

**Hitesh Vasant Manjare**

**(software developer)**

**Basic example**

**Example :**

**BasicExample.java**

package com.test.example;

public class BasicExample

{

public static void main(String[] args)

{

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

}

}

**Output :**

Hello World

**Questions :**

1. Explain **public static void main(String[] args) ?**



|  |  |
| --- | --- |
| **public** | **Access modifier** |
| **static** | **keyword** |
| **void** | **Return type** |
| **main** | **method** |
| **String[] args** | **Command line argument** |

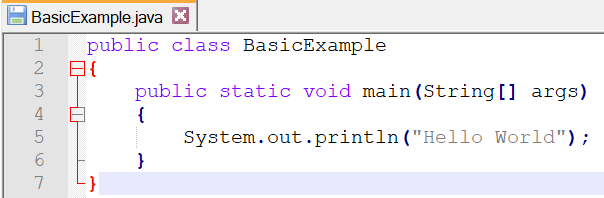
1. Explain **System.*out*.println("Hello World");**



|  |  |
| --- | --- |
| **System** | **class** |
| ***out*** | **object** |
| **println** | **method** |

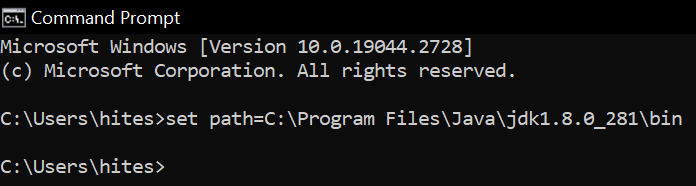
**Step 1 : write java code in .txt file**

**Run program by cmd**



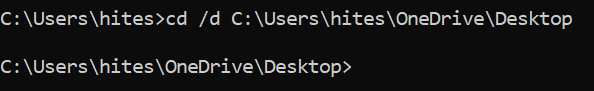
**Step 2 : open cmd and set path to jdk**

|  |  |
| --- | --- |
| **command** | **set path=C:\Program Files\Java\jdk1.8.0\_281\bin** |



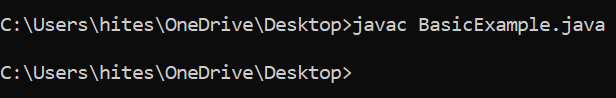
**Step 3 : change path to where java file is stored**

|  |  |
| --- | --- |
| **command** | **cd /d C:\Users\hites\OneDrive\Desktop** |



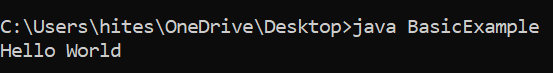
**Step 4 : compile java code**

|  |  |
| --- | --- |
| **command** | **javac BasicExample.java** |



**Step 5 : run java code**

|  |  |
| --- | --- |
| **command** | **java BasicExample** |



**Command line arguments**

**Eclipse IDE**

**Example :**

**Test.java**

package com.test.example;

public class Test

{

public static void main(String[] args)

{

System.*out*.println("At position 1 : " + args[1]);

System.*out*.println("--------------------------- ");

for (String s : args)

{

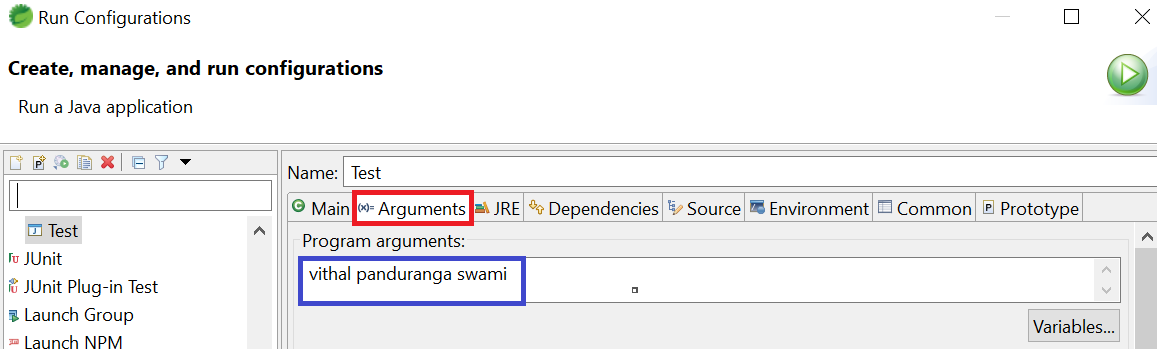
System.*out*.println("arguments is : " + s);

}

}

}

**Right click on** **projectName** **> Run as > Run configurations > Arguments**



**Output :**

At position 1 : panduranga

---------------------------

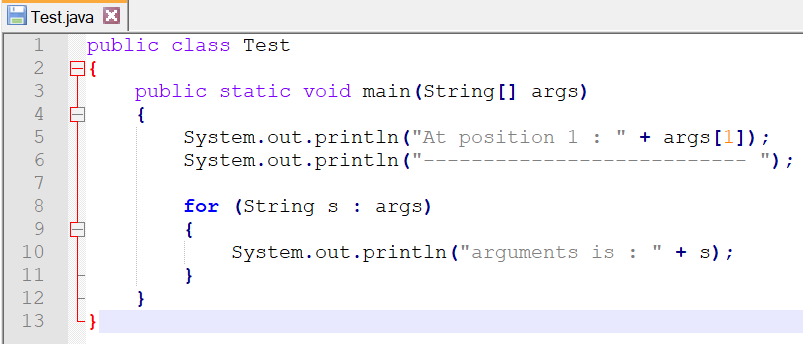
arguments is : vithal

arguments is : panduranga

arguments is : swami

**cmd**

**Example :**



**compile java code**

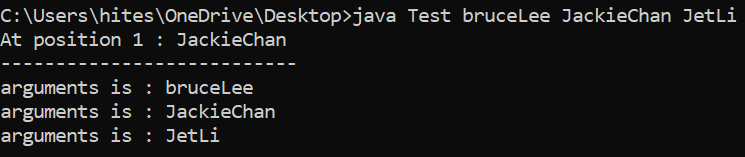
|  |  |
| --- | --- |
| **command** | **javac Test.java** |



**run java code**

|  |  |
| --- | --- |
| **command** | **java Test bruceLee JackieChan JetLi** |

**Output :**



**Write multiple class in single class**

**Example :**

**Test.java**

package com.test.example;

public class Test // this must be public

{

public static void main(String[] args)

{

System.*out*.println("Test is public class");

A a = new A();

a.m1();

B b = new B();

b.m2();

}

}

class A // this must be default

{

public void m1()

{

System.*out*.println("A is default class");

}

}

class B // this must be default

{

public void m2()

{

System.*out*.println("B is default class");

}

}

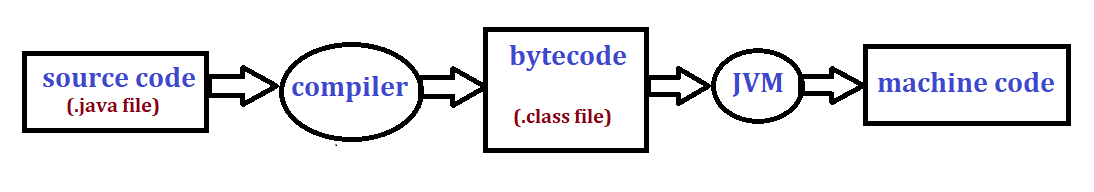
**Output :**

Test is public class

A is default class

B is default class

**Java internal working**



1. **Source code :**

* **Human readable code**

1. **Compiler :**

* **Compiler finds any syntax error in the code**
* **It converts .java file to .class file**

1. **Bytecode :**

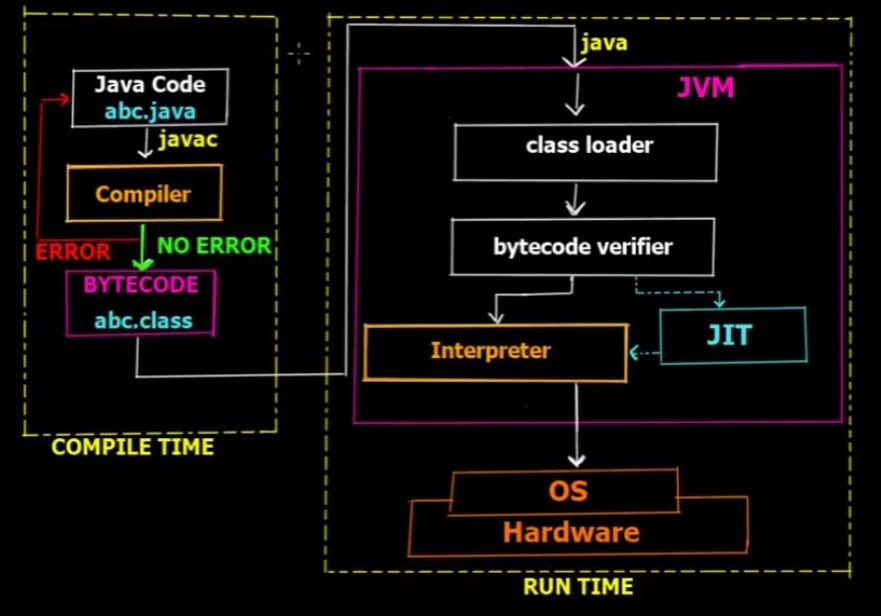
* **Not readable humans**

1. **JVM :**

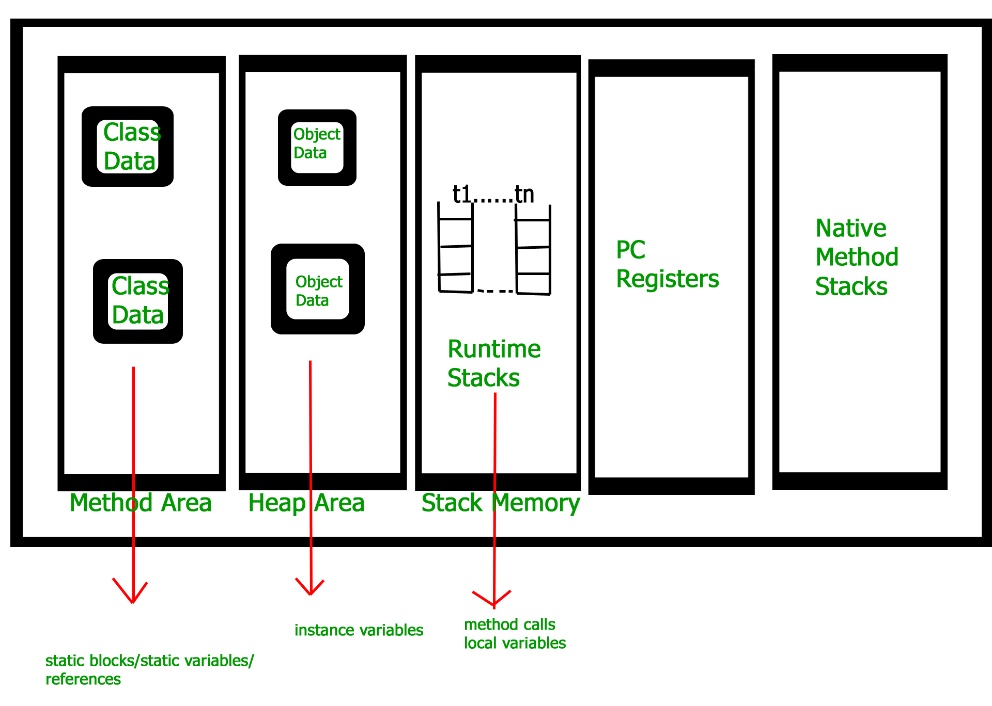
* **Java virtual machine**
* **Converts bytecode to machine code**

1. **Machine code :**

* **Code readable by operating systems**



**JVM memory**



1. **method area :**

* **cretaed when jvm started**
* **stores static data and variables**

1. **Heap memory :**

* **cretaed when jvm started**
* **stores objects, instance variables, and arrays**
* **memory free done automatically by garbage collecor**

1. **Stack memory :**

* **created when new thread started**
* **stores local variables, and currrent methods**
* **It has LIFO structure**
* **Memory free done automatically**

1. **PC registers :**

* **Holds address of new executing instruction**

1. **Native method stacks :**

* **Methods of other languages like c, c++ stored here**

**Garbage collection**

* **Garbage collector removes unwanted objects from heap memory**
* **It is done automatically by jvm**
* **When jvm started it creates 3 threads as below,**

1. **Main thread**
2. **Thread scheduler**
3. **Garbage collector**

**Question :**

1. **How can be object unreferenced in java ?**

* **There are 3 ways to do that,**

**a) By nulling te reference**

**e.g. s1=null;**

**making object null**

**b) By assigning a reference to another**

**e.g. s2=s3;**

**s2 object becomes unreferenced and pointing to object s3**

**c) By anonymous object**

**e.g. new student(4,”vithal”);**

**created object but not stored its reference value**

**Example :**

**Test.java**

package com.test.example;

public class Test

{

public static void main(String args[])

{

Student s1 = new Student(1, "ram");

Student s2 = new Student(2, "krishna");

Student s3 = new Student(3, "hari");

s1 = null; // making object null

s2 = s3; // object s2 becomes unreferenced and pointing to s3 object

new Student(4, "vithal"); // created object but not stored its refrence value

}

}

**Student.java**

**package** com.test.example;

**public** **class** Student

{

**private** **int** rollno;

**private** String name;

Student(**int** r, String n)

{

**this**.rollno=r;

**this**.name=n;

}

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

{

System.***out***.println("Roll no : "+rollno+ ", name : "+name);

}

}

**Output :**

**Finalize method**

* **Garbage collectors gc() method first internally calls finalize() method before garbaging object**
* **Finalize method closes all connections of objects so that it will not affect further**
* **Real world example :**

**After passing exam before giving books to other people or raddiwala**

**We first check that book if it contains any of our secret data like hall ticket or id card in it.**

* **So basicallly we are closing connection before removing that object**

**Syntax :**

**Student.java**

**System.gc(); //garbage collector method**

**protected void finalize() //finalize method**

**{**

**//closing open connections**

**con.close(); //closing database connection**

**file.close(); //closing file connection**

**}**

**Question :**

1. **Where finalize() method is located ?**

* **In Object.class**

1. **Is finalize() method is deprecated ?**

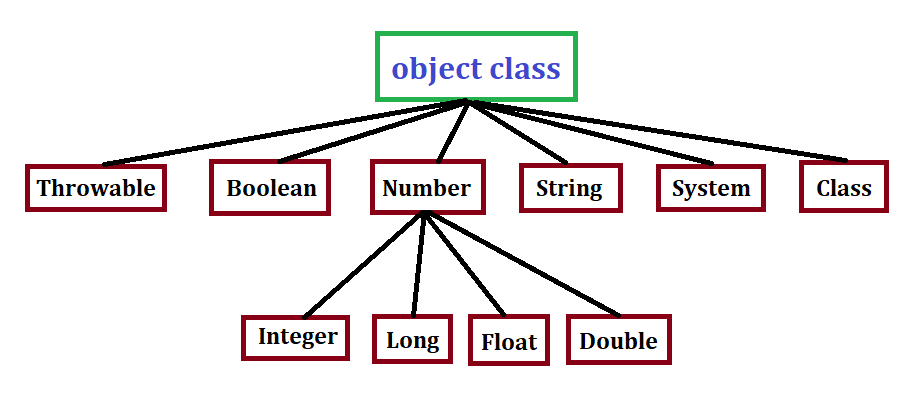
* **Yes**

**Object class**

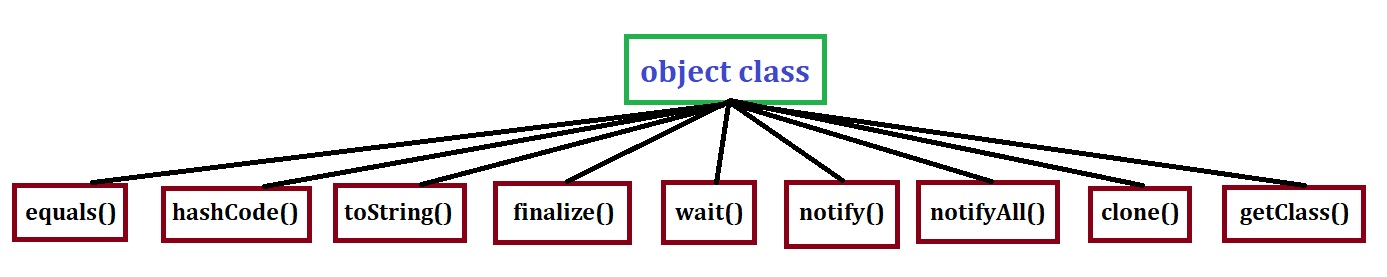
**- object class is the topmost parent class in java**

**- all other classes extends directly or indirectly object class only**

**Classes under Object.class :**



**methods of Object.class :**



**hashcode() method from Object.class**

* **hashcode() method generates unique number for each object for storing in memory**

**Example 1 :**

**Test.java**

package com.test.example;

public class Test

{

public static void main(String[] args)

{

Student s1 = new Student(101, "ram");

System.*out*.println("hashcode for s1 is : " + s1.hashCode());

}

}

**Student.java**

package com.test.example;

public class Student

{

private int rollNum;

private String name;

public Student(int rollNum, String name)

{

this.rollNum = rollNum;

this.name = name;

}

}

**Output :**

hashcode for s1 is : **1651191114**

**Example 2 :**

**Test.java**

package com.test.example;

public class Test

{

public static void main(String[] args)

{

Student s1 = new Student(101, "ram");

**Student s2 = new Student(101, "ram");**

System.*out*.println("hashcode for s1 is : " + s1.hashCode());

System.*out*.println("hashcode for s2 is : " + s2.hashCode());

}

}

**Student.java**

package com.test.example;

public class Student

{

private int rollNum;

private String name;

public Student(int rollNum, String name)

{

this.rollNum = rollNum;

this.name = name;

}

}

**Output :**

hashcode for s1 is : **1651191114**

hashcode for s2 is : **1579572132**

**Example 3 :**

**Test.java**

package com.test.example;

public class Test

{

public static void main(String[] args)

{

Student s1 = new Student(101, "ram");

Student s2 = new Student(**102**, "ram");

System.*out*.println("hashcode for s1 is : " + s1.hashCode());

System.*out*.println("hashcode for s2 is : " + s2.hashCode());

}

}

**Student.java**

package com.test.example;

public class Student

{

private int rollNum;

private String name;

public Student(int rollNum, String name)

{

this.rollNum = rollNum;

this.name = name;

}

}

**Output :**

hashcode for s1 is : **1651191114**

hashcode for s2 is : **1579572132**

**Example 4 :**

**Test.java**

package com.test.example;

public class Test

{

public static void main(String[] args)

{

Student s1 = new Student(101, "ram");

Student s2 = new Student(102, "**krishna**");

System.*out*.println("hashcode for s1 is : " + s1.hashCode());

System.*out*.println("hashcode for s2 is : " + s2.hashCode());

}

}

**Student.java**

package com.test.example;

public class Student

{

private int rollNum;

private String name;

public Student(int rollNum, String name)

{

this.rollNum = rollNum;

this.name = name;

}

}

**Output :**

hashcode for s1 is : **1651191114**

hashcode for s2 is : **1579572132**

**Example 5 : (overriding hashCode() method)**

**Test.java**

package com.test.example;

public class Test

{

public static void main(String[] args)

{

Student s1 = new Student(101, "ram");

Student s2 = new Student(102, "**krishna**");

System.*out*.println("hashcode for s1 is : " + s1.hashCode());

System.*out*.println("hashcode for s2 is : " + s2.hashCode());

}

}

**Student.java**

package com.test.example;

public class Student

{

private int rollNum;

private String name;

public Student(int rollNum, String name)

{

this.rollNum = rollNum;

this.name = name;

}

@Override

public int hashCode()

{

return 123;

}

}

**Output :**

hashcode for s1 is : **123**

hashcode for s2 is : **123**

**Example 6 : (overriding hashCode() method)**

**Test.java**

package com.test.example;

public class Test

{

public static void main(String[] args)

{

Student s1 = new Student(101, "ram");

Student s2 = new Student(102, "**krishna**");

System.*out*.println("hashcode for s1 is : " + s1.hashCode());

System.*out*.println("hashcode for s2 is : " + s2.hashCode());

}

}

**Student.java**

package com.test.example;

public class Student

{

private int rollNum;

private String name;

public Student(int rollNum, String name)

{

this.rollNum = rollNum;

this.name = name;

}

@Override

public int hashCode()

{

return rollNum;

}

}

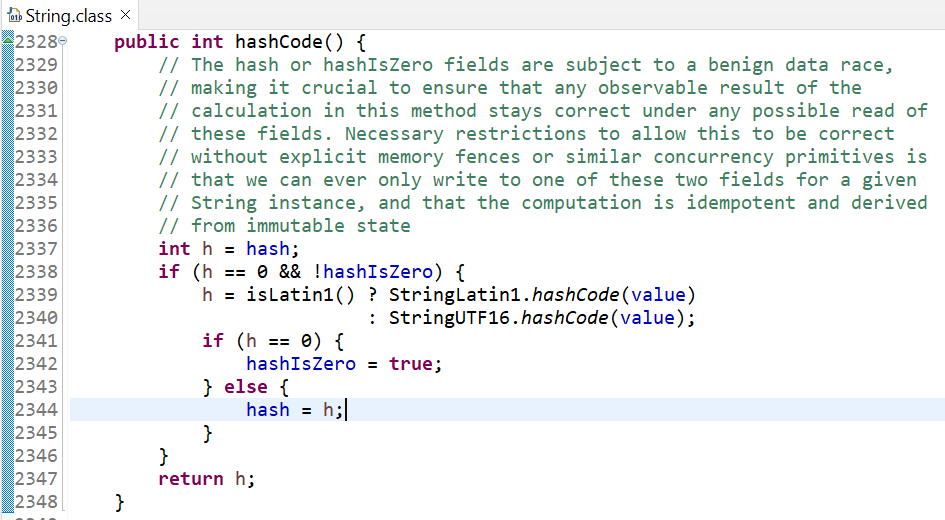
**Output :**

hashcode for s1 is : **101**

hashcode for s2 is : **102**

**hashcode() method from String.class**

* **String.class overrided hashCode() method from Object.class only**



**Example 1 :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String str1 = "bruce";

String str2 = "jackie";

System.***out***.println("hashcode for str1 is : " + str1.hashCode());

System.***out***.println("hashcode for str2 is : " + str2.hashCode());

}

}

**Output :**

hashcode for str1 is : **94016839**

hashcode for str2 is : **-1167640261**

**Example 2 : (Strings which retruns same hashcode numbers)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String str1 = "FB";

String str2 = "Ea";

System.***out***.println("hashcode for str1 is : " + str1.hashCode());

System.***out***.println("hashcode for str2 is : " + str2.hashCode());

