this.id = id;

this.name = name;

this.address=address;

}

void display(){

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

System.out.println(address.city+" "+address.state+" "+address.country);

}

public static void main(String[] args) {

Address address1=new Address("gzb","UP","india");

Address address2=new Address("gno","UP","india");

Emp e=new Emp(111,"varun",address1);

Emp e2=new Emp(112,"arun",address2);

e.display();

e2.display();

}

}

Emp.java

Emp has –A relation with Address.

-if we have a class and already created then we can use that class with reference to another class

-here Address class is use in Emp class.

**Polymorphism**

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations. This is called polymorphism.   
Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word “poly” means many and “morphs” means forms, So it means many forms.

In Java polymorphism is mainly divided into two types:

* Compile-time Polymorphism
* Runtime Polymorphism

Poly= many

Morphs=forms

Method overriding, method overloading comes under polymorphism

**Polymorphism**

**Static| compile time| early binding dynamic| runtime| late binding**

**Overloading method hiding overriding**

1 **Compile time polymorphism**

It is also known as static polymorphism. This type of polymorphism is achieved by method overloading or operator overloading. (operator overloading doesn’t support in java).

**Method Overloading**

If a [class](https://www.javatpoint.com/object-and-class-in-java) has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

**Method overloading increases the readability of the program.**

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

-java doesn’t support by changing data type

In java, method overloading is not possible by changing the return type of the method only because of ambiguity

**Case 1**

class Test

{

public void m1(){

sopln(“no arg method”);

}

public void m1(int i);{

sopln(“int arg method”);

}

public void m1(double d){

sopln(“double arg method”);

}

}

p s v main(String[] args);{

Test t=new Test();

t.m1(); //no arg method

t.m1(10); // int arg method

t.m1(10.5); // double arg method

**Case 2**

Automatic promotion of argument types in method overloading



-in this case if defined argument type is doesn’t contain in calling method then compiler will promote that argument type and display output.

class Test

{

public void m1(int i){

sopln(“int argument type”);

}

public void m1(float){

sopln(“float argument type”);

}

p s v m(String[] args){

Test t=new Test();

t.m1(10); //int argument type

t.m1(10.5); // float argument type

t.m1(“abc”); //promoted to int argument type

t.m1(10.5d) // error as double can’t be promote

**Case 3**

class Test

{

Public void m1(object o);{

Sopln(“object version”);

}

Public void m1(String s);{

Sopln(“string version”)

}

P s v m(String[] args){

Test t=new Test();

t.m1(new object); // object version

t.m1(“Durga”); //string version

t.m1(null); //string version

}

Null is valid for any type of object reference

For null

Object -parent

String -child

**If parent and child argument are there then child will get preference**

**Case 4**

class Test

{

Public void m1(string s){

Sopln(“string version”);

}

Public void m1(stringBuffer sb){

Sopln(“stringbuffer version”);

}

}

P s v m(String[] args){

Test t=new Test();

t.m1(“durga”); //string version

t.m1(new stringBuffer(“durga”); //stringBuffer version

t.m1(null); //CE

**Note: if both methods arguments match and there is no parent child relation then CE will get**.

Object

String stringBuffer

**Exact match will get highest priority**

**Case 5**

class Test

{

Public void m1(int i, float f){

Sopln(“int-float version”);

}

Public void m1(float f, int i){

Sopln(“float-int version”);

}

P s v m(String[] args);

Test t=new Test();

t.m1(10, 10.5f); //int-float version

t.m1(10.5f, 10); // float-int version

t.m1(10,10); //CE

t.m1(10.5f,10.5f); //CE

}

**Case 6**

Class Animal

{  
}

Class Monkey extends Animal{

}

Class Test

{

Public void m1(Animal a){

Sopln(“animal version”)

}

Public void m1(monkey m){

Sopln(“monkey version”)

}

P s v m(String[] args){

Test t=new Test();

Animal a=new Animal();

t.m1(a); //animal version

Monkey m=new Monkey();

t.m1(m); //monkey version

Animal a1=new Monkey();

t.m1(a1); // animal version

}

}

**In method overloading method resolution is always take care by compiler based on reference type, runtime object never play any role.**

**Overloading also known as compile time polymorphism, static polymorphism or early binding.**

**Method Overriding**

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Sometimes child class may not satisfy parent method implementation, then child is allowed to redefined method according to its requirement it is call method overriding.