}

}

**Output :**

hashcode for str1 is : **2236**

hashcode for str2 is : **2236**

**Note : same hashcode generation is done in String.class only otherwise it creates unique hashcode numbers**

**Example 1 : (== operator)**

**equals() method from Object.class**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

Student s1 = **new** Student(101, "ram");

Student s2 = **new** Student(101, "ram");

System.***out***.println(s1 == s2);

}

}

**Student.java**

**package** com.test.example;

**public** **class** Student

{

**private** **int** rollNum;

**private** String name;

**public** Student(**int** rollNum, String name)

{

**this**.rollNum = rollNum;

**this**.name = name;

}

}

**Output :**

**false**

**Explaination :**

* **Both s1 and s2 belongs to different objects in heap memory**



**Example 2 : (== operator)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

Student s1 = **new** Student(101, "ram");

Student s2 = s1;

System.***out***.println(s1 == s2);

}

}

**Student.java**

**package** com.test.example;

**public** **class** Student

{

**private** **int** rollNum;

**private** String name;

**public** Student(**int** rollNum, String name)

{

**this**.rollNum = rollNum;

**this**.name = name;

}

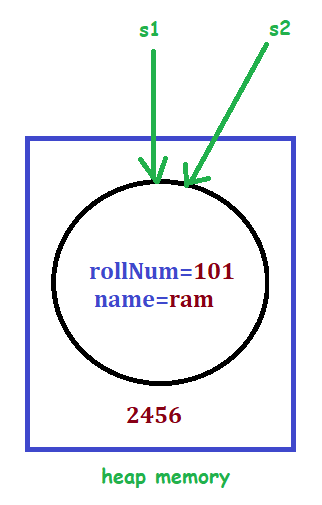
}

**Output :**

**true**

**Explaination :**

* **Both s1 and s2 belongs to same objects in heap memory**



**Example 3 : (equals method)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

Student s1 = **new** Student(101, "ram");

Student s2 = **new** Student(101, "ram");

System.***out***.println(s1.equals(s2));

}

}

**Student.java**

**package** com.test.example;

**public** **class** Student

{

**private** **int** rollNum;

**private** String name;

**public** Student(**int** rollNum, String name)

{

**this**.rollNum = rollNum;

**this**.name = name;

}

}

**Output :**

**false**

**Example 4 : (equals method)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

Student s1 = **new** Student(101, "ram");

Student s2 = s1;

System.***out***.println(s1.equals(s2));

}

}

**Student.java**

**package** com.test.example;

**public** **class** Student

{

**private** **int** rollNum;

**private** String name;

**public** Student(**int** rollNum, String name)

{

**this**.rollNum = rollNum;

**this**.name = name;

}

}

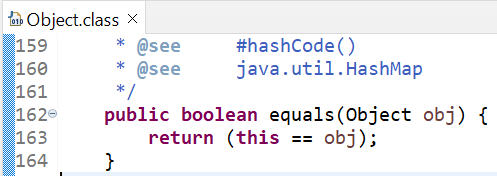
**Output :**

**true**

**Question :**

1. **Why ==operator and equals() method output is same ?**

* **Because equals() method of Object.class does == operation only**



**Example 5 : (equals method)**

**Test.java**

package com.test.example;

public class Test

{

public static void main(String[] args)

{

Student s1 = new Student(101, "ram");

Student s2 = new Student(101, "ram");

Student s3 = new Student(101, "krishna");

Student s4 = new Student(102, "ram");

System.*out*.println("s1.equals(s2) : "+ s1.equals(s2));

System.*out*.println("s1.equals(s3) : "+ s1.equals(s3));

System.*out*.println("s1.equals(s4) : "+ s1.equals(s4));

}

}

**Student.java**

**package** com.test.example;

**public** **class** Student

{

**private** **int** rollNum;

**private** String name;

**public** Student(**int** rollNum, String name)

{

**this**.rollNum = rollNum;

**this**.name = name;

}

}

**Output :**

s1.equals(s2) : false

s1.equals(s3) : false

s1.equals(s4) : false

**Example 6 : (overriding equals method)**

**Test.java**

package com.test.example;

public class Test

{

public static void main(String[] args)

{

Student s1 = new Student(101, "ram");

Student s2 = new Student(101, "ram");

Student s3 = new Student(101, "krishna");

Student s4 = new Student(102, "ram");

System.*out*.println("s1.equals(s2) : "+ s1.equals(s2));

System.*out*.println("s1.equals(s3) : "+ s1.equals(s3));

System.*out*.println("s1.equals(s4) : "+ s1.equals(s4));

}

}

**Student.java**

**package** com.test.example;

**public** **class** Student

{

**private** **int** rollNum;

**private** String name;

**public** Student(**int** rollNum, String name)

{

**this**.rollNum = rollNum;

**this**.name = name;

}

@Override

**public** **boolean** equals(Object obj)

{

**return** **true**;

}

}

**Output :**

s1.equals(s2) : true

s1.equals(s3) : true

s1.equals(s4) : true

**Example 7 : (overriding equals method)**

**Test.java**

package com.test.example;

public class Test

{

public static void main(String[] args)

{

Student s1 = new Student(101, "ram");

Student s2 = new Student(101, "ram");

Student s3 = new Student(101, "krishna");

Student s4 = new Student(102, "ram");

System.*out*.println("s1.equals(s2) : "+ s1.equals(s2));

System.*out*.println("s1.equals(s3) : "+ s1.equals(s3));

System.*out*.println("s1.equals(s4) : "+ s1.equals(s4));

}

}

**Student.java**

**package com.test.example;**

**public class Student**

**{**

**private int rollNum;**

**private String name;**

**public Student(int rollNum, String name)**

**{**

**this.rollNum = rollNum;**

**this.name = name;**

**}**

**@Override**

**public boolean equals(Object obj)**

**{**

**Student student = (Student) obj;**

**if (this.rollNum != student.rollNum)**

**{**

**return false;**

**}**

**return true;**

**}**

**}**

**Output :**

s1.equals(s2) : true

s1.equals(s3) : true

s1.equals(s4) : false

**Example 8 : (overriding equals method)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

Student s1 = **new** Student(101, **null**);

Student s2 = **new** Student(101, "ram");

Test t= **new** Test();

System.***out***.println("s1.equals(t) : "+ s1.equals(t));

}

}

**Student.java**

**package** com.test.example;

**public** **class** Student

{

**private** **int** rollNum;

**private** String name;

**public** Student(**int** rollNum, String name)

{

**this**.rollNum = rollNum;

**this**.name = name;

}

@Override

**public** **boolean** equals(Object obj)

{

**if** (obj == **null**)

{

**return** **false**;

}

**if** (obj.getClass() != **this**.getClass())

{

**return** **false**;

}

Student student = (Student) obj;

**if** (**this**.rollNum != student.rollNum)

{

**return** **false**;

}

**if** (**this**.name == **null**)

{

**if** (student.name != **null**)

{

**return** **false**;

}

}

**if** (!**this**.name.equals(student.name)) // equals() of String.class

{

**return** **false**;

}

**return** **true**;

}

}

**Output :**

s1.equals(t) : false

**- String.class overrided equals() method from Object.class only**

**equals() method from String.class**

**Example 1 : (equals method)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String str1="hitesh";

String str2="man";

System.***out***.println("str1.equals(str2) : "+ str1.equals(str2));

}

}

**Output :**

s1.equals(t) : false

**Example 2 : (equals method)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String str1="hitesh";

String str2="hitesh";

System.***out***.println("str1.equals(str2) : "+ str1.equals(str2));

}

}

**Output :**

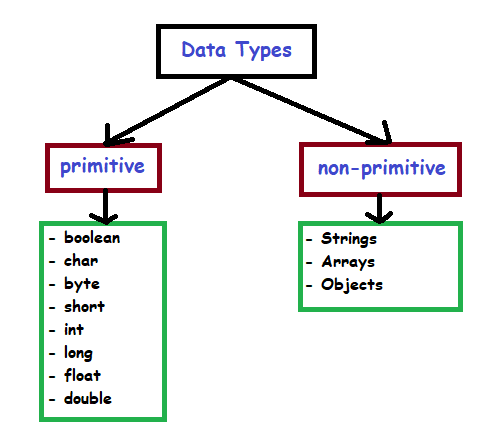
s1.equals(t) : true

**Note :**

**- equals() method of String.class works differently than as in Object.class**

**- Object.class internally checks reference is same or not i.e. (s1=s2)**

**- String class checks content is same or not i.e. (hitesh=hitesh)**



**Data types**

**Primitive data types**

**example**

**Data types**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **range** | **size** |  |
| **boolean** | true/false |  | boolean b = true; |
| **char** | 0 to 65535 | 16 bits | Char c = ‘A’ |
| **byte** | -128 to 127 | 8 bits | byte a = 100; |
| **short** | -32,768 to 32767 | 16 bits | short s = 1000; |
| **int** | -2,147,483,648 to 2,147,483, 647 | 32 bits | int i = 1000; |
| **long** | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 | 64 bits | long l = 100L; |
| **float** | -3,4e38 to 3.4e38 | 32 bits | float f = 234.5f; |
| **double** | -1.7e308 to 1.7e308 | 32 bits | double d = 234.5; |

**Non-Primitive data types**

|  |  |
| --- | --- |
| **Data types** | **range** |
| **String** | String str= “hello”; |
| **array** | DataType[] arr = new DataType[arraySize]; |
| **class** | Public class Test { } |
| **interface** | Public interface Test { } |

**variables**

**Example :**

**int a = 10; //a is variable here**

**Local variable**

**- Declared inside methods, constuctor, blocks**

**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

Test v = **new** Test();

v.display();

}

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

{

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

**int** b = 20; // local variable

**int** c = a + b;

System.***out***.println("sum is : " + c);

}

}

**Output :**

sum is : 30

**Instance variable**

**- Declared inside class but outside methods, constuctor, blocks**

**Example :**

**Test.java**

**package com.test.example;**

**public class Test**

**{**

**public String name; // instance variable**

**private double salary; // instance variable**

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

**{**

**Test empOne = new Test("Hitesh");**

**empOne.setSalary(1000);**

**empOne.printEmp();**

**}**

**public Test (String empName)**

**{**

**name = empName;**

**}**

**public void setSalary(double empSal)**

**{**

**salary = empSal;**

**}**

**public void printEmp()**

**{**

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

**System.*out*.println("salary :" + salary);**

**}**

**}**

**Output :**

name : Hitesh

salary :1000.0

**Static / class variable**

**- Declared inside class but outside methods, constuctor, blocks**

**- uses static keyword to declare**

**Example :**

**Test.java**

**package com.test.example;**

**public class Test**

**{**

**private static double *salary*; // static variable**

**public static final String *DEPARTMENT* = "IT";// static variable**

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

**{**

***salary* = 1000;**

**System.*out*.println(*DEPARTMENT* + "average salary:" + *salary*);**

**}**

**}**

**Output :**

ITaverage salary:1000.0

**Question :**

1. **Instance vs static variable ?**

* **Static variables, methods can be called using clasname also.**
* **So no need to create an object.**

**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

**int** a;

**static** **int** *b* = 30;

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

{

Test obj = **new** Test();

System.***out***.println("instance variable using object : " + obj.a);

System.***out***.println("static variable using object : " + obj.*b*);

System.***out***.println("static variable using classname : " + Test.*b*);

}

}

**Output :**

Value of instance variable using object : 0

Value of static variable using object : 30

Value of static variable using classname : 30

* **Static variables initializes values once only.**
* **But instance variables initializes value again and again.**

**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

**int** a = 0;

**static** **int** *b* = 0;

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

{

Test obj1 = **new** Test();

obj1.a++;

System.***out***.println("instance variable after increment : " + obj1.a);

Test obj2 = **new** Test();

obj2.a++;

System.***out***.println("instance variable after increment : " + obj2.a);

obj1.*b*++;

System.***out***.println("static variable after increment : " + obj1.*b*);

obj2.*b*++;

System.***out***.println("static variable after increment : " + obj2.*b*);

}

}

**Output :**

instance variable after increment : 1

instance variable after increment : 1

static variable after increment : 1

static variable after increment : 2

**modifier**

**Access modifiers**

**Description**

**Access modifier**

|  |  |
| --- | --- |
| **Access modifiers** | **Description** |
| **Default** | * **No modifier name** * **Accessible within same package** |
| **Public** | * **Accessible to all** |
| **Private** | * **Accessible to class only** |
| **Protected** | * **Accessible to subclass even in different packages**   **(only if it is inherited)** |

**Subclass**

**diff package**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Access modifiers** | **class** | **package** | **Subclass**  **Same package** |  | **world** |
|  |  |  |  |  |  |
| **Default** | **yes** | **yes** | **yes** | **no** | **no** |
| **public** | **yes** | **yes** | **yes** | **yes** | **yes** |
| **private** | **yes** | **no** | **no** | **no** | **no** |
| **protected** | **yes** | **yes** | **yes** | **yes** | **no** |

**Note :**

* **Class and interface can not be private or protected.**

**They are public or default only.**

* **Methods, variables, constructors can be private, protected, public, default.**

**Non-Access modifiers**

1. **Static : keyword of class level**
2. **final : keyword to avoid access**
3. **abstract : keyword used to achieve abstraction**
4. **synchronized : used in multithreading for concurrency output**
5. **volatile : modifies variable value of threads**
6. **transient : used in serialization to avoid serialization of specific variables**

**Call private methods**

**Example : (using public method)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

Sample t = **new** Sample();

t.show();

}

}

**Sample.java**

**package** com.test.example;

**public** **class** Sample

{

**public** **void** show()

{

display();

}

**private** **void** display()

{

System.***out***.println("private method of Sample class");

}

}

**Output :**

private method of Sample class

**Example : (using reflection API)**

**Test.java**

**package** com.test.example;

**import** java.lang.reflect.Method;

**public** **class** Test

{

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

{

Class cls = Class.*forName*("com.test.example.Sample");

Sample sample = (Sample) cls.newInstance();

Method mt = cls.getDeclaredMethod("display", **null**);

mt.setAccessible(**true**);

mt.invoke(sample, **null**);

}

}

**Sample.java**

**package** com.test.example;

**public** **class** Sample

{

**private** **void** display()

{

System.***out***.println("private method of Sample class");

}

}

**Output :**

private method of Sample class

**while loop**

**loops**

**Syntax :**

**while(condition)**

**{**

**//code**

**}**

**Example :**

**Test.java**

package com.test.example;

public class Test

{

public static void main(String[] args)

{

int i = 4;

while (i <= 5)

{

System.*out*.println(i);

i++;

}

}

}

**Output :**

4

5

**Example : (while true)**

**Test.java**

package com.test.example;

public class Test

{

public static void main(String[] args)

{

while (true)

{

System.*out*.println("vithal panduranga");

}

}

}

**Output :**

vithal panduranga

vithal panduranga

.

.

.

.

.

**- same as while loop but executes at least once before testing condition in while loop**

**do while loop**

**Syntax :**

**do**

**{**

**//code**

**}**

**while(condition)**

**Example :**

**Test.java**

package com.test.example;

public class Test

{

public static void main(String[] args)

{

int i = 2;

do

{

System.*out*.println(i);

i++;

} while (i >= 5);

}

}

**Output :**

2 //executed once even though condition not matches

**Example : (while true)**

**Test.java**

package com.test.example;

public class Test

{

public static void main(String[] args)

{

do

{

System.*out*.println("vithal pandurang");

} while (true);

}

}

**Output :**

vithal panduranga

vithal panduranga

.

.

.

.

.

**Syntax :**

**for(initialization; condition; updateValue)**

**{**

**//code**

**}**

**for loop**

**Example :**

**Test.java**

public class Test

{

public static void main(String[] args)

{

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

{

System.*out*.println(i);

}

}

}

**Output :**

1

2

3

**for each loop**

**Syntax :**

**for(declaration; listObjects)**

**{**

**//code**

**}**

**Example :**

**Test.java**

public class Test

{

public static void main(String[] args)

{

int[] input = { 125, 132, 95};

for (int num : input)

{

System.*out*.println(num);

}

}

}

**Output :**

125

132

95

**loop control**

**break**

**- helps in breaking the loop**

**Example :**

**Test.java**

package com.test.example;

public class Test

{

public static void main(String[] args)

{

int[] input = { 10, 20, 30, 40, 50 };

for (int x : input)

{

if (x == 30)

{

break;

}

System.*out*.println(x);

}

}

}

**Output :**

10

20

**- helps in breaking the statement and continues loop**

**continue**

**Example :**

**Test.java**

package com.test.example;

public class Test

{

public static void main(String[] args)

{

int[] input = { 10, 20, 30, 40, 50 };

for (int x : input)

{

if (x == 30)

{

continue;

}

System.*out*.println(x);

}

}

}

**Output :**

10

20

40

50

**lables**

**- If we have nested loops and we apply break/continue it will work on inner loop**

**- So if we want to apply break/continue on outer loops we can use label**

**- We can specify which loop to break/continue using label**

**- We can specify name of loop using labels which we want to break or continue**

**Syntax :**

**LabelName : any loop**

**{**

**inner loops/statements**

**{**

**break labelName;**

**Or**

**continue labelName;**

**}**

**}**

**Example :**

**Break label**

**Test.java**

**package com.test.example;**

**public class Test**

**{**

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

**{**

**Task: for (int i = 1; i <= 4; i++)**

**{**

**System.*out*.println("-------- Outer loop value : " + i);**

**for (int j = 1; j < 10; j++)**

**{**

**System.*out*.println("Inner loop value : " + j);**

**if (j == 2)**

**{**

**System.*out*.println("breaking outer loop");**

**break Task;**

**}**

**}**

**}**

**}**

**}**

**Output :**

**-------- Outer loop value : 1**

**Inner loop value : 1**

**Inner loop value : 2**

**breaking outer loop**

**Example :**

**continue label**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

Task: **for** (**int** i = 1; i <= 4; i++)

{

System.***out***.println("-------- Outer loop value : " + i);

**for** (**int** j = 1; j < 10; j++)

{

System.***out***.println("Inner loop value : " + j);

**if** (j == 2)

{

System.***out***.println("breaking outer loop");

**continue** Task;

}

}

}

}

}

**Output :**

**-------- Outer loop value : 1**

**Inner loop value : 1**

**Inner loop value : 2**

**breaking outer loop**

**-------- Outer loop value : 2**

**Inner loop value : 1**

**Inner loop value : 2**

**breaking outer loop**

**-------- Outer loop value : 3**

**Inner loop value : 1**

**Inner loop value : 2**

**breaking outer loop**

**-------- Outer loop value : 4**

**Inner loop value : 1**

**Inner loop value : 2**

**breaking outer loop**

**statements**

**if**

**Syntax :**

**if(condition)**

**{**

**//code**

**}**

**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

**int** age = 20;

**if** (age > 18)

{

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

}

}

}

**Output :**

Age is greater than 18

**if else**

**Syntax :**

**if(condition)**

**{**

**//code**

**}**

**else**

**{**

**//code**

**}**

**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

**int** age = 16;

**if** (age >= 18)

{

System.***out***.print("inside if loop");

} **else**

{

System.***out***.print("inside else loop");

}

}

}

**Output :**

inside else loop

**nested if else**

**Syntax :**

**if(condition)**

**{**

**//code**

**if(condition)**

**{**

**//code**

**}**

**else**

**{**

**//code**

**}**

**}**

**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

**int** age = 15;

**if** (age <= 18)

{

System.***out***.println("inside if loop");

**if** (age == 15)

{

System.***out***.println("age is 15");

}

} **else**

{

System.***out***.print("inside else loop");

}

}

}

**Output :**

inside if loop

age is 15

**ternary / conditional operator**

**Syntax :**

**exp1 ? exp2 : exp3;**

**if exp1 is true,then exp2 is the answer**

**if exp1 is false, then exp3 is the answer**

**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

**int** n1 = 5;

**int** n2 = 10;

**int** maxNum = (n1 > n2) ? n1 : n2;

System.***out***.println("Maximum is = " + maxNum);

}

}

**Output :**

Maximum is = 10

**Switch case**

**Syntax :**

**Switch**(**CaseValue**)

{

**Case value** : //statements

**break;**

**Case value** : //statements

**break;**

**default** : //statements  **//optional**

}

**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

**char** grade = 'D';

**switch** (grade)

{

**case** 'A':

System.***out***.println("Excellent!");

**break**;

**case** 'B':

**case** 'C':

System.***out***.println("Well done");

**break**;

**case** 'D':

System.***out***.println("You passed");

**case** 'F':

System.***out***.println("Better try again");

**break**;

**default**:

System.***out***.println("Invalid grade");

}

}

}

**Output :**

You passed **//as D has no break statement it goes to F**

Better try again

**Increment / decrement operators**

**Example : (post-increment)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

**int** i = 1;

i++;

System.***out***.println(" i : " + i);

}

}

**Output :**

i : 2

**Example : (pre-increment)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

**int** i = 1;

++i;

System.***out***.println(" i : " + i);

}

}

**Output :**

i : 2

**Example : (post-decrement)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

**int** i = 1;

i--;

System.***out***.println(" i : " + i);

}

}

**Output :**

i : 0

**Example : (pre-decrement)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

**int** i = 1;

--i;

System.***out***.println(" i : " + i);

}

}

**Output :**

i : 0

**Enumeration**

**- Enumeration created using enum keyword**

**- enum is a data type**

**- It contains fixed set of constants**

**- E.g. days {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY}**

**directions {NORTH, SOUTH, EAST, WEST}**

**colors {RED, YELLOW, BLUE, GREEN, WHITE, BLACK}**

**- According to java, these constants must be written in capital letters**

**- These constants are static and final implicitly (i.e. already static and final) So we cant change their value**

**- It can have constructors, methods, and instance variables**

**Example :**

**Test.java**

**package com.test.example;**

**public class Test**

**{**

**public enum Days**

**{**

***SUNDAY*, *MONDAY*, *TUESDAY*, *WEDNESDAY*, *THURSDAY*, *FRIDAY*, *SATURDAY***

**}**

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

**{**

**for (Days s : Days.*values*())**

**{**

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

**}**

**}**

**}**

**Output :**

SUNDAY

MONDAY

TUESDAY

WEDNESDAY

THURSDAY

FRIDAY

SATURDAY



**STRING**

**String**

**- String objects are stored in Heap memory**

**- Because String is a class and java treated String as an objects**

**- We can create String objects in two ways,**

**String literals**

**- e.g. String str = “Hitesh”;**

**- String literals are stored in String constant pool inside heap memory**

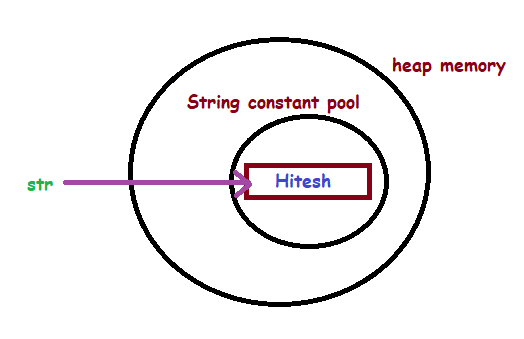
**- String constant pool is a cache stored inside heap memory**

**- till java 1.6 String constant pool was in method area of jvm (PEMGEN SPACE) but its size was fix there so memory issue was there**

**- from java 1.7 its stored in heap area of jvm so size issue sorted**

**- When we create new object its value stored in pool**

**- only 1 object is created inside String constant pool and no object creation inside heap**



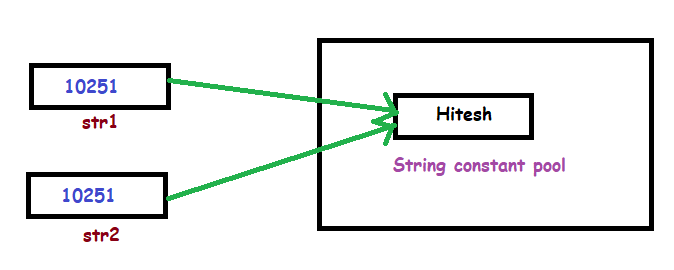
**If same value is present for another object then it refers to previous value in pool**

**i.e. String str1 = “Hello”;**

**String str2 = “Hello”;**

**Here, as object str1 & str2 both having same value.**

**Both will refer same value in String constant pool**



**This helps in saving memory**

**new keyword**

**- e.g. String str = new String(“Hitesh”);**

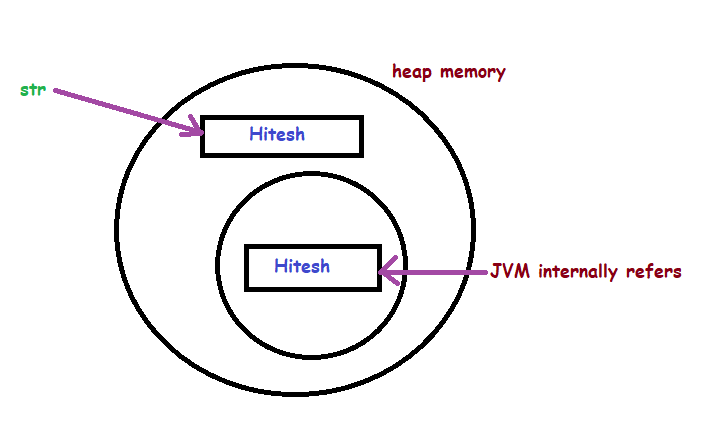
**- String with new keyword are stored normally in heap memory and object str refering to it**

**- String(“Hitesh”);**

**Will store String inside String constant pool but that object str is not refering to it.**

**Instead jvm internally creates reference for it so that it may use in future. This is extra object created**

**- here total 2 objects created in memory (i.e.one in heap and another in String constant pool)**



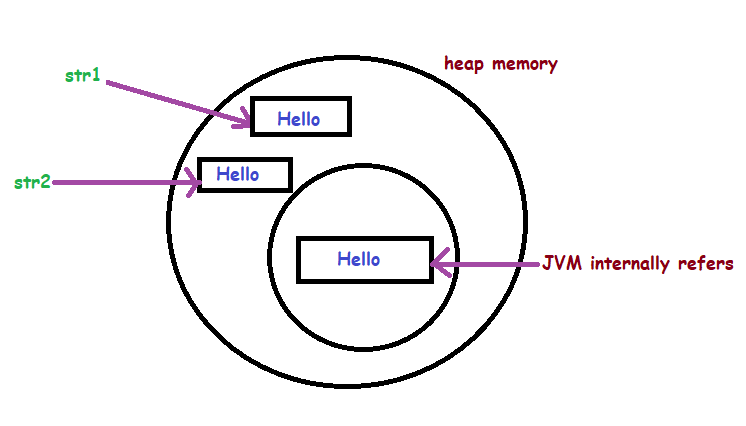
**- If same value is present for another object then it refers to previous in String constant pool but create new object in heap memory**

**- i.e. String str1 = new String(“Hello”);**

**String str2 = new String(“Hello”);**

**Here, as object str1 & str2 both having same value.**

**Both will refers to previous object in String constant pool but create new object in heap memory**



**Example :**

**Test.java**

**package com.test.example;**

**public class Test**

**{**

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

**{**

**// String created using 'new' keyword**

**String s1 = new String("TAT");**

**String s2 = new String("TAT");**

**// String created using String literal**

**String s3 = "TAT";**

**String s4 = "TAT";**

**String s5 = new String("CAT");**

**String s6 = "BAT";**

**System.*out*.println(s1);**

**System.*out*.println(s2);**

**System.*out*.println(s3);**

**System.*out*.println(s4);**

**System.*out*.println(s5);**

**System.*out*.println(s6);**

**}**

**}**

**Output :**

**TAT**

**TAT**

**TAT**

**TAT**

**CAT**

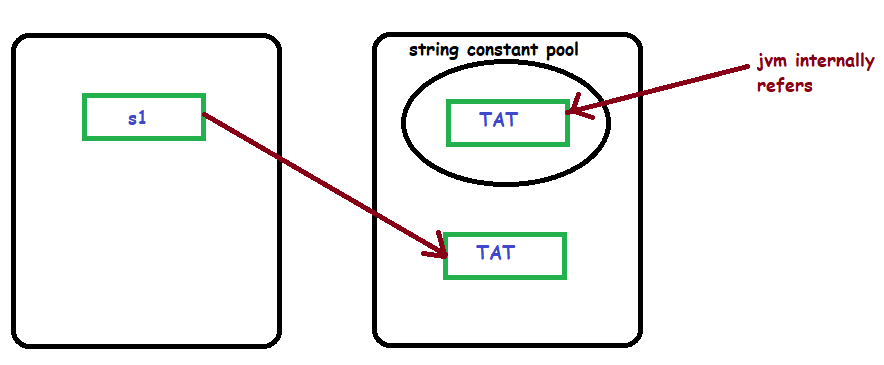
**BAT**

**Explanation :**

**1) String s1 = new String("TAT");**

**- TAT is created in heap memory and object s1 pointing to it**

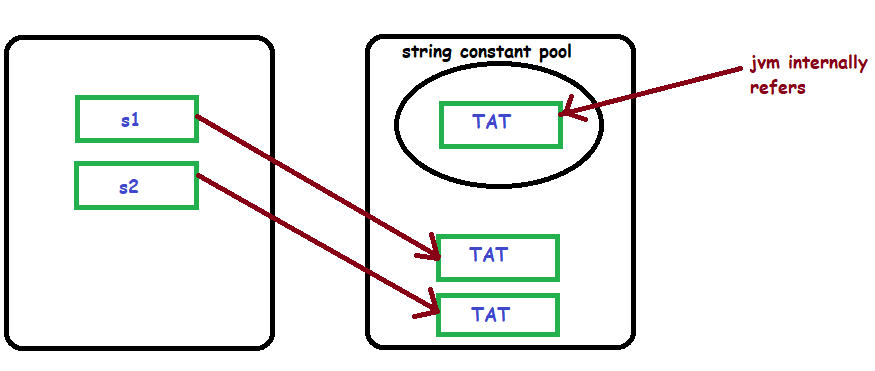
**- TAT is also created in String constant pool and JVM internally pointing to it**



**2) String s2 = new String("TAT");**

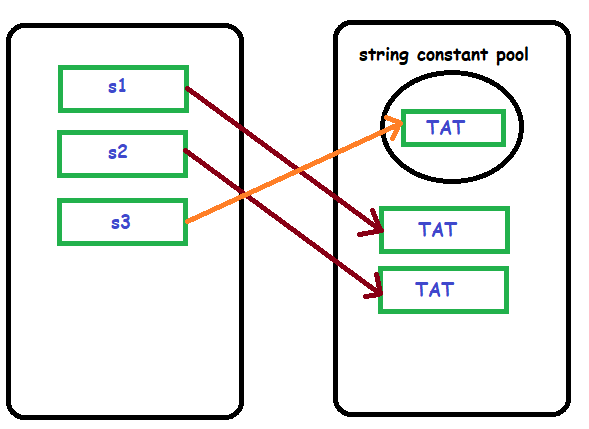
**- TAT is created in heap memory and object s2 pointing to it**

**- TAT is already present in String constant pool so another not created**



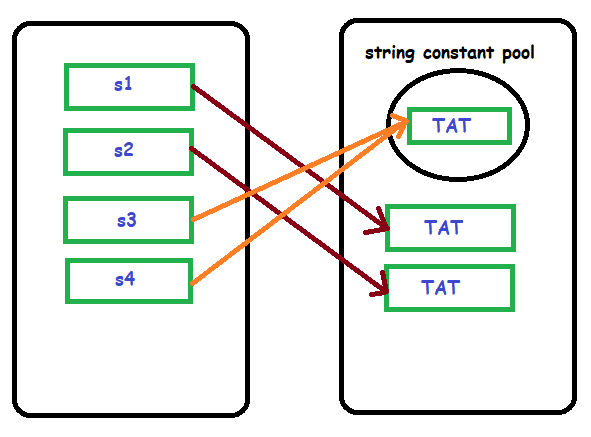
**3) String s3 = "TAT";**

**- TAT is already present in String constant pool so another not created and jvm internally pointer gets replaced with object S3**



**4) String s4 = "TAT";**

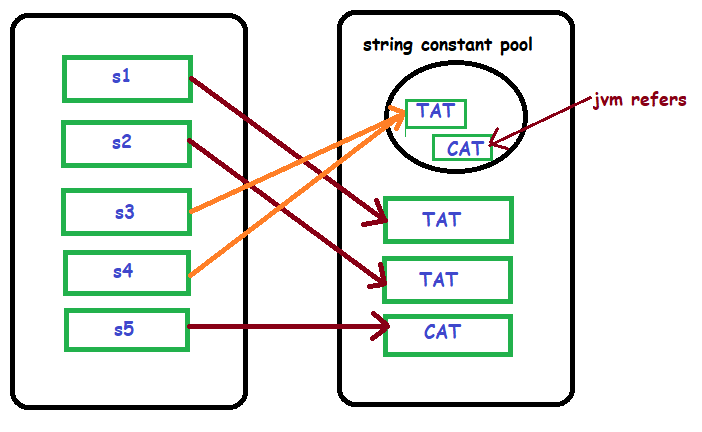
**- TAT is already present in String constant pool so another not created and object S4 is pointing to it**



**5) String s5 = new String("CAT");**

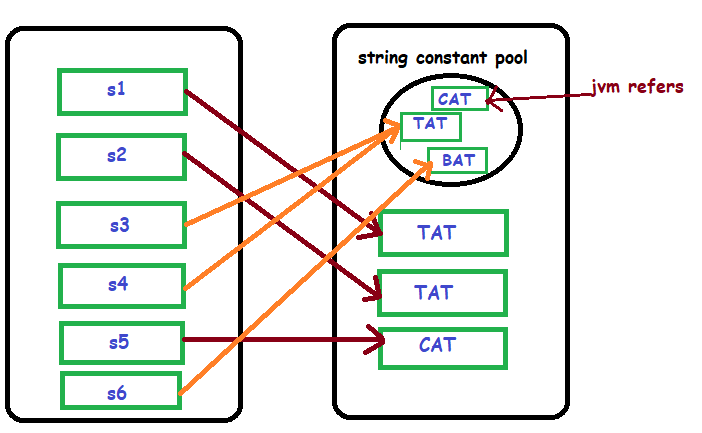
**- CAT is created in heap memory and object s5 pointing to it**

**- CAT is also created in String constant pool and JVM internally pointing to it**



**6) String s6 = "BAT";**

**- BAT is created in String constant pool and object S6 is pointing to it**



**Example : (Difference between string literal and new keyword)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

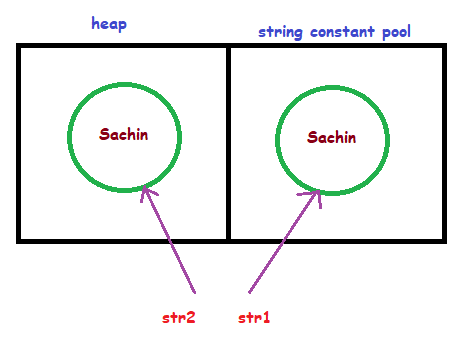
String str1 = "Sachin";

String str2 = **new** String("Sachin");

}

}

**Explanation :**



**Here**

**str1 refers to String constant pool (helps in saving memory as duplicate values can stored as single value)**

**Str2 refers to heap memory**

**immutable**

**- Immutable means which can not be changed / updated But new object is created**

**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String s = "Sachin";

s.concat(" Tendulkar");

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

}

}

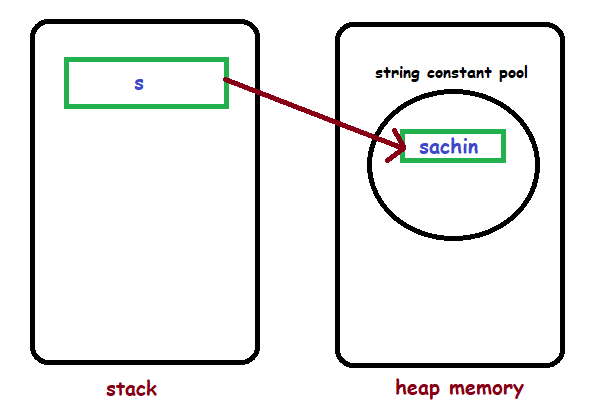
**Output :**

Sachin

**Explanation :**

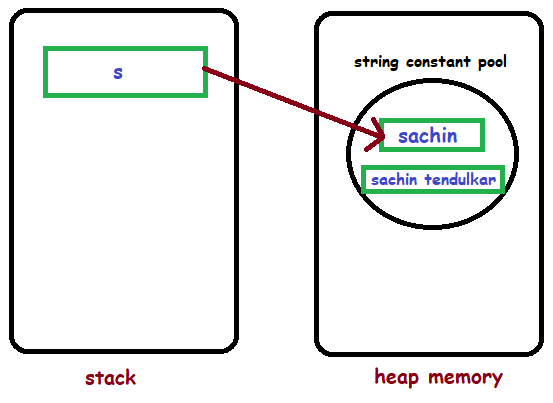
1. **String s = "Sachin";**

* **Sachin is created in string constant pool**
* **Object s is refering to it**



1. **s.concat(" Tendulkar");**

* **Sachin Tendulkar is created separately in string constant pool**
* **But object s is stiil refering to Sachin only**



**- As object s is still refering to previous value sachin so it is immutable**

**- It means two objects created but object s refers to Sachin only**

**Example : (solution to immutable)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String s = "Sachin";

s = s.concat(" Tendulkar");

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

}

}

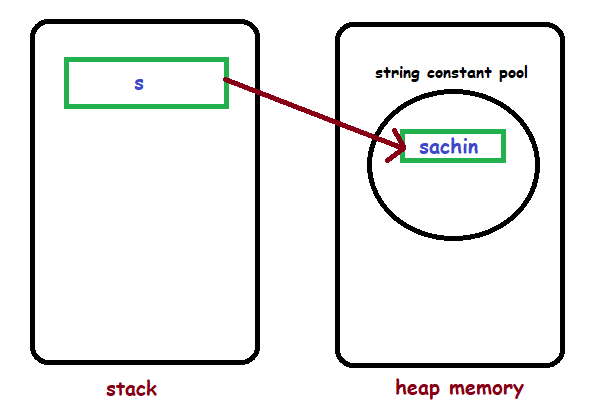
**Output :**

Sachin Tendulkar

**Explanation :**

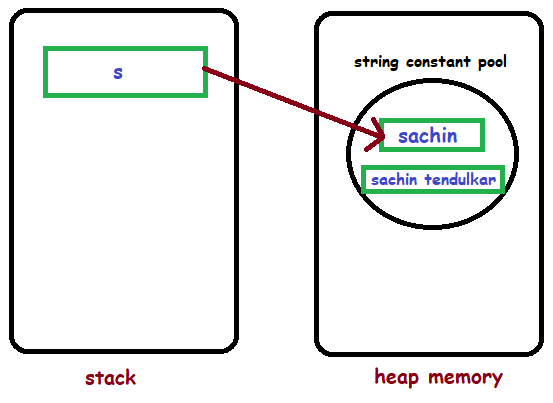
1. **String s = "Sachin";**

* **Sachin is created in string constant pool**
* **Object s is refering to it**



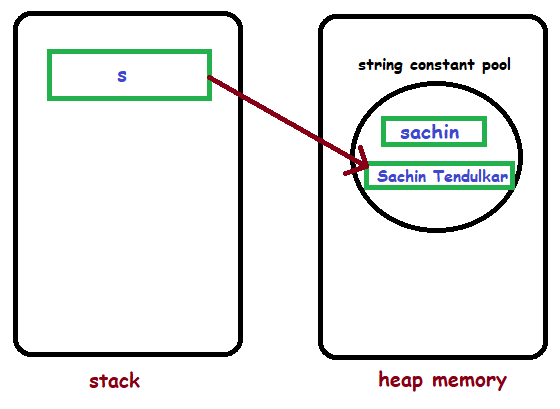
1. **s.concat(" Tendulkar");**

* **Sachin Tendulkar is created separately in string constant pool**
* **But object s is stiil refering to Sachin only**



1. **s=s.concat(" Tendulkar");**

**- object s is refering to Sachin Tendulkar now**



**Question :**

**1) Why String is immutable ?**

**➔**

We can understand its reason by below example,

**Test.java**

**public** **class** Test

{

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

{

String str1 = "Sachin";

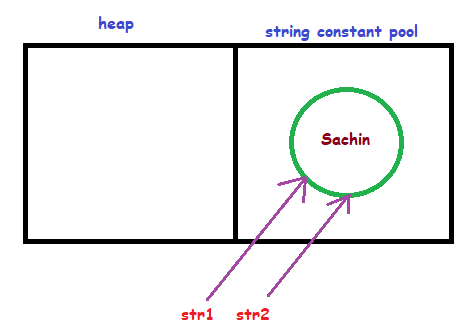
String str2 = "Sachin";

}

}

**Here, both str1 and str2 refers to same object in String constant pool**

**So memory is saved**



Now we willl try to modify string value in below example,

**Test.java**

**public** **class** Test

{

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

{

String str1 = "Sachin";

String str2 = "Sachin";

str2.concat("Dhoni");

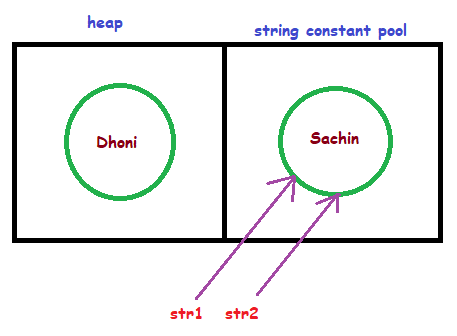
}

}

**Here, both str1 and str2 refers to same object in String constant pool**

**When str2 tries to concat its value as to Dhoni**

**Then value gets stored in memory but its not getting refferd so removed by garbage collector**



Other reasons are given as below,

**a) Security :**

**We use Strings for database connections, usernames/paswords**

**So if Strings were mutable these things ould be easily changed**

**b) synchronization & concurrency :**

**making String immutable automatically makes them thread safe**

**so solves synchronization issues**

**c) caching :**

**If two variables have same value then it creates only one object and both of them refer to that same object**

**i.e. String a =”test”;**

**String b =”test”;**

**Here, Only one object will create in SCP for both variables a & b**

**d) class loading :**

**In java String is used as argument while class loading**

**i.e. public static void main(String args[])**

**so if String was mutable then it could load wrong class**

**string methods**

**charAt()**

**- Returns character of specific position**

**Example :**

**Test.java**

**package com.test.example;**

**public class Test**

**{**

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

**{**

**String str = "Welcome to Java";**

**char ch = str.charAt(6);// returns the char value at the 6th index**

**System.*out*.println("position 6 -> " + ch);**

**System.*out*.println("--------------------------");**

**for (int i = 0; i <= str.length() - 1; i++)**

**{**

**System.*out*.println("Char at " + i + " place " + str.charAt(i));**

**}**

**}**

**}**

**Output :**

position 6 -> e

--------------------------

Char at 0 place W

Char at 1 place e

Char at 2 place l

Char at 3 place c

Char at 4 place o

Char at 5 place m

Char at 6 place e

Char at 7 place

Char at 8 place t

Char at 9 place o

Char at 10 place

Char at 11 place J

Char at 12 place a

Char at 13 place v

Char at 14 place a

**- converts String to array of characters**

**toCharArray()**

**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String s1 = "Welcome to Java";

**char**[] ch = s1.toCharArray();

**for** (**int** i = 0; i <= ch.length -1; i++)

{

System.***out***.println(ch[i]);

}

}

}

**Output :**

W

e

l

c

o

m

e

t

o

J

a

v

a

**- It concats multiple strings**

**concat()**

**Example :**

**Test.java**

package com.test.example;

public class Test

{

public static void main(String args[])

{

String str1 = "Hello";

String str2 = "Javatpoint";

String str3 = "Reader";

str1=str1.concat("world");

//Concatenating Space among strings

String str4 = str1.concat(" ").concat(str2).concat(" ").concat(str3);

System.*out*.println(str4);

//Concatenating Special Chars

String str5 = str1.concat("!!!");

System.*out*.println(str5);

String str6 = str1.concat("@").concat(str2);

System.*out*.println(str6);

}

}

**Output :**

Helloworld Javatpoint Reader

Helloworld!!!