Used of method overriding

* Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
* Method overriding is used for runtime polymorphism

Method overriding is also known as runtime polymorphism, dynamic polymorphism or late binding.

#### **Rules for Java Method Overriding**

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

**Note:** if we want to change a method which is already declare in parent class then should go for method overriding.

class P

{

Public void property(){

Sopln(“cash+gold”);

}

Public void marry(){

Sopln(“sita”+”gita”);

}

}

Class C extends P

{

Public void marry(){

Sopln(“sunny leone”)

}

}

Class Test

{

P s v m(String[] args){

P p=new P();

p.marry(); //parent method

C c=new C();

c.marry(); //child method

P p1=new C();

p1.marry(); //child method

}

}

**Eg**:

class Bank{

int getRateOfInterest(){return 0;}

}

//Creating child classes.

class SBI extends Bank{

int getRateOfInterest(){return 8;}

}

class ICICI extends Bank{

int getRateOfInterest(){return 7;}

}

class AXIS extends Bank{

int getRateOfInterest(){return 9;}

}

//Test class to create objects and call the methods

class Test2{

public static void main(String args[]){

SBI s=new SBI();

ICICI i=new ICICI();

AXIS a=new AXIS();

System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());

System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());

System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());

}

}

**How method resolution is works in overriding**

in overriding method resolution is always take care by JVM based on runtime object reference type.

**Note:**

**While overriding we can’t reduce scope of modifiers but can increase**.

Parent public protected default

Child public protected/public default/protected/public

**Note:**

1 we can’t override static method.

It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

-therefore we also can’t override main method.

**Method Hiding**

**Method hiding** can be defined as, "if a subclass defines a static method with the same signature as a static method in the super class, in such a case, the method in the subclass hides the one in the superclass." The mechanism is known as **method hiding**. It happens because static methods are resolved at compile time.

Static methods are bonded during compile time using types of reference variables not object. We know that static methods are accessed by using the class name rather than an object. Note that the static method can be overloaded, but cannot be overridden in Java.

when parent and child both methods static then such overriding is call is method hiding.

in this method resolution is take care by compiler base on reference type.

class Complex {

    public static void f1()

    {

        System.out.println(

            "f1 method of the Complex class is executed.");

    }

}

// class child extend Demo class

class Sample extends Complex {

    public static void f1()

    {

        System.out.println(

            "f1 of the Sample class is executed.");

    }

}

public class Main {

    public static void main(String args[])

    {

        Complex d1 = new Complex();

        // d2 is reference variable of class Demo that

        // points to object of class Sample

        Complex d2 = new Sample();

        // But here method will be call using type of

        // reference

        d1.f1();

        d2.f1();

    }

}

**O/P**

f1 method of the Complex class is executed.

f1 method of the Complex class is executed.

**method resolution is take care by compiler, here reference d1 & d2 are of Complex i’e parent class therefore it will call parent class f1 method.**

**Difference between method overriding and hiding**.

|  |  |
| --- | --- |
| **Method Hiding** | **Method Overriding** |
| both parent and child should be static | both parent and child should be non-static |
| method resolution always take care by compiler base on reference type | method resolution always take by JVM base on runtime object |
| best known as compile time polymorphism, static, early biding | best known as runtime polymorphism dynamic or late biding. |

**Overriding wrt var-arg methods**

method resolution is always takes by JVM base on runtime object.

public void m1(int ….i);

{

sopln(“parent”);

}

class C extends P

{

public void m1(int …..i);

{

sopln(‘child”);

}

P p=new P();

C c=new C();

P p1=new C();

p.m1(); //parent

c.m1(); //child

p1.m1(); //child

child method should always be var-args method.

**Difference between overloading & overriding**

|  |  |  |
| --- | --- | --- |
| Property | Overloading | Overriding |
| method name | must be same | must be same |
| argument type | must be different(at least order) | must be different(including order) |
| private/final static method | can be overload | cannot override |
| return type | no restriction | return type of child and parent must be same. |
| method resolution | compiler based on reference type | JVM based on object method |
| other name | C T polymorphism, static, early biding | R T polymorphism, dynamic, late biding. |

**Encapsulation**

**Package**

A **java package** is a group of similar types of classes, interfaces and sub-packages.

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 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.