Helloworld@Javatpoint

**- checks if string contains another specified string**

**contains()**

**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String str = "what do you know about me";

// checks string str contains below strings or not

System.***out***.println(str.contains("do you know"));

System.***out***.println(str.contains("about"));

System.***out***.println(str.contains("ABOUT")); // it is case sensitive

System.***out***.println(str.contains("hello"));

**if** (str.contains("know"))

{

System.***out***.println("This string contains know");

} **else**

{

System.***out***.println("Result not found");

}

}

}

**Output :**

true

true

false

false

This string contains know

**startsWith()**

**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String str = "java is coding";

**if** (str.startsWith("java"))

{

System.***out***.println("starts with java");

} **else**

{

System.***out***.println("Result not found");

}

**if** (str.startsWith("v", 2))

{

System.***out***.println("starts with 'v' at position 2");

} **else**

{

System.***out***.println("Result not found");

}

}

}

**Output :**

starts with java

starts with 'v' at position 2

**Example :**

**endsWith()**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String str = "java is coding";

**if** (str.endsWith("java"))

{

System.***out***.println("ends with coding");

} **else**

{

System.***out***.println("Result not found");

}

**if** (str.endsWith("g"))

{

System.***out***.println("ends with 'g'");

} **else**

{

System.***out***.println("Result not found");

}

}

}

**Output :**

Result not found

ends with 'g'

**Example :**

**equals()**

**Test.java**

**public** **class** Test

{

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

{

String s1 = "java";

String s2 = "java";

String s3 = "JAVA";

System.***out***.println(s1.equals(s2)); // True because content is same

**if** (s1.equals(s3)) // equals is case sensitive

{

System.***out***.println("both strings are equal");

} **else**

System.***out***.println("both strings are unequal");

}

}

**Output :**

true

both strings are unequal

**==**

**Example :**

**Test.java**

**public** **class** Test

{

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

{

String s1 = "java";

String s2 = "java";

String s3 = "JAVA";

System.***out***.println(s1 == s2); // True because content is same in String constant pool

**if** (s1 == s3) // == is case sensitive

{

System.***out***.println("both strings are equal");

} **else**

System.***out***.println("both strings are unequal");

}

}

**Output :**

true

both strings are unequal

**- equals() compares values**

**equals() vs ==**

**- == compares objects**

**- They will produce same result for String literals as shown in above examples. Because String literals refers same object for duplicate values But == will give false result if we create String using new keyword as shown in below example.**

**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String s1 = **new** String("HELLO");

String s2 = **new** String("HELLO");

**if** (s1 == s2)

{

System.***out***.println("result true for ==");

} **else**

{

System.***out***.println("result false for ==");

}

**if** (s1.equals(s2))

{

System.***out***.println("result true for equals");

} **else**

{

System.***out***.println("result false for equals");

}

}

}

**Output :**

result false for ==

result true for equals

**Example :**

**equalsIgnoreCase()**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String s1 = **new** String("hello");

String s2 = **new** String("HELLO");

**if** (s1.equalsIgnoreCase(s2)) // not case sensitive

{

System.***out***.println("result true");

} **else**

{

System.***out***.println("result false");

}

}

}

**Output :**

result true

**indexOf()**

**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String s1 = "this index is my example";

**int** i = s1.indexOf("example");// returns the index of "example"

System.***out***.println("position of 'example' -> " + i);

**int** j = s1.indexOf("is", 4);

// returns the index of “is” substring after 4th index

System.***out***.println("position of 'is' after 4th position -> " + j); // second "is"

}

}

**Output :**

position of 'example' -> 17

position of 'is' after 4th position -> 11

**Example :**

**lastIndexOf()**

**Test.java**

**public** **class** Test

{

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

{

String str = "This is last index of example";

**int** i = str.lastIndexOf('s');// s from "last"

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

**int** k = str.lastIndexOf('s', 5);// s from "This"

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

**int** j = str.lastIndexOf("of");

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

**int** index = str.lastIndexOf("of", 25);

// as of is at position 19 so we can mention any number > 19 here

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

index = str.lastIndexOf("of", 10);

//it will result -1 as 10 is < 19

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

}

}

**Output :**

10

3

19

19

-1

**Example :**

**isEmpty()**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String s1 = "";

**if** (s1.length() == 0 || s1.isEmpty())// Either length is zero or isEmpty is true

{

System.***out***.println("String s1 is empty");

} **else**

{

System.***out***.println("s1");

}

}

}

**Output :**

String s1 is empty

**Example :**

**join()**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String s = String.*join*("/", "www.hit.com", "email", "inbox");

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

String str = String.*join*("--yo--", "hello", "java", "world");

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

}

}

**Output :**

www.hit.com**/**email**/**inbox

hello**--yo--**java**--yo--**world

**Example :**

**replace()**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String s1 = "javatpoint is a very good website";

String str = s1.replace('a', 'e');

// replaces all occurrences of 'a' to 'e'

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

String str1 = s1.replace("is", "was");

//replaces all occurrences of "is" to "was"

System.***out***.println(str1);

}

}

**Output :**

jev**e**tpoint is e very good website

javatpoint was a very good website

**- It is similar to replace() method**

**replaceAll()**

**- But it not works with chars**

**- In addition it uses regex to replace also i.e. replaceAll(String regex,String replacement)**

**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String s1 = "javatpoint is a very good website";

String str = s1.replaceAll("a", "e");

// replaces all occurrences of 'a' to 'e'

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

String str1 = s1.replaceAll("is", "was");

//replaces all occurrences of "is" to "was"

System.***out***.println(str1);

String str2 = s1.replaceAll("\\s", "");

//regex to replace white spaces

System.***out***.println(str2);

}

}

**Output :**

j**e**v**e**tpoint is **e** very good website

javatpoint **was** a very good website

javatpointisaverygoodwebsite

**- splits String and stores in array**

**spilt()**

**Example 1 :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String str = "Hello-world-java";

String[] arr = str.split("-");

**for** (String s : arr)

{

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

}

}

}

**Output :**

Hello

world

java

**Example 2 :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String str = "Hello world java";

String[] arr = str.split(" ");

**for** (String s : arr)

{

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

}

}

}

**Output :**

Hello

world

java

**Example 3 : (using \\s+** **to not consider white spaces)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String str = "Hello world java";

String[] arr = str.split("\\s+");

**for** (String s : arr)

{

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

}

}

}

**Output :**

Hello

world

java

**Example :**

**length()**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String s1 = "javatpoint";

String s2 = "python";

System.***out***.println("length of javatpoint is: " + s1.length());

System.***out***.println("length of python is: " + s2.length());

}

}

**Output :**

length of javatpoint is: 10

length of python is: 6

**- reads String from specific position to specific postion**

**substring()**

**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String s1 = "java is coding";

String substr = s1.substring(0); // Starts with 0 and goes to end

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

String substr1 = s1.substring(2); // Starts with 1 and goes to end

System.***out***.println(substr1);

String substr2 = s1.substring(5, 10); // Starts from 5 and goes to 10

System.***out***.println(substr2);

}

}

**Output :**

java is coding

va is coding

is co

**- helps in removing leading and trailing spaces**

**trim()**

**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String s1 = " hello string ";

System.***out***.println("---" + s1 + "---");// without trim()

System.***out***.println("---" + s1.trim() + "---");// with trim()

}

}

**Output**

--- hello string ---

---hello string---

**Example :**

**toLowerCase()**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String s1 = "JAVATPOINT HELLO stRIng";

String s1lower = s1.toLowerCase(); // converts to lower case

System.***out***.println(s1lower);

}

}

**Output :**

javatpoint hello string

**Example :**

**toUpperCase()**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String s1 = "hello string";

String s1upper = s1.toUpperCase();

System.***out***.println(s1upper);

}

}

**Output :**

HELLO STRING

**StringBuffer**

**- It is mutable**

**- Methods are synchronized**

**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

StringBuffer obj = **new** StringBuffer("hello");

obj.append(" java");

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

}

}

**Output :**

hello java

**StringBuilder**

**- It is mutable**

**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

StringBuilder obj = **new** StringBuilder("hello");

obj.append(" java");

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

}

}

**Output :**

hello java

**Question :**

1. **Difference between String vs StringBuffer vs StringBuilder?**



|  |  |  |  |
| --- | --- | --- | --- |
| **Sr no.** | **String** | **StringBuffer** | **StringBuilder** |
| **1.** | Creates **immutable** objects | Creates **mutable** objects | Creates **mutable** objects |
| **2.** | **Not thread safe** | **thread safe** | **Not thread safe** |
| **3.** | **Slow** performance | **Fast** performance | More **fast** performance |
| **4.** | If **data** is **not changing** frequently then use String | If **data** is **changing** frequently then use StringBuffer **e.g.calculator app** | If **data** is **changing** frequently then use StringBuilder **e.g.calculator app** |
| **5.** | If we assign same values to multiple objects then it **requires less memory** As it uses String constant pool for storage  i.e.  String s1=”hello”;  String s2=”hello”; | If we assign same values to multiple objects then it **requires more memory** | If we assign same values to multiple objects then it **requires more memory** |



**ARRAY**

**Array**

**- superclass of Array is Object class**

**- Stores data in heap memory**

**- Stores multiple values in single variable**

**- We can only store same type of variable i.e. int only or String only**

**Declaration :**

|  |  |
| --- | --- |
| **types** | **meaning** |
| **int[] a;** | **a is array** |
| **int[] a,b;** | **a and b both are array** |
| **int []a;** | **a is array** |
| **int []a,b;** | **a and b both are array** |
| **int a[];** | **a is array** |
| **int a[],b;** | **a is array and b is variable** |
| **int[]a;** | **a is array** |

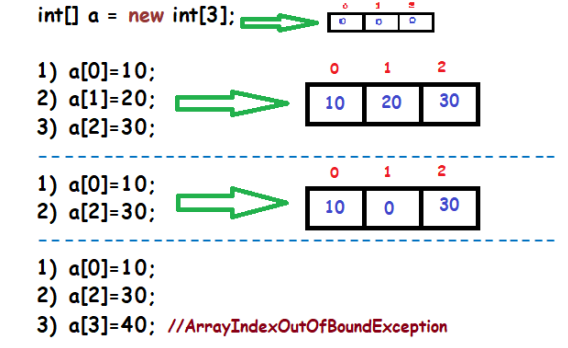
**creation :**

1. **int[] a;**

**a=new int [3];**

1. **int[] a=new int [3];**

**initialization :**



**Declaration, creation, initialization :**

1. **int[] a= {10,20,30};**
2. **int[] a= new int [ ]** **{10,20,30};**

**Syntax :**

**DataType [ ] arr = new DataType [ araySize ];**

**arr [ 5 ] = 6; //stored value 6 at location 5 of array**

**Example 1 :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

**int**[] arr = **new** **int**[10];

arr[5] = 6;

arr[2] = 86;

System.***out***.println("value at index 5 is : " + arr[5]);

**for** (**int** i : arr)

{

System.***out***.print(" " + i);

}

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

**for** (**int** i = 0; i < arr.length; i++)

{

System.***out***.print(" " + arr[i]);

}

}

}

**Output :**

value at index 5 is : **6**

0 0 86 0 0 **6** 0 0 0 0

0 0 86 0 0 **6** 0 0 0 0

**\\\\**

**Example 2 :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

String[] arr1 =

{ "RAM", "KRISHNA", **null**, "VITHAL", "", "NARSIMHA" };

**for** (String str : arr1)

{

System.***out***.println("elements : " + str);

}

}

}

**Output :**

elements : RAM

elements : KRISHNA

elements : null

elements : VITHAL

elements :

elements : NARSIMHA

**\\\\**

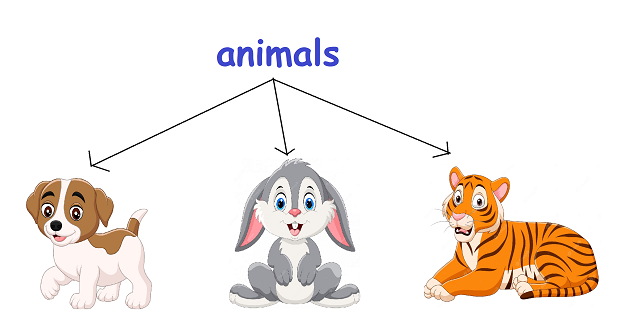


**OOPS**

**Introduction to oops**



**class**



**- In above fig, ANIMALS is a class**

**- Dog, rabbit ,tiger are its objects**

**- Class is a collection of objects i.e. Animals is collection of objects like dog, rabbit, tiger**

**- Class is not a real world entity It is a blueprint**

**- Class does not occupy memory i.e. Animals will not occupy memory but dog, giraffe, cat, tiger will occupy memory**

**Syntax :**

**access\_modifier class ClassName**

**{**

**methods()**

**contructor**

**block/variables**

**nested/inner class**

**}**

**Example :**

**public** **class** Animals

{

**public** **void** dog()

{

System.***out***.println("Helllo world");

}

}

**- object is an instance of class**

**object**

**i.e. Dog is an instance of Animals**

**- object is a real world entity**

**- object consists of :**

**1) Identity -> name**

**2) state/attribute -> color, age**

**3) behavior -> eating, running (methods)**

**Syntax :**

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

**Example :**

**Animals dog = new Animals();**

**dog.bark(); //call method bark()**

**Example :**

**package** com.test.example;

**public** **class** Animal

{

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

{

Animal obj = **new** Animal();

obj.eat();

obj.run();

}

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

{

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

}

**public** **void** run()

{

System.***out***.println("running...");

}

}

**Output :**

eating...

running...

**Question :**

1. **How object is stored in java ?**

**- when we declare a variable of class type, only a reference is created (memory not allocated) and it will give compile time error as *“Error here because obj is not initialized”***

**i.e. Classname obj;**

**- In java, Objects are stored in “heap memory” dynamically But for that we have to initialize object using “new” as below,**

**i.e. Classname obj = new Classname();**

**polymorphism**

**1) Compile time polymorphism : - achieved using method overloading**

**2) Run time polymorphism - achieved using method overriding**

**- Multiple methods with same name but different parameters is method overloading**

**Method overloading**

**- This is compile-time polymorphism as which method to excecute is decided at compile time**

**Real example (MOBILE) :**

**U want to save name of person & his mobile number so we create method saveContact(String name, int mob1) But few people can have more than one mobile numbers so we create method as saveContact(String name, int mob1, int mob2)**

**Rules :**

**- Methods are within same class**

**- Methods have same name and different parameters**

**- All those methods should have same return type**

**Example 1 : (problem with methods same name and same parameters)**

**Test.java**

**public** **class** Test

{

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

{

Test obj = **new** Test();

obj.show();

}

**void** show()

{

System.***out***.println("hello");

}

**void** show()

{

System.***out***.println("java");

}

}

**Output :**

**Compile time error!**

**Explaination :**

Here **compiler gets confused** which method to execute as both are with same name and same parameters

**Example 2 : (solution is to methods same name and different parameters)**

**Test.java**

**public** **class** Test

{

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

{

Test obj = **new** Test();

obj.show();

}

**void** show()

{

System.***out***.println("hello");

}

**void** show(**int** a)

{

System.***out***.println("java");

}

}

**Output :**

**hello**

**Example 3 : (return type should be same)**

**Test.java**

**public** **class** Test

{

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

{

Test obj = **new** Test();

obj.show();

}

**void** show()

{

System.***out***.println("hello");

}

**int** show(**int** a) // used different return type int

{

System.***out***.println("java");

}

}

**Output :**

**Compile time error!**

**Example 4 : (overloading main() method)**

**Test.java**

**public** **class** Test

{

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

{

Test obj = **new** Test();

obj.*main*(10);

}

**public** **static** **void** main(**int** a)

{

System.***out***.println("hello");

}

}

**Output :**

hello

**Example 5 :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

Test obj = **new** Test();

obj.show('c'); // passing char

}

**void** show(String str)

{

System.***out***.println("hello");

}

}

**Output :**

**Compile time error!**

**Explaination :**

compile time error as **no method to pass char value**

**Example 6 : (Automatic promotion)**

**Test.java**

**public** **class** Test

{

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

{

Test obj = **new** Test();

obj.show('c'); // passing char

}

**void** show(String str)

{

System.***out***.println("hello");

}

**void** show(**int** i)

{

System.***out***.println("java");

}

**void** show(Object j)

{

System.***out***.println("world");

}

}

**Output :**

**java**

**Explaination :**

- we passed char value ‘c’

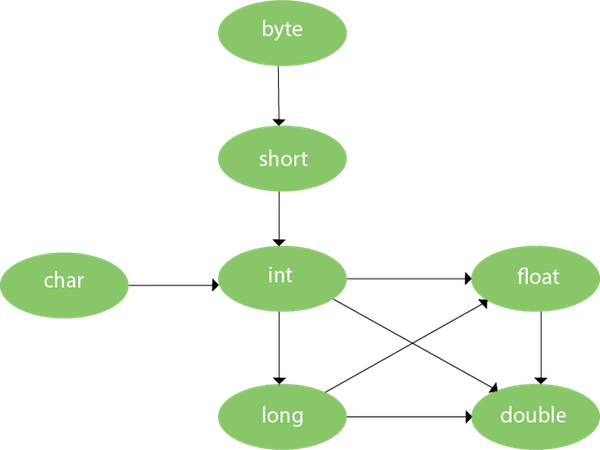
- but we have **no method written** for **char**

- we have methods with parameters **String** and **int**

- but it **will execute method** with parameter **int**

- because of **Automatic promotion** in java as shown below

- as shown in fig, **char** gets promoted to **int**



**Example 7 : (Automatic promotion)**

**Test.java**

**public** **class** Test

{

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

{

Test obj = **new** Test();

obj.show('c'); // passing char

obj.show("this is string");

}

**void** show(String str)

{

System.***out***.println("hello");

}

**void** show(Object j)

{

System.***out***.println("world");

}

}

**Output :**

world

hello

**Explaination :**

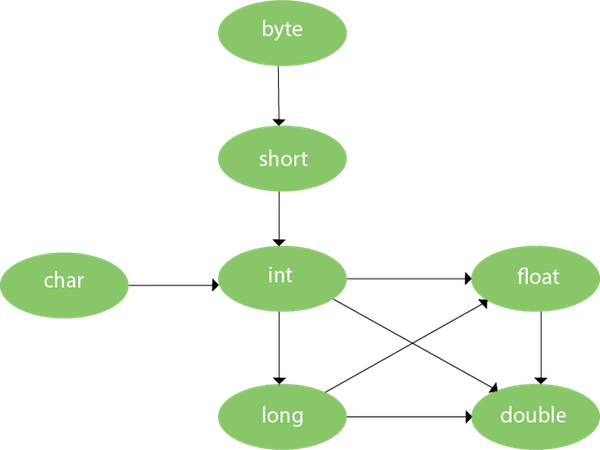
- we passed char value ‘c’

- but we have **no method written** for **char**

- we have methods with parameters **String** and **Object**

- but it **will execute method** with parameter **Object**

- because String class extends Object class



**Example 8 : (Automatic promotion)**

**Test.java**

**public** **class** Test

{

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

{

Test obj = **new** Test();

obj.show(10, 20.5f);

obj.show(20.5f, 40);

}

**void** show(**int** a, **float** b)

{

System.***out***.println("hello");

}

**void** show(**float** a, **int** b)

{

System.***out***.println("world");

}

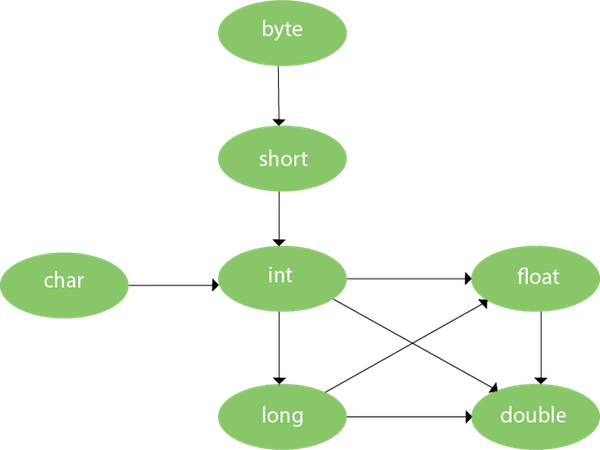
}

**Output :**

hello

world

**Explaination :**



**Example 9 : (Automatic promotion)**

**Test.java**

**public** **class** Test

{

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

{

Test obj = **new** Test();

obj.show(10, 20); // both values are int

}

**void** show(**int** a, **float** b)

{

System.***out***.println("hello");

}

}

**Output :**

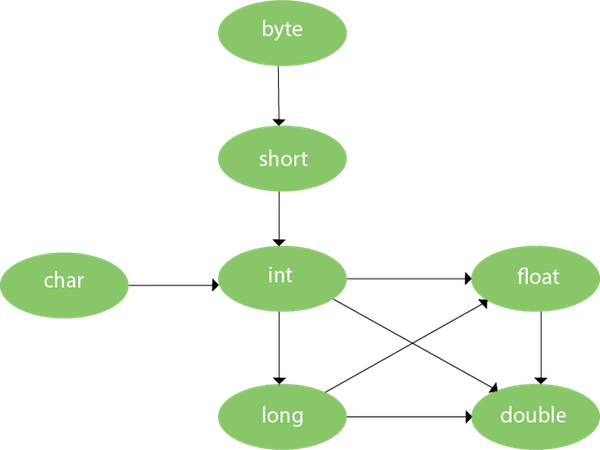
hello

**Explaination :**

- we passed both int values

- it work as per Automatic promotion in java as shown below

- so **int value 20** gets promoted to float here



**Example 10 : (Automatic promotion)**

**Test.java**

public class Test

{

public static void main(String[] args)

{

Test obj = new Test();

obj.show(10, 20); // both values are int

}

void show(int a, float b)

{

System.*out*.println("hello");

}

void show(float a, int b)

{

System.*out*.println("world");

}

}

**Output :**

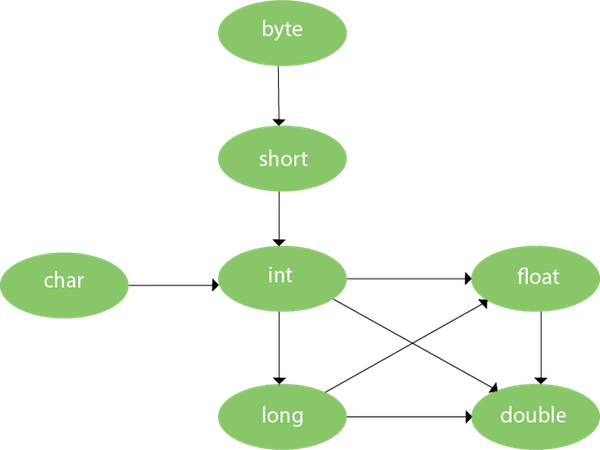
**Compile time error!**

**Explaination :**

- we passed both int values

- it work as per Automatic promotion in java as shown below

- but still **compiler gets confused** due to **2 methods written**



**Example 11 : (Automatic promotion)**

**Test.java**

**public** **class** Test

{

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

{

Test obj = **new** Test();

obj.show(10.5f, 20.6f); // both values are float

}

**void** show(**int** a, **float** b)

{

System.***out***.println("hello");

}

**void** show(**float** a, **int** b)

{

System.***out***.println("world");

}

}

**Output :**

**Compile time error!**

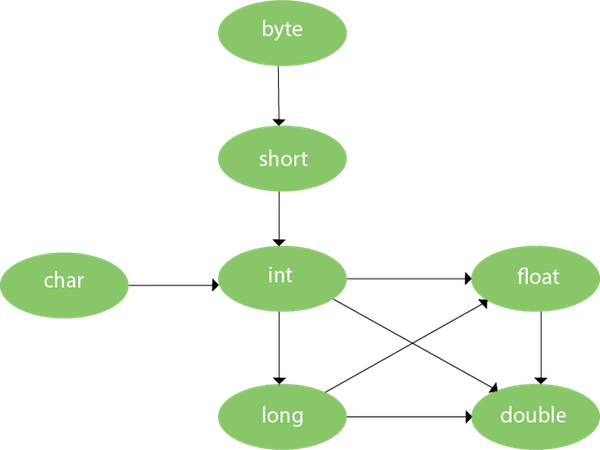
**Explaination :**

- we passed both int values

- it work as per Automatic promotion in java as shown below

- **float** **can not goes back** to **int**

- so we get error here



**Example 12 :**

**Test.java**

**public** **class** Test

{

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

{

Test obj = **new** Test();

obj.show(10);

obj.show();

obj.show(10, 20, 30, 40);

}

**void** show(**int** a)

{

System.***out***.println("hello");

}

**void** show(**int**... a)

{

System.***out***.println("world");

}

}

**Output :**

hello

world

world

- Can achieve **Run-time polymorphism**

**Method overriding**

as which method to execute is decided at run time by JVM When child class assigned to the parent class

i.e. **ParentClass** **p** = **new** **ChildClass**();

then after program execution, JVM figures out the object type and run the method that belongs to that particular object

- JVM will **execute method** of **child class**

**Real world example :**

Bank provides functionality to calculate **RateOfInterates**.

But Its value varies according to banks (even though formula is same).

e.g. SBI & ICICI could provide 8%, 9% rate of interest.

**Rules :**

- Method are in **different class** (subclass)

- Methods must have **same name** and **same parameters** as in parent class

- There must be **IS-A relationship** (i.e. inheritance,interface,abstract)

**Example 1 :**

**Test.java**

**package** com.test.example;

**public** **class** Test **extends** A

{

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

{

A t1 = **new** A();

t1.display();

Test t2 = **new** Test();

t2.display();

A t3 = **new** Test(); // Run-time polymorphism

t3.display();

// Test t4 = new A(); //will give compile time error

}

**public** **void** display() // overriden method

{

System.***out***.println("display in class Test");

}

}

**A.java**

**package** com.test.example;

**public** **class** A

{

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

{

System.***out***.println("display in class A");

}

}

**Output :**

display in class A

display in class Test

display in class Test

**Example 2 :**

**Test.java**

public class Test

{

public static void main(String args[])

{

SBI s = new SBI();

ICICI i = new ICICI();

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

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

Bank bank = new ICICI(); // child class ICICI assigned to the parent class Bank reference

System.*out*.println("RUNTIME POLYMORPHISM : " + bank.getRateOfInterest());

}

}

**Bank.java**

**public** **class** Bank

{

**int** getRateOfInterest()

{

**return** 0;

}

}

**SBI.java**

**public** **class** SBI **extends** Bank

{

**int** getRateOfInterest()

{

**return** 8;

}

}

**ICICI.java**

**public** **class** ICICI **extends** Bank

{

**int** getRateOfInterest()

{

**return** 7;

}

}

**Output :**

SBI Rate of Interest: 8

ICICI Rate of Interest: 7

**RUNTIME POLYMORPHISM** : **7**

**Example 3 : (using different return types)**

**Test.java**

**package** com.test.example;

**public** **class** Test **extends** XYZ

{

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

{

Test obj = **new** Test();

obj.show();

XYZ st = **new** XYZ();

st.show();

}

String show() // ovrriden method

{

System.***out***.println("world");

**return** **null**;

}

}

**XYZ.java**

**package** com.test.example;

**public** **class** XYZ

{

Object show()

{

System.***out***.println("hello");

**return** **null**;

}

}

**Output :**

world

hello

**Explaination :**

**- generally different return types not allowed in method overriding**

**- but its allowed if parent class has parent return type**

**- in our example Object is parent return type of String**

**Example 4 : (using different return types)**

**Test.java**

**package** com.test.example;

**public** **class** Test **extends** XYZ

{

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

{

Test obj = **new** Test();

obj.show();

XYZ st = **new** XYZ();

st.show();

}

String show() // ovrriden method

{

System.***out***.println("world");

**return** **null**;

}

}

**XYZ.java**

**package** com.test.example;

**public** **class** XYZ

{

**int** show()

{

System.***out***.println("hello");

**return** 12;

}

}

**Output :**

Compile time error!

**Explaination :**

**- generally different return types not allowed in method overriding**

**- in our example int is not parent return type of String**

**- so we get error**

**Example 5 : (using different access modifiers)**

**Test.java**

**package** com.test.example;

**public** **class** Test **extends** XYZ

{

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

{

Test obj = **new** Test();

obj.show();

XYZ st = **new** XYZ();

st.show();

}

**public** **void** show() // overriden method

{

System.***out***.println("world");

}

}

**XYZ.java**

**package** com.test.example;

**public** **class** XYZ

{

**protected** **void** show()

{

System.***out***.println("hello");

}

}

**Output :**

world

hello

**Explaination :**

- different access modifiers are **allowed** in method overriding

- but **child class** should have **same or less restrict** access modifier than its **parent class**

- in our example **public** of **child class** is **less restricted** than **protected** of **parent class**

**Example 6 : (using different access modifiers)**

**Test.java**

**package** com.test.example;

**public** **class** Test **extends** XYZ

{

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

{

Test obj = **new** Test();

obj.show();

XYZ st = **new** XYZ();

st.show();

}

**void** show() // ovrriden method

{

System.***out***.println("world");

}

}

**XYZ.java**

**package** com.test.example;

**public** **class** XYZ

{

**protected** **void** show()

{

System.***out***.println("hello");

}

}

**Output :**

Compile time error!

**Explaination :**

- different access modifiers are **allowed** in method overriding

- but **child class** should have **same or less restrict** access modifier than its **parent class**

- in our example **default** of **child class** is **more restricted** than **public** of **parent class**

**Example 7 : (using super keyword)**

**Test.java**

**package** com.test.example;

**public** **class** Test **extends** XYZ

{

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

{

Test obj = **new** Test();

obj.show();

}

**void** show() // ovrriden method

{

System.***out***.println("world");

**super**.show();

}

}

**XYZ.java**

**package** com.test.example;

**public** **class** XYZ

{

**void** show()

{

System.***out***.println("hello");

}

}

**Output :**

world

hello

**Questions :**

1. Can we override **static method** ?

* **NO**. because **static method** belongs to **class** while **instance method** belongs to **object**.

1. Can we override main method?

* **NO**. As **main method** is **static**.

1. Can we override **synchronized method** ?

* **YES**

1. Which methods **can not** override ?

* **final** , **staic**, **private**

**constructor**

- Constructor is a block **similar to method**.

- It is used to **initialize objects**

- It also perform operations like **object creation**, **starting thread**, **calling a method**

- It is invoked when class is created

**Rules :**

1) **Constructor name** must be same as **class name**

2) **Don’t** use any **return type** or **void**.

E.g. public ConstructorName **//is correct.**

Public **void** ConstructorName **//is incorrect**

public **int** ConstructorName **//is incorrect**

3) **Don’t** invoke **explicitly**

E.g. A obj =new **A(6,”hello”);** **//is correct**.

------------------------------

A obj =new A();

**Obj.A(6,”hello”);** **//is incorrect**

4) **Can not** be **abstract**, **final**, **static**, **synchronized**

**Syntax :**

**access\_modifier className()**

**{**

**//code**

**}**

**Example :**

**public Test()**

**{**

**//code**

**}**

**Types of constructors :**

1. **default / no-args**
2. **parameterized**

**Example : (problem without constructor)**

**Employee.java**

**package** com.test.example;

**public** **class** Employee

{

String name;

**int** id;

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

{

Employee e1 = **new** Employee();

Employee e2 = **new** Employee();

System.***out***.println("e1.name : " + e1.name);

System.***out***.println("e1.id : " + e1.id);

System.***out***.println("e2.name : " + e2.name);

System.***out***.println("e2.id : " + e2.id);

}

}

**Output :**

e1.name : null

e1.id : 0

e2.name : null

e2.id : 0

**Explaination :**

**Both objects getting same values**

**Example : (solved problem without constructor)**

**Employee.java**

**package** com.test.example;

**public** **class** Employee

{

String name;

**int** id;

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

{

Employee e1 = **new** Employee();

e1.name = "krishna";

e1.id = 10;

Employee e2 = **new** Employee();

e2.name = "panduranga";

e2.id = 20;

System.***out***.println("e1.name : " + e1.name);

System.***out***.println("e1.id : " + e1.id);

System.***out***.println("---------------");

System.***out***.println("e2.name : " + e2.name);

System.***out***.println("e2.id : " + e2.id);

}

}

**Output :**

e1.name : krishna

e1.id : 10

---------------

e2.name : panduranga

e2.id : 20

**Explanation :**

**- Both objects getting different values**

- To solve previous problem Here we just added below 2 lines,

e1.name = "krishna";

e1.id = 10;

But what if we want **such 1000 objects** **Will u still add this 2 lines 1000 times**?

Answer is **No**.

The better solution for this problem is to create the **constructor**

**Example : (solved problem with constructor)**

**Employee.java**

**package** com.test.example;

**public** **class** Employee

{

String name;

**int** id;

Employee(String name, **int** id)

{

**this**.name = name;

**this**.id = id;

}

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

{

Employee e1 = **new** Employee("krishna", 10);

Employee e2 = **new** Employee("panduranga", 20);

System.***out***.println("e1.name : " + e1.name);

System.***out***.println("e1.id : " + e1.id);

System.***out***.println("---------------");

System.***out***.println("e2.name : " + e2.name);

System.***out***.println("e2.id : " + e2.id);

}

}

**Output :**

e1.name : krishna

e1.id : 10

---------------

e2.name : panduranga

e2.id : 20

**Explaination :**

**- Both objects getting different values**

**Example : (constructor overloading)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

**private** Test(**int** i, String s) // constructor

{

System.***out***.println("i : " + i);

System.***out***.println("s : " + s);

}

**private** Test(**int** j, String d, **int** k) // overloading of above constructor

{

System.***out***.println("j : " + j);

System.***out***.println("d : " + d);

System.***out***.println("k : " + k);

}

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

{

Test b = **new** Test(5, "hello world");

System.***out***.println("------------------");

Test c = **new** Test(6, "hello world", 16);

}

}

**Output :**

i : 5

s : hello world

------------------

j : 6

d : hello world

k : 16

**Example : (constructor chaining)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

Test()

{

**this**(10);

System.***out***.println("constructor 1 ...");

}

Test(**int** i)

{

**this**(20, "hit");

System.***out***.println("constructor 2 ...");

}

**private** Test(**int** j, String d)

{

System.***out***.println("constructor 3 ...");

}

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

{

Test t = **new** Test();

}

}

**Output :**

constructor 3 ...

constructor 2 ...

constructor 1 ...

**Default/ no-args constructor**

**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

Test() //no-args constructor

{

System.***out***.println("Hello world!");

}

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

{

Test b = **new** Test(); // call to constructor

}

}

**Output :**

Hello world!

**parameterized constructor**

**Example :**

**Test.java**

**public** **class** Test

{

Test(**int** i) //parameterized constructor

{

System.***out***.println("number is : "+i);

}

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

{

Test b = **new** Test(10); // call to constructor

}

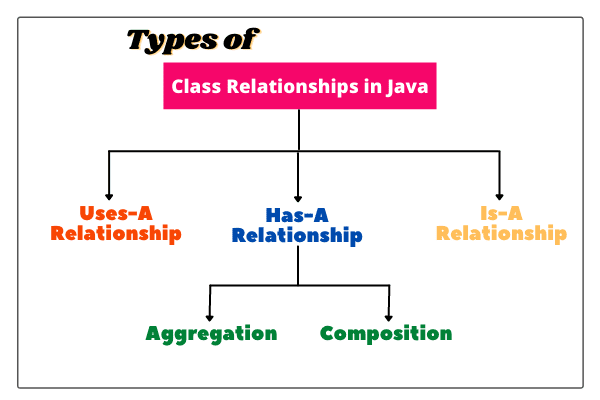
}

**Output :**

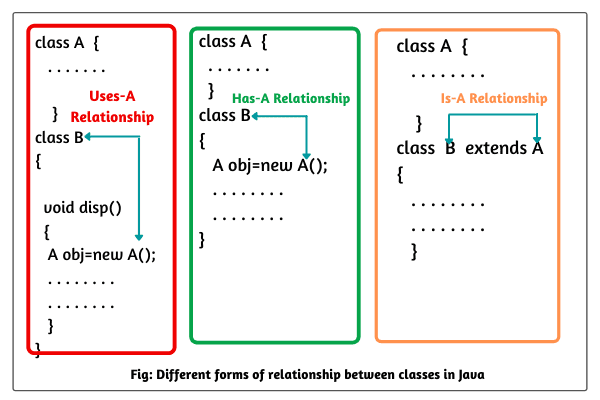
number is : 10

**Types of relationship :**

**relationship**



**Examples of relationship :**



**Uses-A relationship**

**- Dependent relationship**

**- when method of one class uses an object of another class.**

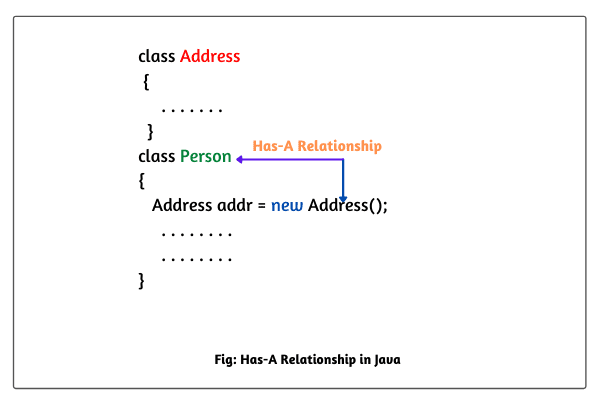
**Has-A relationship**

**- Association relationship**

**- When an object of one class is created as data member inside another class**

**- In below example,**

**Object of Class Address is ceated as data member inside another class Person**



**- Association means creating relationship between two classes**

**Example :**

**Employee.java**

**public** **class** Employee

{

String name;

Address address; //association

**public** Employee(String name, Address address)

{

**this**.name = name;

**this**.address = address;

}

**...**

}

**Address.java**

**public** **class** Address

{

String state, country;

**public** Address(String state, String country)

{

**this**.state = state;

**this**.country = country;

}

}

**Types of Association :**

1. **Aggregation :**

**- When classes are not dependent on each other then it called as aggregation**

**- suppose we delete one class then other class will not have any effect on it. Means it will not stop working.**

**- So it represents weak bonding**

**Example :**

**Employee.java**

**public** **class** Employee

{

String name;

Address address; //association

**public** Employee(String name, Address address)

{

**this**.name = name;

**this**.address = address;

}

**void** display()

{

System.***out***.println("address of " + name +" is " + address.state + " " + address.country);

}

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

{

Address address = **new** Address("UP", "india");

Employee e = **new** Employee("varun", address);

e.display();

}

}

**Address.java**

**public** **class** Address

{

String state, country;

**public** Address(String state, String country)

{

**this**.state = state;

**this**.country = country;

}

}

**Output :**

**address of varun is UP india**

1. **Composition :**

**- When classes are dependent on each other then it called as composition**

**- suppose we delete one class then other class will have effect on it.**

**- So it represents strong bonding**

**- for example,**

**car has a engine**

**so if we delete engine from car then car will not work.**

**Example :**

**Car.java**

**public** **class** Car

{

String modelName;

Engine engine; //association

Car(String modelName)

{

**this**.modelName=modelName;

}

**void** show()

{

System.***out***.println("car model name is : "+modelName);

}

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

{

Engine engine=**new** Engine(7);

Car car=**new** Car("Ferrari");

car.show();

}

}

**Engine.java**

**public** **class** Engine

{

**int** gen;

//composition

**public** Engine(**int** i)

{

**this**.gen=gen;

System.***out***.println("Car engine is started...");

}

}

**Output :**

**Car engine is started...**

car model name is : Ferrari

**- Inheritance relationship**

**IS-A relationship**

**- relationship between two classes in which one class extends another class**

**- we will learn this in detail in the next topic**

**- Inheritance is the mechanism where object of subclass acquires all behaviours and properties of its parent class**

**Inheritance**

**- Helps in reusability of code**

**- Represents IS-A relationship (e.g. Dog IS A subclass of its parent Animal class)**

**- We can achieve polymorphism (method overriding)**

**- Its disadvantage is that it is tightly coupled i.e. if we change in one class it also affects another class**

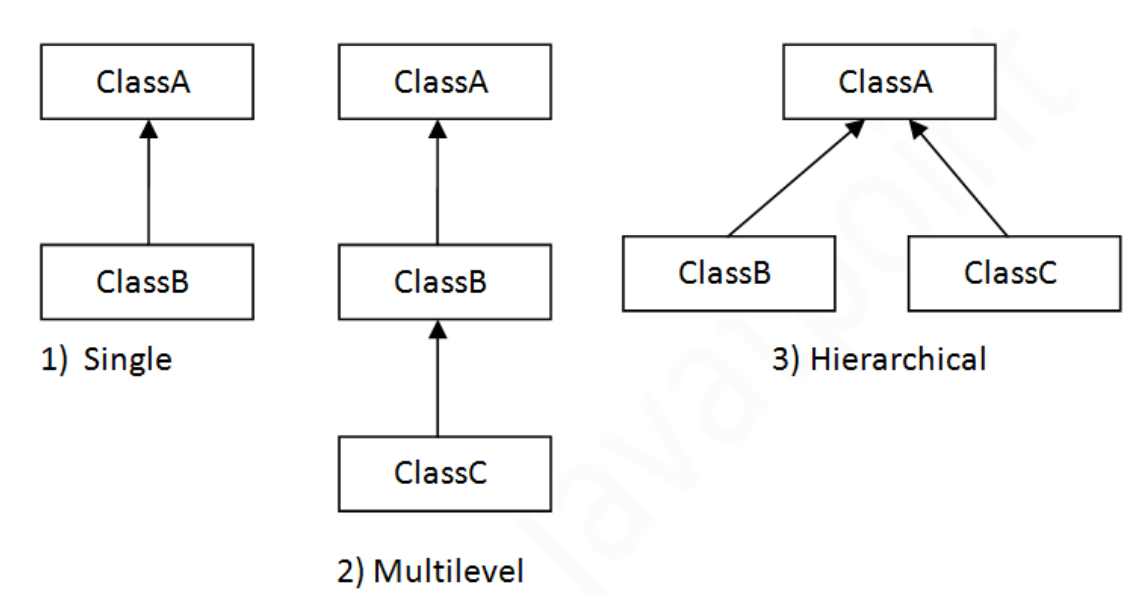
**Syntax :**

**public class Subclass extends ParentClass**

**{**

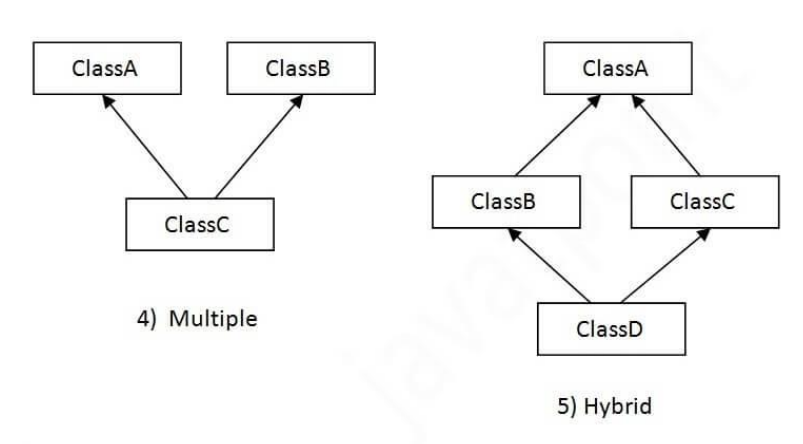
**}**

**Types :**



**Note :**

**- Multiple inheritance is not possible in java**



**single level**

**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

Animal a = **new** Animal();

a.eat();

System.***out***.println("----------------");

Dog d = **new** Dog();

d.bark();

d.eat();

}

}

**Animal.java**

**package** com.test.example;

**public** **class** Animal

{

**void** eat()

{

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

}

}

**Dog.java**

**package** com.test.example;

**public** **class** Dog **extends** Animal

{

**void** bark()

{

System.***out***.println("barking...");

}

}

**Output :**

eating...

**----------------**

barking...

eating...

**multi level**

**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

System.***out***.println("####### class with same class obj creation #######");

Animal a = **new** Animal(); // contain methods of Animal only

a.eat();

Dog d = **new** Dog(); // contain methods of Animal, Dog only

d.bark();

d.eat();

BabyDog b = **new** BabyDog(); // contain methods of Animal, Dog, BabyDog

b.bark();

b.eat();

b.weep();

System.***out***.println("####### class with Diff class obj creation #######");

Animal a1 = **new** Dog(); // contain methods of Animal only

a1.eat();

Animal a2 = **new** BabyDog(); // contain methods of Animal only

a2.eat();

Dog d1 = **new** BabyDog(); // contain methods of Animal, Dog only

d1.eat();

d1.bark();

}

}

**Animal.java**

**package** com.test.example;

**public** **class** Animal

{

**void** eat()

{

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

}

}

**Dog.java**

**package** com.test.example;

**public** **class** Dog **extends** Animal

{

**void** bark()

{

System.***out***.println("barking...");

}

}

**BabyDog.java**

**package** com.test.example;

**public** **class** BabyDog **extends** Dog

{

**void** weep()

{

System.***out***.println("weeping...");

}

}

**Output :**

**####### class with same class obj creation #######**

eating...

barking...

eating...

barking...

eating...

weeping...

**####### class with Diff class obj creation #######**

eating...

eating...

eating...

barking...

**hierarchical**

**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

Animal a = **new** Animal();

a.eat();

System.***out***.println("----------------");

Dog d = **new** Dog();

d.bark();

d.eat();

System.***out***.println("----------------");

Cat c = **new** Cat();

c.meow();

c.eat();

}

}

**Animal.java**

**package** com.test.example;

**public** **class** Animal

{

**void** eat()

{

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

}

}

**Dog.java**

**package** com.test.example;

**public** **class** Dog **extends** Animal

{

**void** bark()

{

System.***out***.println("barking...");

}

}

**Cat.java**

**package** com.test.example;

**public** **class** Cat **extends** Animal

{

**void** meow()

{

System.***out***.println("meowing...");

}

}

**Output :**

eating...

**----------------**

barking...

eating...

**----------------**

meowing...

eating...

**Question :**

1. **why multiple inheritance is not allowed in java ?**



**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test **extends** A,B

{

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

{

Test obj = **new** Test();

obj.show();

}

}

**A.java**

**package** com.test.example;

**public** **class** A

{

**void** show()

{

System.***out***.println("class A...");

}

}

**B.java**

**package** com.test.example;

**public** **class** B

{

**void** show()

{

System.***out***.println("class B...");

}

}

**Output :**

**Compile time error!**

**Explaination :**

**- Compiler will get confused that from which class show() method should call class A or class B ?**

1. **Is constructor gets inherited ?**

* **NO**

1. **Is private method gets inherited ?**

* **NO**

1. **If class B extends class A. then who is parent class of class A ?**

* **Object class.**

**As Object class is the topmost class.**

**Abstract class**

- Abstraction can be achieved in **two ways**,

1) **abstract class :** we can achieve **o to 100%** abstraction

2) **interface** : we can achieve **100%** abstraction

- Abstraction is the process of **hiding internal working** and **showing only functionality** to user

- Abstract class can have both abstract and non-abstract methods

**Real world example :**

**SMS sending**. where you only know type and send message. but **don’t know its internal working** how it works.

- Similarly in abstract class or interface we just write method names. But its actual implementation or internal working we write in different class.