Naming conflict- decreases

Maintainability –increases

Security –increases

**Syntax**

package mypack;

public class Simple{

public static void main(String args[]){

System.out.println("Welcome to package");

}

}

**Access package from one to another**

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

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

**1 using package.\*; statement**

If you use package.\* then all the classes and interfaces of this package will be accessible but not subpackages.

package pack;

public class A{

public void msg(){System.out.println("Hello");}

}

package mypack;

import pack.\*;

class B{

public static void main(String args[]){

A obj = new A();

obj.msg();

}

}

**2 using packagename.classname; statement**

If you import package.classname then only declared class of this package will be accessible.

package pack;

public class A{

public void msg(){System.out.println("Hello");}

}

package mypack;

import pack.A;

class B{

public static void main(String args[]){

A obj = new A();

obj.msg();

}

}

**3 using fully qualified name.**

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

package pack;

public class A{

public void msg(){System.out.println("Hello");}

}

package mypack;

class B{

public static void main(String args[]){

pack.A obj = new pack.A();//using fully qualified name

obj.msg();

}

}

**Note:**

-if you are using import statement then all the classes & interface of that package will get imported not sub-packages.

-in java source file at most one package is allowed.

-package statement must be first statement.

-any import statement can be possible.

**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 Java access modifiers:**

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **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.
3. **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.
4. **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.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 |

**1 Private**

The private access modifier is specified using the keyword **private**.

* The methods or data members declared as private are accessible only **within the class** in which they are declared.
* Any other **class of**the **same package will not be able to access** these members.

package p1;

class A

{

private void display()

    {

        System.out.println("GeeksforGeeks");

    }

}

class B

{

public static void main(String args[])

    {

        A obj = new A();

        // Trying to access private method

        // of another class

        obj.display();

    }

}

**2 Default**

When no access modifier is specified for a class, method, or data member – It is said to be having the **default** access modifier by default.

* The data members, class or methods which are not declared using any access modifiers i.e. having default access modifier are accessible **only within the same package**.

package p1;

// Class Geeks is having Default access modifier

class Geek

{

    void display()

    {

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

    }

}

package p2;

import p1.\*;

// This class is having default access modifier

class GeekNew

{

    public static void main(String args[])

    {

        // Accessing class Geek from package p1

        Geeks obj = new Geek();

        obj.display();

    }

}

O/P: compilation error

**3 Protected**

* The methods or data members declared as protected are **accessible within the same package or subclasses in different packages.**

if variable/method is declare as protected then it can be **accessible anywhere within same package** but **only in child method/class of another package.**

**protected=<default> + child**

eg:

package pack1;

public class A{

protected void m1(){

sopln(“A class protected method”);

}

}

public class B extends A{

p s v m(String[] args){

A a=new A();

a.m1();

B b=new B();

b.m1();

A a1=new B();

a1.m1();

}

}

-since child class is also in the same package, we can use protected method with parent reference, child reference and parent reference with child method also.

-but child class is outside the package then we can only use child reference for the method.

Eg:

package pack1;

public class A{

protected void m1(){

sopln(“A class is protected method”);

}

}

package pack2;

import pack1;

public class B extends A{

p s v m(String[] args){

A a=new A();

a.m1();

B b=new B(); work fine

b.m1(); give CE

A a1=new B();

a1.m1();

}

}

**4 Public**

The public access modifier is specified using the keyword **public**.

* The public access modifier has the **widest scope** among all other access modifiers.
* Classes, methods, or data members that are declared as public are **accessible from everywhere** in the program. There is no restriction on the scope of public data members.

package p1;

public class A

{

public void display()

    {

        System.out.println("GeeksforGeeks");

    }

}

package p2;

import p1.\*;

class B {

    public static void main(String args[])

    {

        A obj = new A();

        obj.display();

    }

}

Summary of public, protected and <default>

|  |  |  |  |
| --- | --- | --- | --- |
| **visibility** | **public** | **protected** | **<default>** |
| Within the same class | yes | yes | yes |
| From child class of same package | yes | yes | yes |
| From non-child class of same package | yes | yes | yes |
| From child class of another package | yes | yes(only if reference is also child class) | no |
| From non-child class of another package | yes | no | no |

**Note:**

-recommended modifier for variable=private

-recommended modifier for method=public

-recommended modifier for class=public

private=class level

<default>=package level

public=global level

**Encapsulation**

**Encapsulation in Java** is a *process of wrapping code and data together into a single unit*, for example, a capsule which is mixed of several medicines.

**Encapsulation** is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is, that it is a protective shield that prevents the data from being accessed by the code outside this shield.

* Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of its own class in which it is declared.
* As in encapsulation, the data in a class is hidden from other classes using the data hiding concept which is achieved by making the members or methods of a class private, and the class is exposed to the end-user or the world without providing any details behind implementation using the abstraction concept, so it is also known as a **combination of data-hiding and abstraction**.
* Encapsulation can be achieved by Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.
* It is more defined with the setter and getter method.

**Advantages of Encapsulation**

* **Data Hiding:**it is a way of restricting the access of our data members by hiding the implementation details. Encapsulation also provides a way for data hiding. The user will have no idea about the inner implementation of the class. It will not be visible to the user how the class is storing values in the variables. The user will only know that we are passing the values to a setter method and variables are getting initialized with that value.
* **Increased Flexibility:** We can make the variables of the class read-only or write-only depending on our requirement. If we wish to make the variables read-only then we have to omit the setter methods like setName(), setAge(), etc. from the above program or if we wish to make the variables write-only then we have to omit the get methods like getName(), getAge(), etc. from the above program
* **Reusability:** Encapsulation also improves the re-usability and is easy to change with new requirements.
* **Testing code is easy:** Encapsulated code is easy to test for unit testing.

Program without encapsulation

class Area {

  int length;

  int breadth;

  // constructor to initialize values

  Area(int length, int breadth) {

    this.length = length;

    this.breadth = breadth;

  }

  // method to calculate area

  public void getArea() {

    int area = length \* breadth;

    System.out.println("Area: " + area);

  }

}

class Main {

  public static void main(String[] args) {

    Area rectangle = new Area(2, 16);

    rectangle.getArea();

  }

}

Program with encapsulation

class Area{

private int length;

private int breadth;

public int getLength()

{

return length;

}

public int getBreadth()

{

return breadth;

}

public int getArea()

{

return length\*breadth;

}

public void setLength(int newLength)

{

length=newLength;

}

public void setBreadth(int newBreadth)

{

breadth=newBreadth;

}

public class Main{

public static void main(String[] args){

Area obj=new Area()

obj.setLength(17);

obj.setBreadth(14);

System.out.print(“Area of rectangle”+ getArea());

}

}

**eg 2**

class Account {

//private data members to hide the data

private long acc\_no;

private String name,email;

private float amount;

//public getter and setter methods

public long getAcc\_no() {

    return acc\_no;

}

public void setAcc\_no(long acc\_no) {

    this.acc\_no = acc\_no;

}

public String getName() {

    return name;

}

public void setName(String name) {

    this.name = name;

}

public String getEmail() {

    return email;

}

public void setEmail(String email) {

    this.email = email;

}

public float getAmount() {

    return amount;

}

public void setAmount(float amount) {

    this.amount = amount;

}

}

public class GFG {

public static void main(String[] args) {

    //creating instance of Account class

    Account acc=new Account();

    //setting values through setter methods

    acc.setAcc\_no(7310805450L);

    acc.setName("MD FAIZ");

    acc.setEmail("mdfaiz689@gmail.com");

    acc.setAmount(100000f);

    //getting values through getter methods

    System.out.println(acc.getAcc\_no()+" "+acc.getName()+" "+acc.getEmail()+" "+acc.getAmount());

}

}

**Abstraction**

Abstraction is hiding of internal implementation & just highlighting the setup services that we are offering.

data abstraction deals with exposing the interface to the end user and hiding the details of implementation.

**Data Abstraction** is the property by virtue of which only the essential details are displayed to the user. The trivial or the non-essential units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.

Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviours of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of a car or applying brakes will stop the car, but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc. in the car. This is what abstraction is.

**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.

Abstraction lets you focus on what the [object](https://www.javatpoint.com/object-and-class-in-java) does instead of how it does it.

**There are two ways to achieve abstraction in java**

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

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.  
Abstraction can be achieved with either **abstract classes** or [**interfaces**](https://www.w3schools.com/java/java_interface.asp)

The abstract keyword is a non-access modifier, used for classes and methods:

**1 using abstract class**

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.

**Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

**Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

abstract class Animal {

public abstract void animalSound();

public void sleep() {

System.out.println("Zzz");

}

}

**Rules for creating Abstract class.**

1. An instance of an abstract class cannot be created.
2. Constructors are allowed.
3. We can have an abstract class without any abstract method.
4. There can be a **final method** in abstract class but any abstract method in class(abstract class) cannot be declared as final  or in simpler terms final method cannot be abstract itself as it will yield an error: “Illegal combination of modifiers: abstract and final”
5. We can define static methods in an abstract class
6. We can use the **abstract keyword** for declaring ***top-level classes (Outer class) as well as inner classes*** as abstract
7. If a**class** contains at least **one abstract method**then compulsory should declare a class as abstract
8. If the**Child class** is unable to provide implementation to all abstract methods of the**Parent class**then we should declare that **Child class as abstract**so that the next level Child class should provide implementation to the remaining abstract method

Note:

1 a method without body is known as abstract class.