**Rules :**

1) Declare abstract class with **keyword** **abstract**

2) It can have both **abstract** and **non-abstract** **methods**

3) It **can not** be **initiated**

4) It can have normal **final** methods.

But that method should not be abstract else it will give compile-time error.

e.g. public **abstract** **final** void show(); **//is incorrect**

5) It can have **constructors** and **static methods** also

**Syntax for class :**

**abstract** class **classname**

{

**//code here**

}

**Syntax for method :**

**abstract** void methodname()**;** **//put semicolon (;) after method**

**Example 1 :**

**Bike.java**

**package** com.test.example;

**public** **class** Bike

{

**void** start()

{

System.***out***.println("bike starts with kick");

}

**public** **static** **void** main()

{

Vehicle v = **new** Vehicle(); // can not create object of astract class

}

}

**Vehicle.java //abstarct class**

**package** com.test.example;

**public** **abstract** **class** Vehicle

{

**abstract** **void** start();

}

**Car.java**

**package** com.test.example;

**public** **class** Car **extends** Vehicle

{

**void** start()

{

System.***out***.println("car starts with key");

}

}

**Output :**

**Compile time error!**

**Example 2 :**

**Bike.java**

**package** com.test.example;

**public** **class** Bike

{

**void** start()

{

System.***out***.println("bike starts with kick");

}

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

{

Car c = **new** Car();

c.start();

Bike b = **new** Bike();

b.start();

}

}

**Vehicle.java //abstract class**

**package** com.test.example;

**public** **abstract** **class** Vehicle

{

**abstract** **void** start();

}

**Car.java**

**package** com.test.example;

**public** **class** Car **extends** Vehicle

{

**void** start()

{

System.***out***.println("car starts with key");

}

}

**Output :**

car starts with key

bike starts with kick

**Example 3 :**

**Test.java**

**public** **class** Test

{

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

{

Bike obj = **new** Honda();

obj.run();

obj.changeGear();

obj.show();

}

}

**Bike.java //abstract class**

**public** **abstract** **class** Bike

{

Bike() // constructor

{

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

}

**abstract** **void** run(); // abstract method

**void** changeGear() // normal method

{

System.***out***.println("gear changed");

}

**public** **final** **void** show() // this final method cannot be overriden

{

System.***out***.println("bike show");

}

}

**Honda.java**

**public** **class** Honda **extends** Bike

{

**void** run()

{

System.***out***.println("running safely..");

}

}

**Output :**

bike is created

running safely..

gear changed

bike show

**interface**

- Interface is the **blueprint of a class**

- It helps in **achieving abstraction**

- It helps in achieving **multiple inheritance**

**Rules :**

1) It contains **only abstract methods** , not contain normal methods with body

By default methods are **public abstract**

e.g.we create method,

void show();

then compiler automatically makes it as,

public abstract void show();

2) By default variables are **public static final**,

e.g.we create variable,

int a=10;

then compiler automatically makes it as,

public static final int a=10;

3) Since **java 8**, it can have **default & static methods** also

4) Since **java 9**, it can have **private methods** also

**Syntax for creation :**

**interface interfaceName**

**{**

**//code here**

**}**

**Syntax for calling interface :**

**Public class ClassName implements interfaceName**

**{**

**//code here**

**}**

**Example 1 :**

**Test.java**

**public** **class** Test **implements** Sample

{

@Override

**public** **void** show()

{

System.***out***.println("In method show");

}

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

{

// Sample s = new Sample(); //COMPILE ERROR as can’t create object of interface

Test t = **new** Test();

t.show();

t.m1();

}

}

**Sample.java //interface**

**public** **interface** Sample

{

**void** show(); // internally it is public abstract void show();

**int** ***a*** = 10; // internally it is public static final int a=10;

**default** **void** m1() // default method of java 8

{

System.***out***.println("In default method");

*m2*();

m3();

}

**static** **void** m2() // sataic method of java 8

{

System.***out***.println("In static method");

}

**private** **void** m3() // private method of java 9

{

System.out.println("In private method");

}

}

**Output :**

In method show

In default method

In static method

In private method

**Example 2 :**

**Test.java**

**public** **class** Test

{

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

{

//Bank b =new Bank(); //INCORRECT will give compile time error

//SBI sbi=new PNB(); //INCORRECT will give compile time error

SBI sbi=**new** SBI();

System.***out***.println("SBI RI: " + sbi.rateOfInterest());

PNB pnb =**new** PNB();

System.***out***.println("PNB RI: " + pnb.rateOfInterest());

Bank bank1 = **new** SBI();

System.***out***.println("SBI RI: " + bank1.rateOfInterest());

Bank bank2 = **new** PNB();

System.***out***.println("PNB RI: " + bank2.rateOfInterest());

System.***out***.println("---------------------------------------");

bank2.msg();//calling msg method

System.***out***.println("---------------------------------------");

System.***out***.println("CUBE is : "+ Bank.*cube*(3)); //as method is static it can't invoked like bank2.Cube();

}

}

**Bank.java //interface**

**public** **interface** Bank

{

**public** **float** rateOfInterest();

**default** **void** msg()

{

System.***out***.println("JAVA 8 default method");

}

**public** **static** **int** cube(**int** num)

{

**return** num\*num\*num;

}

}

**SBI.java**

**public** **class** SBI **implements** Bank

{

**public** **float** rateOfInterest()

{

**return** 9.15f;

}

}

**PNB.java**

**public** **class** PNB **implements** Bank

{

**public** **float** rateOfInterest()

{

**return** 9.7f;

}

}

**Output :**

SBI RI: 9.15

PNB RI: 9.7

SBI RI: 9.15

PNB RI: 9.7

---------------------------------------

JAVA 8 default method

---------------------------------------

CUBE is : 27

**Example 3 : (multiple inheritance achieved by interface)**

**Test.java**

**public** **class** Test **impments** A, B

{

**public** **void** show()

{

System.***out***.println("showing...");

}

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

{

Test obj = **new** Test();

obj.show();

}

}

**A.java //interface**

**public** **interface** A

{

**void** show();

}

**B.java //interface**

**public** **interface** B

{

**void** show();

}

**Output :**

showing...

**Explanation :**

**Even though interface A and interface B has same method name show() we are able to call it.**

**Example 3 : (interface extends interface)**

**Test.java**

**package** com.test.example;

**public** **class** Test **implements** B

{

**public** **void** print()

{

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

}

**public** **void** show()

{

System.***out***.println("Welcome");

}

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

{

Test obj = **new** Test();

obj.print();

obj.show();

}

}

**A.java //interface**

**public** **interface** A

{

**void** print();

}

**B.java //interface**

**public** **interface** B **extends** A

{

**void** show();

}

**Output :**

Hello

Welcome

**Question :**

1. What is **Marked** or **Tagged** **interface**?

* An **empty interface**.

Means interface which **don’t contain any method** is known as marked/taged interface.

e.g. **Seriliazable**, **Cloneable**, **Remote**, **EventListener** are few examples of marked interface.

This interfaces provides **some useful information to JVM** so that JVM may perform some operation using that information.

1. Can **interface** **implements** **abstract class**?

* **NO**.

because **abstract is a class** **not an interface**.

1. Can **abstract class** **implements** **interface**?

* **Yes**.

because **class** **implements** **interface**.

1. Explain which **access-modifiers** are suitable for **abstract class** and **abstract methods**?

* Below table shows all conditions :

|  |  |
| --- | --- |
| **public** abstract class Printable  {  **public** abstract void print();  } | **Abstract Class and abstract method can public** |
| **private** abstract class Printable  {  **private** abstract void print();  } | **Abstract Class and abstract method cannot private** |
| **protected** abstract class Printable  {  **protected** abstract void print();  } | **Abstract Class cannot be protected**  **Abstract method can be protected** |
| abstract **final** class Printable  {  abstract **final** void print();    private **final** void msg()  {  System.out.println("hello");  }  } | **Abstract Class cannot be final**  **Abstract method cannot be final**  **normal method can be final and private/protected in abstract class** |

1. Explain which **access-modifiers** are suitable for **interface** and its **methods**?

* Below table shows all conditions :

|  |  |
| --- | --- |
| **public** interface Showable  {  **public** void show();  } | **interface and its method can public** |
| **private** interface Showable  {  **private** void show();  } | **interface and its method cannot private** |
| **protected** interface Showable  {  **protected** void show();  } | **interface and its method cannot protected** |
| public **final** interface Showable  {  public **final** void show();    static **final** int cube(int x)  {  return x \* x \* x;  }  } | **interface its method and even normal method in interface cannot be final** |

1. Can **interface** **implements** **interface**?

* **No**

1. Can **interface** **extends** **interface**?

* **YES**

1. **Abstract class** **vs** **interface**?



|  |  |  |
| --- | --- | --- |
| **Sr no.** | **Abstract class** | **interface** |
| **1.** | Can create **constructor** | **Can not** create **constructor** |
| **2.** | **Can not** achieve **multiple inheritance** | Can achieve **multiple inheritance** |
| **3.** | Less **strict** | **More** **strict** |

**Encapsulation**

- Encapsulation means **wrapped capsule**

- In which technique of **data hiding** is used

**Real world Example (water tank) :**

Suppose Water is our **data** which is secured in tank and only can be accessible via tap.

It means we have done data hiding on water using tank And filling water in yank via pipe(i.e.**setter method**),

accessing data (water) via tap (i.e.**getter method**)



- Java bean is the best example of fully encapsulation

**Steps to perform encapsulation :**

1. make data member **private**
2. create **setter method**
3. create **gettter method**

- we can achieve encapsulation by making all data members private and create public setter and getter methods.

- By creating setter and getter method we can make class read-only and write-only. Because in setters we set some values which is write only And in getters we get values which is read only. e.g.we read xml using getter methods and write(create) json using setter method

- Encapsulation helps in **security of data**.

As we **don’t allow unauthorized access** to it.

As data can not be accessible directly. It can be accessed only using **setter & getter** methods.

**Example 1 : (problem without encapsulation)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

Employee e = **new** Employee();

e.empId = 101; // by doing this data is accessible anywhere

}

}

**Emploee.java**

**package** com.test.example;

**public** **class** Employee

{

**int** empId;

}

**Explanation :**

In above example,

Data is **updatable/accessible** anywhere

So **security** is the problem

**Example 2 : (using encapsulation)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

Employee e = **new** Employee();

e.setEmpId(101);

System.***out***.println("ID : " + e.getEmpId());

}

}

**Emploee.java**

**package** com.test.example;

**public** **class** Employee

{

**private** **int** empId;

**public** **int** getEmpId()

{

**return** empId;

}

**public** **void** setEmpId(**int** empId)

{

**this**.empId = empId;

}

}

**Output :**

ID : 101

**Example 3 : (using encapsulation)**

**Test.java**

**public** **class** Test

{

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

{

Employee e = **new** Employee();

e.setEmpId(16);

System.***out***.println("ID : " + e.getEmpId());

}

}

**Emploee.java**

**public** **class** Employee

{

**private** **int** empId;

**public** **int** getEmpId()

{

**if**(empId>20)

{

**return** empId;

}

**return** 0;

}

**public** **void** setEmpId(**int** empId)

{

**this**.empId = empId;

}

}

**Output :**

ID : 0

**Explanation :**

**- here we authenticated employee Id**

**- If user enterd valid empId then only he will get his correct Id**

**- else he will get 0 in return**

**Question :**

1. Is **Data hiding** vs **Encapsulation**?

* **- Data hiding** is the **technique to hide data**

where we keep data members as private.

* **Encapsulation** is the process which **uses data hiding** to secure data but data is accessible using setters & getters methods

1. Is **Abstraction** V/S **Encapsulation** ?



|  |  |
| --- | --- |
| **Abstraction** | **Encapsulation** |
| **Abstraction is about hiding working And shows only functionality** | **Encapsulation is about hiding data** |
| **e.g. mobile phones we don’t know its internal coding but we know its functionaities** | **e.g. water tank where we hide water and give access only via tap** |

1. why **Encapsulation** is needed?

* **- If we don’t use encapsulation then instance variables of class are easily accesible to other classes**
* **This can lead to modifying values of instance variables**
* **In encapsulation we are meking those instance variables private (data hiding)**
* **Which can be only accessible by methods of that class (setter and getter)**
* **So it makes instance variables protected**
* **We can also put some authentication on setter and getter method to validate a user**

**Note :** **There is no call by value in java**

**Call by reference**

**Example : (without call by reference)**

**Test.java**

**public** **class** Test

{

**int** data = 50;

**void** change(**int** data)

{

data = data + 100;// changes will be in the local variable only

}

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

{

Test op = **new** Test();

System.***out***.println("before change " + op.data);

op.change(500);

System.***out***.println("after change " + op.data);

}

}

**Output :**

before change 50

after change 50

**Example : (using call by reference)**

**Test.java**

**public** **class** Test

{

**int** data = 50;

**void** change(Test op)

{

op.data = op.data + 100; // changes will be in the instance variable

}

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

{

Test op = **new** Test();

System.***out***.println("before change " + op.data);

op.change(op);// passing object

System.***out***.println("after change " + op.data);

}

}

**Output :**

before change 50

after change 150

**Recursion**

- **Recursion** is the process in which a **method calls itself** infinitely or continuously

**Syntax :**

returnType **methodName()**

{

**methodName();**

}

**Example : (recursion with infinite loop)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

**static** **void** show()

{

System.***out***.println("hello");

*show*(); // method calls itself here

}

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

{

*show*();

}

}

**Output :**

hello

hello

hello

. . . .

**helloException in thread "main" java.lang.StackOverflowError**

**Example : (recursion with finite loop)**

**Test.java**

**public** **class** Test

{

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

{

**int** result = *sum*(15);

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

}

**public** **static** **int** sum(**int** k)

{

**if** (k > 0)

{

**return** k + *sum*(k - 1); // method calls itself here

} **else**

{

**return** 0;

}

}

}

**Output :**

120

**instanceof**

- It checks whether the **object** is an **instance of** specified **class / subclass / interface**

- It returns **true** or **false** value

**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

Printable p = **new** XYZ();

// beacuse we initialized XYZ class with printable here it will be instance of it

Call c = **new** Call();

c.invoke(p);

}

}

**Call.java**

**package** com.test.example;

**public** **class** Call

{

**void** invoke(Printable p) // upcasting

{

**if** (p **instanceof** ABC)

{

System.***out***.println("p is instance of ABC");

ABC a = (ABC) p;// Downcasting

a.a();

}

**if** (p **instanceof** XYZ)

{

System.***out***.println("p is instance of XYZ");

XYZ b = (XYZ) p;// Downcasting

b.b();

}

}

}// end of Call class

**ABC.java**

**package** com.test.example;

**public** **class** ABC **implements** Printable

{

**public** **void** a()

{

System.***out***.println("a method");

}

}

**XYZ.java**

**package** com.test.example;

**public** **class** XYZ **implements** Printable

{

**public** **void** b()

{

System.***out***.println("b method");

}

}

**Printable.java**

**package** com.test.example;

**public** **interface** Printable

{

}

**Output :**

**p** is **instance of** **XYZ**

b method

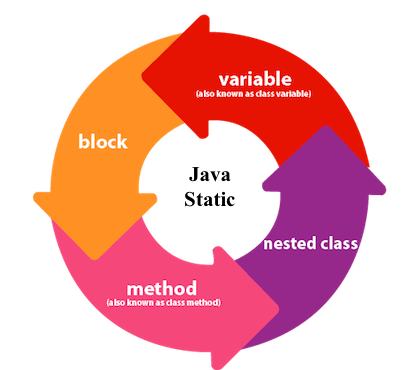
**static keyword**

**- static keyword helps in memory management mainly**

**- staic is class level**

**- static data gets stored in class/method area of JVM**

**- variable, Nested class, block, method can be static**



- It **allocates memory once only** at time of class loading

**static variable**

- So it helps in **saving memory**

**Example 1 : (static variable can not declare inside method)**

**Test.java**

**public** **class** Test

{

**static** **int** *a* = 10;

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

{

System.***out***.println("a : " + *a*);

}

**void** m1()

{

**static** **int** b = 20; // compile time error

}

}

**Output :**

**compile time error!**

**Example 2 : (static variable can be called without creating object)**

**Test.java**

**public** **class** Test

{

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

{

System.***out***.println(Demo.*a*); // called without creating object

}

}

**Demo.java**

**public** **class** Demo

{

**static** **int** *a* = 10;

}

**Output :**

10

**Example 3 : (problem without static variable)**

**Test.java**

**public** **class** Test

{

**int** empId;

String name;

String company;

**public** Test(**int** empId, String name, String company)

{

**this**.empId = empId;

**this**.name = name;

**this**.company = company;

}

**void** display()

{

System.***out***.println(empId + " " + name + " " + company);

}

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

{

Test e1 = **new** Test(101, "Rahul", "Capgemini");

e1.display();

Test e2 = **new** Test(102, "Suresh", "Capgemini");

e2.display();

}

}

**Output :**

101 Rahul **Capgemini**

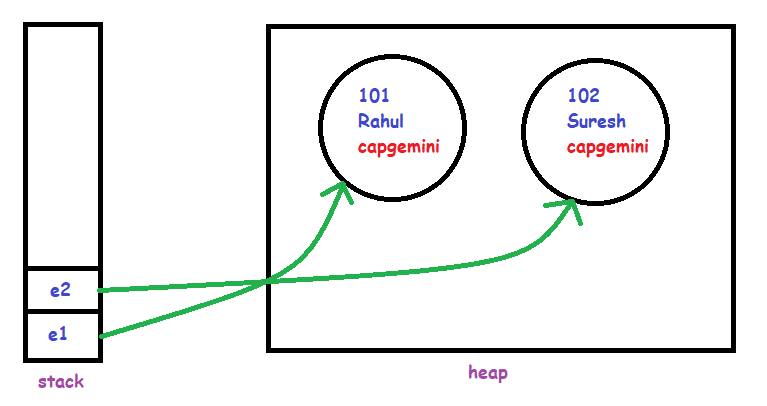
102 Suresh **Capgemini**

**Explanation :**

- In above program **Capgemini** is the same data we are using repetively

- If we create such 1000 objects say **e1,e2,….,e1000**

- Then it will need lots of memory to store capgemini



**Example 4 : (using static variable)**

**Test.java**

**public** **class** Test

{

**int** empId;

String name;

**static** String *company* = "Capgemini";

**public** Test(**int** empId, String name)

{

**this**.empId = empId;

**this**.name = name;

}

**void** display()

{

System.***out***.println(empId + " " + name + " " + *company*);

}

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

{

Test e1 = **new** Test(101, "Rahul");

e1.display();

Test e2 = **new** Test(102, "Suresh");

e2.display();

}

}

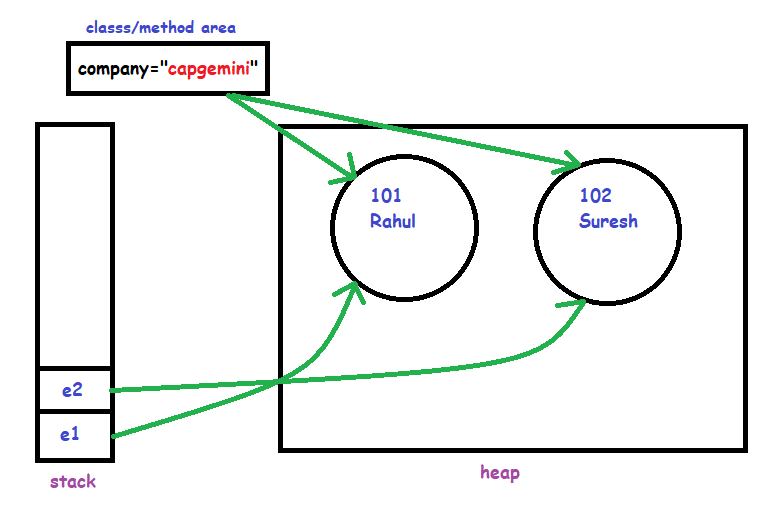
**Output :**

101 Rahul **Capgemini**

102 Suresh **Capgemini**

**Explanation :**

- Here we saved memory by storing **Capgemini** in static variable **once only**



**Example 5 : (problem without static variable)**

**Test.java**

**public** **class** Test

{

**int** count = 0;// will get memory each time when the instance is created

Test()

{

count++;// incrementing value

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

}

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

{

// Creating objects

Test c1 = **new** Test();

Test c2 = **new** Test();

Test c3 = **new** Test();

}

}

**Output :**

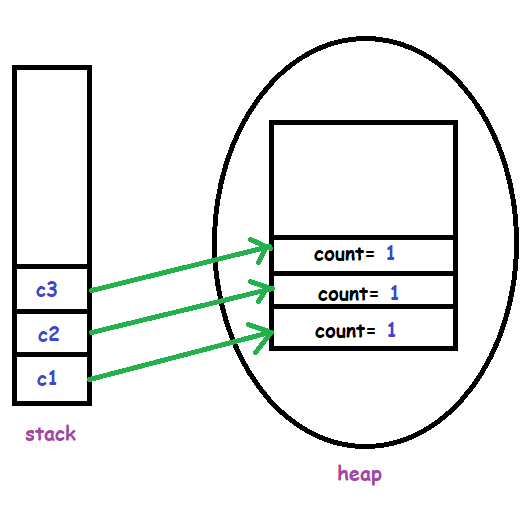
1

1

1

**Explanation :**

- Here int count = 0 gets **initialized again and again**



**Example 6 : (using static variable)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

**static** **int** *count* = 0; // will get memory only once and retain its value

Test()

{

*count*++;// incrementing value

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

}

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

{

// Creating objects

Test c1 = **new** Test(); // call to constructor

Test c2 = **new** Test(); // call to constructor

Test c3 = **new** Test(); // call to constructor

}

}

**Output :**

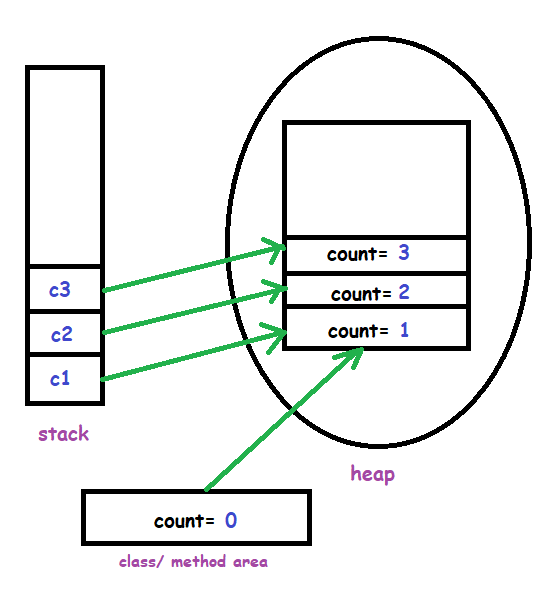
1

2

3

**Explanation :**

Here **static** int count = 0 gets **initialized once only**



- **No need** to **create object** of class we can call static method using classname only.

**static methods**

i.e. **className**.**methodName();**

- **Variables inside static method** also need to be **static**.

i.e. static int k;

**Example 1 :**

**Test.java**

**public** **class** Test

{

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

{

Test t = **new** Test();

t.*show*();

*show*(); // can call static method directly if method is within current class

Test.*show*(); // can call static method directly if method is within current class/ouside class

}

**static** **void** show()

{

System.***out***.println("showing...");

}

}

**Output :**

showing...

showing...

showing...

**Example 2 :**

**Test.java**

**public** **class** Test

{

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

{

Sample t = **new** Sample();

t.*show*();

//show(); can call static method directly if method is within current class

Sample.*show*(); // can call static method using className if method is in class/ouside class

}

}

**Sample.java**

**public** **class** Sample

{

**static** **void** show()

{

System.***out***.println("showing...");

}

}

**Output :**

showing...

showing...

**Example 3 : (static methods can access only static data)**

**Test.java**

**public** **class** Test

{

**int** i = 10;

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

{

Test.*show*();

}

**static** **void** show()

{

System.***out***.println(i); // Compile time error

}

}

**Output :**

**Compile time error!**

**Example 4 : (static methods can access only static data)**

**Test.java**

**public** **class** Test

{

**static** **int** *i* = 10;

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

{

Test.*show*();

}

**static** **void** show()

{

System.***out***.println(*i*); // Compile time error

}

}

**Output :**

**10**

**Example 5 : (static methods can be called through static methods only)**

**Test.java**

**public** **class** Test

{

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

{

Test.*show*();

}

**static** **void** show()

{

display(); // Compile time error

}

**void** display()

{

System.***out***.println("displaying");

}

}

**Output :**

**Compile time error!**

**Example 6 : (static methods can be called through static methods only)**

**Test.java**

**public** **class** Test

{

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

{

Test.*show*();

}

**static** **void** show()

{

*display*(); // Compile time error

}

**static** **void** display()

{

System.***out***.println("displaying");

}

}

**Output :**

displaying

**Example 7 : (Can not use this keyword within static methods)**

**Test.java**

**public** **class** Test

{

**int** i = 10;

**static** **int** *j* = 20;

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

{

Test.*show*();

}

**static** **void** show()

{

System.***out***.println(**this**.i); // Compile time error

System.***out***.println(**this**.j); // Compile time error

}

}

**Output :**

**Compile time error!**

**Example 8 : (Can not use super keyword within static methods)**

**Test.java**

**package** com.test.example;

**public** **class** Test **extends** Sample

{

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

{

Test.*show*();

}

**static** **void** show()

{

System.out.println(**super**.i); //Compile time error

}

}

**Sample.java**

**package** com.test.example;

**public** **class** Sample

{

**static** **int** *i* = 10;

}

**Output :**

**Compile time error!**

- It is used to **initialize static data types**

**static block**

- also used to call **native methods** (methods of other language i.e. c, c++, etc)

- It **executes before main method** executes.

- Before JDK 1.6 and older versions we were able to use static blocks without creating main method in class

- But from JDK 1.7 we must use main method to call static blocks

**Syntax :**

**static**

**{**

**//code**

**}**

**Example 1 :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

**static**

{

System.***out***.println("static block 1 is invoked");

}

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

{

System.***out***.println("main method executed");

}

**static**

{

System.***out***.println("static block 2 is invoked");

}

**static**

{

System.***out***.println("static block 3 is invoked");

}

}

**Output :**

static block 1 is invoked

static block 2 is invoked

static block 3 is invoked

**main method executed**

**Example 2 : (initialiing data in static block)**

**Test.java**

**public** **class** Test

{

**static** **int** *i*;

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

{

System.***out***.println("main method executed");

}

**static**

{

*i* = 50;

System.***out***.println("static block 2 is invoked : " + *i*);

}

}

**Output :**

static block 2 is invoked : 50

main method executed

**instance block**

- instance block invoked **after main method**

- But **before constructor**

**Syntax :**

**{**

**//code**

**}**

**Example :**

**Test.java**

**public** **class** Test

{

Test() // constructor

{

System.***out***.println("Constructor Called");

}

{

System.***out***.println("block called"); // instance block

}

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

{

System.***out***.println("main called");

Test a = **new** Test(); // call to constructor

}

}

**Output :**

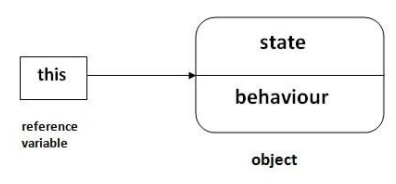
main called

block called

Constructor Called

**this keyword**

**-** Used as a **reference variable** for **current class object**



**Example 1 : (without using this keyword)**

**Test.java**

**public** **class** Test

{

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

{

Sample s = **new** Sample();

s.data(10);

s.show();

}

}

**Sample.java**

**public** **class** Sample

{

**int** i;

**void** data(**int** k) // name of local variable is same

{

i = k;

}

**void** show()

{

System.***out***.println(i); // prints value of instance variable

}

}

**Output :**

**10**

**Example 2 : (problem without using this keyword)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

Sample s = **new** Sample();

s.data(10);

s.show();

}

}

**Sample.java**

**package** com.test.example;

**public** **class** Sample

{

**int** i;

**void** data(**int** i) // name of local variable is same

{

i = i; // it considered i in left side as local variable

}

**void** show()

{

System.***out***.println(i); // prints value of instance variable

}

}

**Output :**

**0**

**Explanation :**

- when we used name of **local variable** same as **instance variable**

- then we not get output as we want

**Example 3 : (using this keyword)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

Sample s = **new** Sample();

s.data(10);

s.show();

}

}

**Sample.java**

**package** com.test.example;

**public** **class** Sample

{

**int** i;

**void** data(**int** i) // name of local variable is same

{

**this**.i = i; // now it considered i in left side as instance variable

}

**void** show()

{

System.***out***.println(i); // prints value of instance variable

}

}

**Output :**

**10**

**Example 4 : (refers instance variable of current class)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

// instance variables

**int** a;

**int** b;

Test(**int** a, **int** b) // Parameterized constructor

{

**this**.a = a;

**this**.b = b;

}

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

{

System.***out***.println("a = " + a + " b = " + b);

}

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

{

Test object = **new** Test(10, 20);

object.display();

}

}

**Output :**

a = 10 b = 20

**Example 5 : (call method of current class)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

Test t = **new** Test();

t.show();

}

**void** display()

{

System.***out***.println("displaying...");

}

**void** show()

{

display(); // compiler call it as this.display() internally

**this**.display();

}

}

**Output :**

displaying...

displaying...

**Explanation :**

- Even though we call method of current class as **display();**

- compiler internally calls it as **this**.display();

**Example 6 : (pass argument to method)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

**void** display(Test obj)

{

System.***out***.println("displaying...");

}

**void** m1()

{

display(**this**);

}

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

{

Test t = **new** Test();

t.m1();

}

}

**Output :**

displaying...

**Explanation :**

- Even though we call method of current class as **display();**

- compiler internally calls it as **this**.display();

**Example 7 : (pass argument to method)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

**int** a;

**int** b;

// Default constructor

Test()

{

a = 10;

b = 20;

}

**void** display(Test obj)

{

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

}

**void** get()

{

display(**this**); // returns current class instance

}

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

{

Test object = **new** Test();

object.get();

}

}

**Output :**

a = 10 b = 20

**Example 8 : (call constructor of current class)**

**Test.java**

**public** **class** Test

{

Test() // Default constructor

{

System.***out***.println("Inside default constructor \n");

}

Test(**int** a) // Parameterized constructor

{

**this**();

System.***out***.println("Inside parameterized constructor");

}

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

{

Test object = **new** Test(10);

}

}

**Output :**

Inside default constructor

Inside parameterized constructor

**Example 9 : (call constructor of current class)**

**Test.java**

**public** **class** Test

{

**int** a;

**int** b;

Test() // Default constructor

{

**this**(10, 20); // refer to parametrized constructor

System.***out***.println("Inside default constructor \n");

}

Test(**int** a, **int** b) // Parameterized constructor

{

**this**.a = a;

**this**.b = b;

System.***out***.println("Inside parameterized constructor");

}

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

{

Test object = **new** Test();

}

}

**Output :**

Inside default constructor

Inside parameterized constructor

**Example 10 : (pass argument to constructor)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

Test t = **new** Test();

t.m1();

}

**void** m1()

{

Sample s = **new** Sample(**this**);

}

}

**Sample.java**

**package** com.test.example;

**public** **class** Sample

{

Sample(Test t)

{

System.***out***.println("in Sample constructor...");

}

}

**Output :**

in Sample constructor...

**Example 11 : (returns instance of current class)**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

Test m1()

{

**return** **this**;

}

**void** m2()

{

System.***out***.println("in m2...");

}

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

{

Test t = **new** Test();

t.m1().m2();

}

}

**Output :**

in m2...

**super keyword**

- **this keyword** refers to **current class object**

- **super keyword** used to refer **parent class object**

**Example 1 :**

**Test.java**

**public** **class** Test **extends** Sample

{

**int** i = 20;

**void** show(**int** i)

{

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

}

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

{

Test t = **new** Test();

t.show(30);

}

}

**Sample.java**

**public** **class** Sample

{

**int** i = 10;

}

**Output :**

30

**Example 2 : (using this keyword)**

**Test.java**

**public** **class** Test **extends** Sample

{

**int** i = 20;

**void** show(**int** i)

{

System.***out***.println(**this**.i);

}

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

{

Test t = **new** Test();

t.show(30);

}

}

**Sample.java**

**public** **class** Sample

{

**int** i = 10;

}

**Output :**

20

**Example 3 : (using super keyword)**

**Test.java**

**public** **class** Test **extends** Sample

{

**int** i = 20;

**void** show(**int** i)

{

System.***out***.println(**super**.i);

}

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

{

Test t = **new** Test();

t.show(30);

}

}

**Sample.java**

**public** **class** Sample

{

**int** i = 10;

}

**Output :**

10

**Example 4 : (using super keyword with variable)**

**Test.java**

**package** com.test.example;

**public** **class** Test **extends** Sample

{

**int** maxSpeed = 180;

**void** display()

{

System.***out***.println("Maximum Speed: " + **super**.maxSpeed); //prints speed of parent class

}

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

{

Test t = **new** Test();

t.display();

}

}

**Sample.java**

**package** com.test.example;

**public** **class** Sample

{

**int** maxSpeed = 120;

}

**Output :**

Maximum Speed: 120

**Example 5 : (using super keyword with method)**

**Test.java**

**package** com.test.example;

**public** **class** Test **extends** Sample

{

**void** message()

{

System.***out***.println("This is CAR");

}

**void** display()

{

**super**.message(); // calls parent class method

}

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

{

Test t = **new** Test();

t.display();

}

}

**Sample.java**

**package** com.test.example;

**public** **class** Sample

{

**void** message()

{

System.***out***.println("This is VEHICLE");

}

}

**Output :**

This is VEHICLE

**Example 6 : (using super keyword with constructor)**

**Test.java**

**package** com.test.example;

**public** **class** Test **extends** Sample

{

Test()

{

**super**();

System.***out***.println("This is CAR");

}

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

{

Test t = **new** Test();

}

}

**Sample.java**

**package** com.test.example;

**public** **class** Sample

{

Sample()

{

System.***out***.println("This is VEHICLE");

}

}

**Output :**

This is VEHICLE

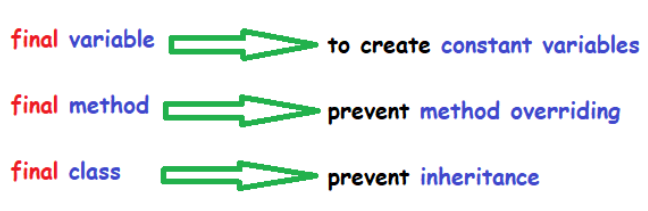
This is CAR

**final keyword**

- **Prevents** **constant variables** access outside method

- **Prevents** **method overriding**

- **Prevents** **inheritance**



**final variable**

**Example :**

**Test.java**

**package** com.test.example;

**public** **class** Test

{

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

{

**final** **int** val = 50;

val = 60; // COMPILE-TIME ERROR as we cant change value once it is final

System.***out***.println("value : " + val);

}

}

**Output :**

**COMPILE-TIME ERROR**

**Example :**

**final method**

**Test.java**

**package** com.test.example;

**public** **class** Test **extends** Car

{

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

{

Car ts = **new** Car();

ts.show();

}

**public** **void** show() // COMPILE-TIME ERROR as method is final

{

System.***out***.println("hello");

}

}

**Car.java**

**package** com.test.example;

**public** **class** Car

{

**public** **final** **void** show()

{

System.***out***.println("hello");

}

}

**Output :**

**COMPILE-TIME ERROR**

**Example :**

**final class**

**Test.java**

**public** **class** Test **extends** Car // COMPILE-TIME ERROR as class is final

{

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

{

Car ts = **new** Car();

ts.show();

}

}

**Car.java**

**public** **final** **class** Car

{

**public** **void** show()

{

System.***out***.println("hello");

}

}

**Output :**

**COMPILE-TIME ERROR**

**Questions :**

1. Can we declare **final** variable as **static** ?

* **YES**

But you have create static block for it to access.

e.g.

**static** **final** **int** ***data***;// static blank final variable

**static**

{

***data*** = 50;

}

1. Can we create **constructor** **final**?

* **NO.**

Because constructor **never** **inherited**

**wrapper class**

- Till now we used premitive data types like int, byte, long, ect.

- But as java is **object oriented programming language** sometimes we need data types as **objects**

- So wrapper classes provides that functionality

- Integer, Byte, Character, Short, Long, Float, Double, Boolean are all **wrapper classes**

- These classes are **final** internally like String class so helps in **not modifying data**

- **Collection framework** work only on wrapper classes

- In **multithreading** we need wrapper classes to support synchronization

**Example : (common example)**

**Test.java**

**public** **class** Test

{

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

{

String str = "20";

**int** a = Integer.*parseInt*(str); // boxing

System.***out***.println("boxing : " + a);

**int** num = 30;

Integer val = Integer.*valueOf*(num); // boxing

System.***out***.println("boxing : " + val);

Integer num1 = 10;

**int** val1 = num1.intValue();

System.***out***.println("unboxing : " + val1);

}

}

**Output :**

boxing : 20

boxing : 30

unboxing : 10

- convert **premitive type** to **object**

**boxing**

- i.e. int to Integer

**Example :**

**Test.java**

**public** **class** Test

{

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

{

**int** num = 20;

Integer obj = **new** Integer(num);

System.***out***.println("boxing : " + obj);

}

}

**Output :**

boxing : 20

**Auto-boxing**

- **automatically** convert **premitive type** to **object**

- i.e. int to Integer

**Example :**

**Test.java**

**public** **class** Test

{

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

{

**int** num = 20;

Integer obj = num;

System.***out***.println("auto-boxing : " + obj);

}

}

**Output :**

auto-boxing : 20

- convert **object** to **premitive type**

**unboxing**

- i.e. Integer to int

**Example :**

**Test.java**

**public** **class** Test

{

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

{

Integer obj = **new** Integer(8);

**int** num = obj.intValue();

System.***out***.println(unboxing : " + num);

}

}

**Output :**

unboxing : 8

**Auto-unboxing**

- **automatically** convert **object** to **premitive type**

- i.e. Integer to int

**Example :**

**Test.java**

**public** **class** Test

{

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

{

Integer obj = **new** Integer(8);

**int** num = obj;

System.***out***.println(unboxing : " + num);

}

}

**Output :**

auto-unboxing : 8

**Data type conversion**

**Example :**

**Test.java**

**public** **class** Test

{

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

{

String value = "12";

**int** a = Integer.*parseInt*(value);

**int** a1 = Integer.*valueOf*(value);

System.***out***.println("INTEGER : " + a + " ------ " + a1);

**long** b = Long.*parseLong*(value);

**long** b1 = Long.*valueOf*(value);

System.***out***.println("LONG : " + b + " ------ " + b1);

**short** c = Short.*parseShort*(value);

**short** c1 = Short.*valueOf*(value);

System.***out***.println("SHORT : " + c + " ------ " + c1);

**byte** d = Byte.*parseByte*(value);

**byte** d1 = Byte.*valueOf*(value);

System.***out***.println("BYTE : " + d + " ------ " + d1);

**double** e = Double.*parseDouble*(value);

**double** e1 = Double.*valueOf*(value);

System.***out***.println("DOUBLE : " + e + " ------ " + e1);

**float** h = Float.*parseFloat*(value);

**float** h1 = Float.*valueOf*(value);

System.***out***.println("FLOAT : " + h + " ------ " + h1);

**boolean** g = Boolean.*parseBoolean*(value);

**boolean** g1 = Boolean.*valueOf*(value);

System.***out***.println("BOOLEAN : " + g + " ------ " + g1);

**for** (**int** i = 0; i < value.length(); i++)

{

**char** f = value.charAt(i);

System.***out***.println("char at " + i + " index is: " + f);

}

}

}

**Output :**

INTEGER : 12 ------ 12

LONG : 12 ------ 12

SHORT : 12 ------ 12

BYTE : 12 ------ 12

DOUBLE : 12.0 ------ 12.0

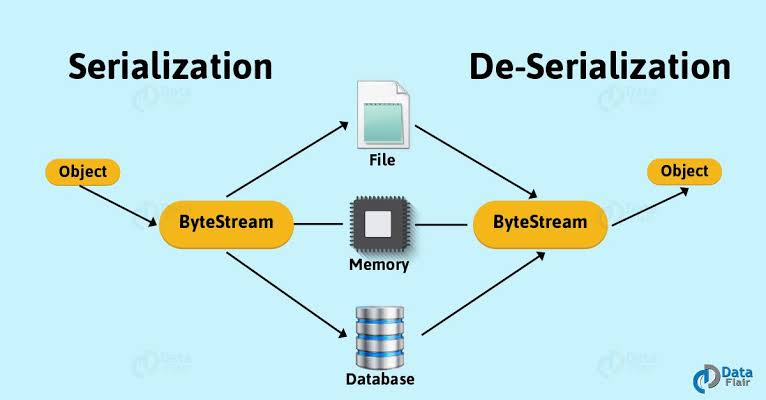
FLOAT : 12.0 ------ 12.0

BOOLEAN : true ------ false

char at 0 index is: 1

char at 1 index is: 2

**Serialization and De-serialization**



**Serialization :**

**- It is a mechanism of converting the state of object into a byte stream**

**De-Serialization :**

**- It is a mechanism of converting byte stream into the object**

**Real world example :**



The **mobile game in which two players play same game** on different mobiles.

When **player1 moves his move** the **player2 can see same on his screen.**

This is because move of player1 gets **converted to bytestream and transferred over network** quickly.

**Syntax :**

**public** **class** clasName **implements** Serializable

**Example :**

**serialization**

**Test.java**

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.ObjectOutputStream;

**public** **class** Test

{

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

{

**try**

{

// Creating the object

Student s1 = **new** Student(211, "ravi");

// Creating stream and writing the object

FileOutputStream fout = **new** FileOutputStream("D:\\MyObj.txt");

ObjectOutputStream out = **new** ObjectOutputStream(fout);

//converting to byte stream

out.writeObject(s1);

out.flush();

out.close();

} **catch** (Exception e)

{

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

}

}

}

**Student.java**

**import** java.io.Serializable;

**public** **class** Student **implements** Serializable

{

**int** id;

String name;

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

{

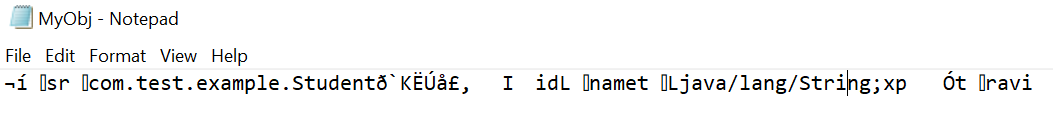
**this**.id = id;

**this**.name = name;

}

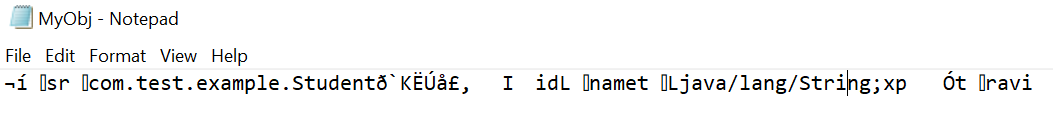
}

**Output :**



**Input :**

**De-serialization**



**Example :**

**Test.java**

**import** java.io.FileInputStream;

**import** java.io.IOException;

**import** java.io.ObjectInputStream;

**public** **class** Test

{

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

{

**try**

{

// Creating stream to read the object

ObjectInputStream in = **new** ObjectInputStream(**new** FileInputStream("D:\\MyObj.txt"));

Student s = (Student) in.readObject();

// printing the data of the serialized object

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

// closing the stream

in.close();

} **catch** (Exception e)

{

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

}

}

}

**Student.java**

**import** java.io.Serializable;

**public** **class** Student **implements** Serializable

{

**int** id;

String name;

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

{

**this**.id = id;

**this**.name = name;

}

}

**Output :**

211 ravi

**- If we use transient keyword with specific data variable then it will not be serialized**

**transient keyword**

**Example : (serialization)**

**Test.java**

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.ObjectOutputStream;

**public** **class** Test

{

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

{

**try**

{

// Creating the object

Student s1 = **new** Student(211, "ravi");

// Creating stream and writing the object

FileOutputStream fout = **new** FileOutputStream("D:\\MyObj.txt");

ObjectOutputStream out = **new** ObjectOutputStream(fout);

//converting to byte stream

out.writeObject(s1);

out.flush();

out.close();

} **catch** (Exception e)

{

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

}

}

}

**Student.java**

**import** java.io.Serializable;

**public** **class** Student **implements** Serializable

{

**transient int** id;

String name;

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

{

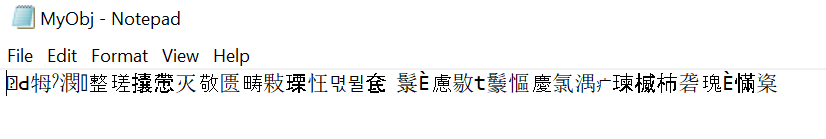
**this**.id = id;