2 implementation of abstract class is done by extended class.

3 abstract method in abstract class are meant to be overridden in derived classes otherwise compile-time error will occur.

// Abstract class

abstract class Animal {

// Abstract method (does not have a body)

public abstract void animalSound();

// Regular method

public void sleep() {

System.out.println("Zzz");

}

}

// Subclass (inherit from Animal)

class Pig extends Animal {

public void animalSound() {

// The body of animalSound() is provided here

System.out.println("The pig says: wee wee");

}

}

class Main {

public static void main(String[] args) {

Pig myPig = new Pig(); // Create a Pig object

myPig.animalSound();

myPig.sleep();

}

}

Use of abstract class & abstract methods.

To achieve security - hide certain details and only show the important details of an object.

**2 using interface**

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

An interface is a completely "**abstract class**" that is used to group related methods with empty bodies:

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

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.

Note:

#### **The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.**



#implementation with abstract class

interface GFG {

    void learnCoding();

    void learnProgrammingLanguage();

    void contribute();

}

// Abstract class Student implementing from GFG interface

abstract class Student implements GFG {

    // Overriding the methods

    @Override public void learnCoding()

    {

        System.out.println(

            "Let's make coding a habit with GFG");

    }

    @Override public void learnProgrammingLanguage()

    {

        System.out.println(

            "Let's master all fundamentals of java with the help of GFG");

    }

}

// Extend the GEEK class by Student abstract class

class GEEK extends Student {

    @Override public void contribute()

    {

        System.out.println(

            "Now let's help others by contributing in GFG");

    }

}

// Driver code

public class Main {

    public static void main(String[] args)

    {

        // New GEEK object is created

        GEEK gfgStudent = new GEEK();

        // Calls to the multiple functions

        gfgStudent.learnCoding();

        gfgStudent.learnProgrammingLanguage();

        gfgStudent.contribute();

    }

}

#implementation with non-abstract class

// Interface

interface Animal {

public void animalSound(); // interface method (does not have a body)

public void sleep(); // interface method (does not have a body)

}

// Pig "implements" the Animal interface

class Pig implements Animal {

public void animalSound() {

// The body of animalSound() is provided here

System.out.println("The pig says: wee wee");

}

public void sleep() {

// The body of sleep() is provided here

System.out.println("Zzz");

}

}

class Main {

public static void main(String[] args) {

Pig myPig = new Pig(); // Create a Pig object

myPig.animalSound();

myPig.sleep();

}

}

#### **Notes on Interfaces:**

* Like **abstract classes**, interfaces **cannot** be used to create objects (in the example above, it is not possible to create an "Animal" object in the MyMainClass)
* Interface methods do not have a body - the body is provided by the "implement" class
* On implementation of an interface, you must override all of its methods
* Interface methods are by default abstract and public
* Interface attributes are by default public, static and final
* An interface cannot contain a constructor (as it cannot be used to create objects)

#### **Why And When To Use Interfaces?**

1) To achieve security - hide certain details and only show the important details of an object (interface).

2) Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces.

interface FirstInterface {

public void myMethod(); // interface method

}

interface SecondInterface {

public void myOtherMethod(); // interface method

}

class DemoClass implements FirstInterface, SecondInterface {

public void myMethod() {

System.out.println("Some text..");

}

public void myOtherMethod() {

System.out.println("Some other text...");

}

}

class Main {

public static void main(String[] args) {

DemoClass myObj = new DemoClass();

myObj.myMethod();

myObj.myOtherMethod();

}

}

**Difference between abstract class & interface.**

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 6) An **abstract class** can extend another Java class and implement multiple Java interfaces. | An **interface** can extend another Java interface only. |
| 7) An **abstract class** can be extended using keyword "extends". | An **interface** can be implemented using keyword "implements". |
| 8) A Java **abstract class** can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| 9)**Example:** public abstract class Shape{ public abstract void draw(); } | **Example:** public interface Drawable{ void draw(); } |

**Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).**