**this**.name = name;

}

}

**Output :**



**Example : (De-serialization)**

**Test.java**

**import** java.io.FileInputStream;

**import** java.io.IOException;

**import** java.io.ObjectInputStream;

**public** **class** Test

{

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

{

**try**

{

// Creating stream to read the object

ObjectInputStream in = **new** ObjectInputStream(**new** FileInputStream("D:\\MyObj.txt"));

Student s = (Student) in.readObject();

// printing the data of the serialized object

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

// closing the stream

in.close();

} **catch** (Exception e)

{

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

}

}

}

**Student.java**

**import** java.io.Serializable;

**public** **class** Student **implements** Serializable

{

**transient int** id;

String name;

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

{

**this**.id = id;

**this**.name = name;

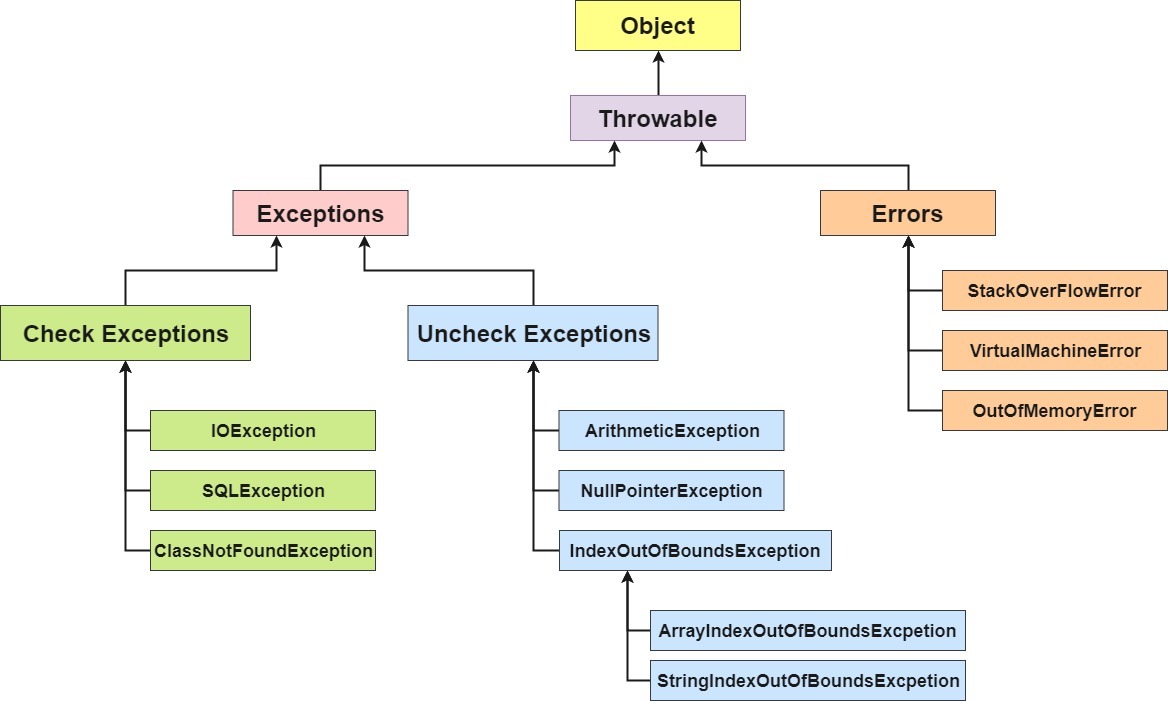
}

}

**Output :**

**0** ravi

**Exception handling**



**Exception handling is the process to handle the runtime errors so that normal flow of the application can be maintained.**

- throws keyword Mostly used to specify **checked exceptions** (compile-time exceptions) but can also used to specify Un-Checked exceptions

**throws**

**Example :**

**Test.java**

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**public** **class** Test

{

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

{

FileInputStream file = **new** FileInputStream("D:\\MyObj.txt");

System.***out***.println("file loaded...");

}

}

**Output :**

file loaded...

**throw**

- **user defined** exception

**Example :**

**Test.java**

**public** **class** Test

{

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

{

*validate*(13);

}

**static** **void** validate(**int** age)

{

**if** (age < 18)

{

**throw** **new** ArithmeticException("age is not valid for voting");

} **else**

{

System.***out***.println("welcome to vote");

}

}

}

**Output :**

Exception in thread "main" java.lang.ArithmeticException: **age is not valid for voting**

at com.test.example.Test.validate(Test.java:14)

at com.test.example.Test.main(Test.java:7)

- try-catch used to specify **Un-checked exceptions** (run-time exceptions) but can also used to specify Checked exceptions

**try-catch block**

**Syntax :**

**try**

**{**

**//code**

**}**

**catch( ExceptionClassName ref )**

**{**

**//print**

**}**

**Example : (without try catch)**

**Test.java**

**public** **class** Test

{

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

{

**int** data = 50 / 0; // may throw exception

System.***out***.println("rest of the code");

}

}

**Output :**

Exception in thread "main" java.lang.ArithmeticException: / by zero

at com.test.example.Test.main(Test.java:7)

**Example : (using try catch)**

**Test.java**

**public** **class** Test

{

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

{

**try**

{

**int** data = 50 / 0; // may throw exception

} **catch** (ArithmeticException e) // handling the exception

{

System.***out***.println("exception is catched : " + e);

}

System.***out***.println("hello");

}

}

**Output :**

exception is catched : java.lang.ArithmeticException: / by zero

hello

**Example : (multi try catch)**

**Test.java**

**public** **class** Test

{

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

{

**try**

{

**int** a[] = **new** **int**[5];

a[5] = 30 / 0;

} **catch** (ArrayIndexOutOfBoundsException e)

{

System.***out***.println("execption is : " + e);

} **catch** (ArithmeticException e)

{

System.***out***.println("execption is : " + e);

} **catch** (Exception e) // general exception must written in last

{

System.***out***.println("common task completed : " + e);

}

System.***out***.println("rest of the code...");

}

}

**Output :**

execption is : java.lang.ArithmeticException: / by zero

rest of the code...

**Example : (nested try catch)**

**Test.java**

**public** **class** Test

{

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

{

**try**

{

**try**

{

System.***out***.println("going to divide");

**int** b = 39 / 0;

} **catch** (ArithmeticException e)

{

System.***out***.println("exception in 2nd try : " + e);

}

**try**

{

**int** a[] = **new** **int**[5];

a[5] = 4;

} **catch** (ArrayIndexOutOfBoundsException e)

{

System.***out***.println("exception in 3rd try : " + e);

}

System.***out***.println("other statement");

} **catch** (Exception e)

{

System.***out***.println("exception in 1st try : " + e);

}

System.***out***.println("normal flow..");

}

}

**Output :**

going to divide

exception in 2nd try : java.lang.ArithmeticException: / by zero

exception in 3rd try : java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5

other statement

normal flow..

**- No matter try block gives an exception or not finally block will always executes**

**finally block**

**- We can write it after try-catch block or just with try block**

**Example :**

**Test.java**

**public** **class** Test

{

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

{

**try**

{

**int** data = 25 / 0;

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

} **catch** (ArithmeticException e)

{

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

} **finally**

{

System.***out***.println("finally block is always executed");

}

System.***out***.println("rest of the code...");

}

}

**Output :**

java.lang.ArithmeticException: / by zero

**finally block is always executed**

rest of the code...

**Custom exception handling**

**Example : (without parameter)**

**Test.java**

**public** **class** Test

{

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

{

**try**

{

**throw** **new** SampleException();

}

**catch**(SampleException e)

{

System.***out***.println("exception caught..."+e.getMessage());

}

}

}

**SampleException.java**

**public** **class** SampleException **extends** Exception

{

**public** SampleException()

{

}

}

**Output :**

exception caught...**null**

**Example : (with parameter)**

**Test.java**

**public** **class** Test

{

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

{

**try**

{

**throw** **new** SampleException("Hello Its my own exception...");

}

**catch**(SampleException e)

{

System.***out***.println("exception caught..."+e.getMessage());

}

}

}

**SampleException.java**

**public** **class** SampleException **extends** Exception

{

**public** SampleException(String str)

{

**super**(str);

}

}

**Output :**

exception caught...**Hello Its my own exception...**

**Rule 1 : If super class don’t declares an exception**

**Method overriding in exception handling**

**Case 1: If super class don’t declaes an exception and child class can have**

**runtime exception**

**Example : (It will get executed)**

**Test.java**

**public** **class** Test **extends** Parent

{

**void** msg() **throws** ArithmeticException //Un-checked exception

{

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

}

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

{

Parent p = **new** Test();

p.msg();

}

}

**Parent.java**

**public** **class** Parent

{

**void** msg()

{

System.***out***.println("parent");

}

}

**Output :**

Child

**Rule 2 : If super class declares an exception**

**Case 1 : If parent class has specific exception** “ArithmeticException”

**then child class can have same exceptions** “ArithmeticException”

**Example :**

**Test.java**

public class Test extends Parent

{

void msg() throws ArithmeticException // same exception as in superclass

{

System.*out*.println("child");

}

public static void main(String args[])

{

Parent p = new Test();

p.msg();

}

}

**Parent.java**

**public** **class** Parent

{

**void** msg() **throws** ArithmeticException // same exception as in subclass

{

System.***out***.println("parent");

}

}

**Output :**

Child

**Case 2 : If parent class has child exception “**ArithmeticException”

**then child class can not have parent exception** “Exception”

**Example :**

**Test.java**

**public** **class** Test **extends** Parent

{

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

{

Parent p = **new** Test();

p.msg();

}

**void** msg() **throws** Exception //compile time error

{

System.***out***.println("child");

}

}

**Parent.java**

**public** **class** Parent

{

**void** msg() **throws** ArithmeticException

{

System.***out***.println("parent");

}

}

**Output :**

compile time error

**Case 3 : If parent class has exception “**Exception**” and child class can have child exception** “ArithmeticException”

**Example :**

**Test.java**

**public** **class** Test **extends** Parent

{

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

{

Parent p = **new** Test();

p.msg();

}

**void** msg() **throws** ArithmeticException

{

System.***out***.println("child");

}

}

**Parent.java**

**public** **class** Parent

{

**void** msg() **throws** Exception

{

System.***out***.println("parent");

}

}

**Output :**

child

**Case 4 : If parent class has some exception** “Exception” **and child class can have no**

**exception**

**Example :**

**Test.java**

**public** **class** Test **extends** Parent

{

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

{

Parent p = **new** Test();

p.msg();

}

**void** msg() // no exception

{

System.***out***.println("child");

}

}

**Parent.java**

**public** **class** Parent

{

**void** msg() **throws** Exception

{

System.***out***.println("parent");

}

}

**Output :**

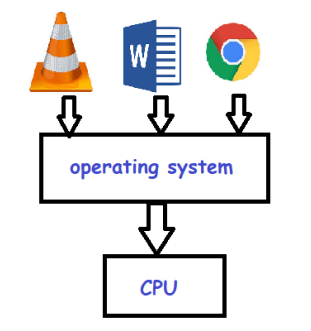
child

**multi tasking**

**multithreading**

- Performing **multiple tasks** at single time

- E.g.



- Here **VLC, Word doc, Browser** running at same time

- It happens because CPU switches tasks one by one

- It happens so fast that we cant see by our eyes

- So we think they all are executing simultaneously

- It inceases CPU performance

- It can be achieved through,

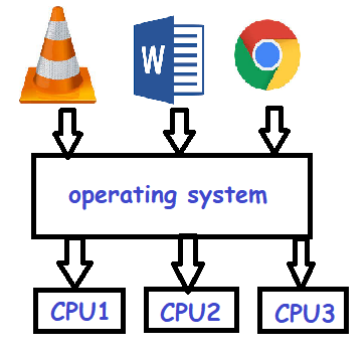
**1) Multi-processing**

**2) Multi-threading**

- When **single system** is connected to **multiple CPU’s** (processors) to complete task

**multi processing**

- E.g.

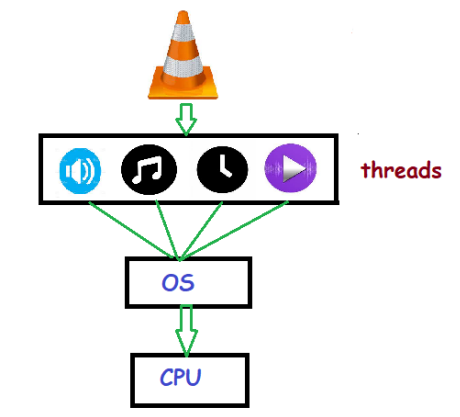


**multi threading**

- E.g. **VLC** has **multiple tasks** like **volume, sidebar, audio, video, timer**, etc

- This tasks are called as **threads**

- This **multiple threads** **executes** at **same time**



- Multithreading used in games,animatins,softwares like VLC

**Example : (without multithreading)**

**VLC.java**

**public** **class** VLC

{

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

{

Video vid = **new** Video();

vid.playVideo();

Audio aud = **new** Audio();

aud.playAudio();

}

}

**Video.java**

**public** **class** Video

{

**public** **void** playVideo()

{

System.***out***.println("video is playing...");

}

}

**Audio.java**

**public** **class** Audio

{

**public** **void** playAudio()

{

System.***out***.println("audio is playing...");

}

}

**Output :**

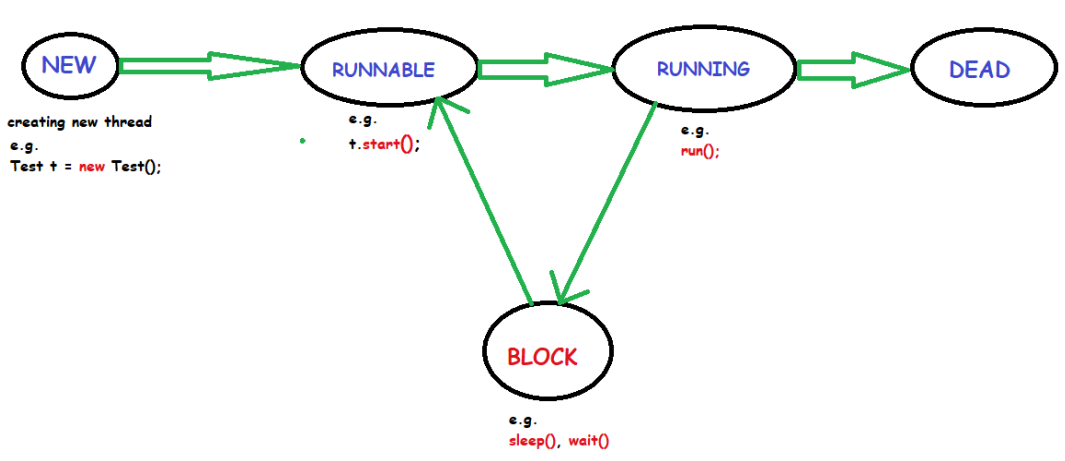
video is playing...

audio is playing...

**Explanation :**

- without multithreading video and audio are **running separately**

- means running **one by one** and **not at same time**



**thread life cycle**

**1) New :**

**Thread is created** using **Thread class**.

But thread is not started yet

**2) Runnable :**

**Thread started** using **start()** method

**3) Running :**

**Thread is in running** state using **run()** method

**4) Blocked (Not Runnable) / waiting :**

**Thread is alive but not running** Its in waiting state due to **sleep(), wait()** methods called

**5) Terminated :**

**Thread is terminated**/dead

**We can create threads in two ways,**

**create threads**

**1) extending Thread class : (it implements Runnable interface internally)**

**Example :**

**Example.java**

**public** **class** Example **extends** Thread

{

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

{

Example e = **new** Example();

e.start();

System.***out***.println("THREAD IS STARTED...");

}

**public** **void** run()

{

System.***out***.println("THREAD IS RUNNING...");

}

}

**Output :**

THREAD IS STARTED...

THREAD IS RUNNING...

**2) implementing Runnable interface :**

**Example :**

**Example.java**

**public** **class** Example **implements** Runnable

{

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

{

Example e = **new** Example();

Thread t1 = **new** Thread(e);

t1.start();

System.***out***.println("THREAD IS STARTED...");

}

**public** **void** run()

{

System.***out***.println("THREAD IS RUNNING...");

}

}

**Output :**

THREAD IS STARTED...

THREAD IS RUNNING...

**Question :**

**1) which way is better extending thread or implements Runnable ?**

* **implements Runnable is better**

**reason behind it is that java does not supports multiple inheritance.**

**e.g.**

**public** **class** A **extends** B, Thread //compiletime exception

{

//code

}

if we use extends Thread as above it is **not possible to extend class B** also at same time.

So we have below thing,

**public** **class** A **extends** B **implements** Runnable

{

//code

}

Now we **can extend class B** also using **implements Runnable.**

- It is **not possible**

**call start() twice**

**Example :**

**Example.java**

**public** **class** Example **extends** Thread

{

**public** **void** run()

{

System.***out***.println("running...");

}

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

{

Example e = **new** Example();

e.start();

e.start(); // Run-time exception

}

}

**Output :**

Exception in thread "main" java.lang.IllegalThreadStateException

running...

at java.base/java.lang.Thread.start(Thread.java:793)

at Example.main(Example.java:12)

- It **is possible** but you will **not get expected result** as multhreading gives

**call run() twice**

**Example :**

**Example.java**

**public** **class** Example **extends** Thread

{

**public** **void** run()

{

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

{

**try**

{

Thread.*sleep*(500);

} **catch** (InterruptedException e)

{

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

}

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

}

}

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

{

Example e1 = **new** Example();

Example e2 = **new** Example();

e1.run(); // calling run() without start()

e2.run(); // calling run() without start()

}

}

**Output : (multithreded is not happening here)**

1

2

3

4

1

2

3

4

**Expected Output is : (multithreded output)**

1

1

2

2

3

3

4

4

**Example : (single task by single thread)**

**multi tasking**

**Example.java**

**public** **class** Example **extends** Thread

{

**public** **void** run()

{

System.***out***.println("running...");

}

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

{

Example t = **new** Example();

t.start();

}

}

**Output :**

running...

**Example : (single task by multiple thread)**

**Example.java**

**public** **class** Example **extends** Thread

{

**public** **void** run()

{

System.***out***.println("running...");

}

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

{

Example t1 = **new** Example();

t1.start();

Example t2 = **new** Example();

t2.start();

}

}

**Output :**

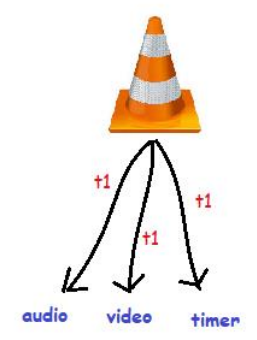
running...

running...

**Example : (multiple task by single thread)**

**Not possible,**

Suppose in VLC player we are trying to play audio, video, timer such **multiple** **tasks** by **single thread t1**



**Example : (multiple tasks by multiple threads)**

**Example.java**

**public** **class** Example

{

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

{

A a = **new** A();

a.start();

B b = **new** B();

b.start();

}

}

**A.java**

**public** **class** A **extends** Thread

{

**public** **void** run()

{

System.***out***.println("class A");

}

}

**B.java**

**public** **class** B **extends** Thread

{

**public** **void** run()

{

System.***out***.println("class B");

}

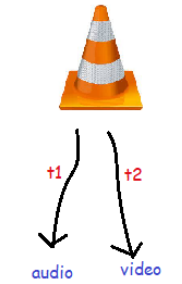
}

**Output :**

class A

class B

**Explanation : (Thread t1 and t2 running at same time)**



- Thread **sleeps for specific time** and then starts execution again

**sleep() method**

**Example 1 : (main thread)**

**Example.java**

**public** **class** Example

{

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

{

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

{

**try**

{

Thread.*sleep*(500); // main Thread sleeps for 500 ms

} **catch** (InterruptedException e)

{

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

}

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

}

}

}

**Output :**

1

//Sleep for 500 ms

2

//Sleep for 500 ms

3

**Example 2 : (user thread)**

**Example.java**

**public** **class** Example **extends** Thread

{

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

{

Example t1 = **new** Example();

t1.start();

}

**public** **void** run()

{

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

{

**try**

{

Thread.*sleep*(500); // Thread sleeps for 500 ms

} **catch** (InterruptedException e)

{

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

}

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

}

}

}

**Output :**

1

//Sleep for 500 ms

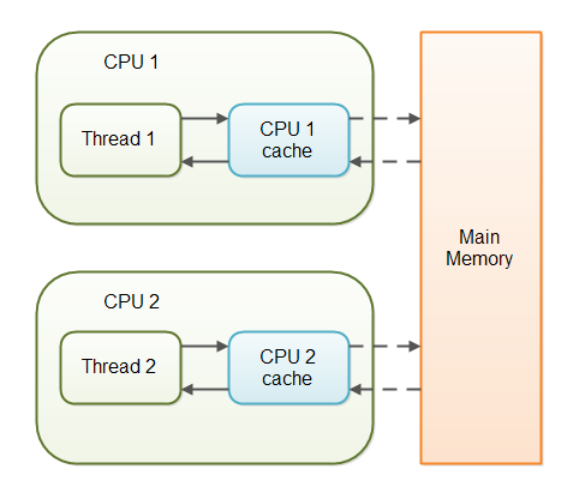
2

//Sleep for 500 ms

3

- as shown in below image **one computor** may have **2 cpu**

**volatile keyword**



**public** **class** Test

{

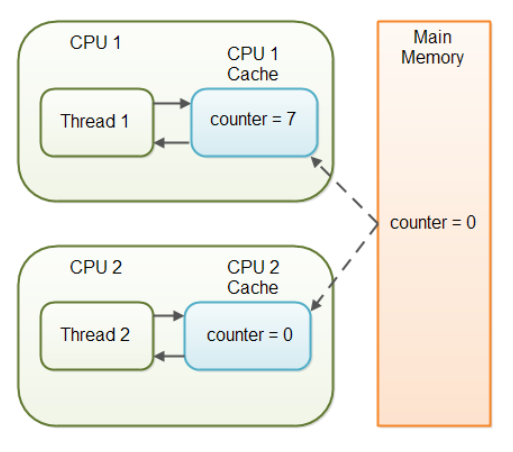
**public** **int** counter = 0;

}

- as shown in below image,

- when **thread1** updates **value of counter as 7** it gets stored in **cache** as **7** but **not yet in main memory**

- at same time **thread2** tries to read value of that variable from **main memory** but as the value is not updated yet so thread2 get **value as 0** only



- It may create problem in whole calulations in systems like bank applications

- by using **volatile keyword** data gets stored directly into **main memory**.

- Means **data not get stored in cache** now.

- so this problem can be **resolved**

**public** **class** Test

{

**public** **volatile** **int** counter = 0;

}

**synchronization**

- when **multiple threads** try to access **same resource** on **same time** then we **may get wrong result**

- to stop this we need synchronization

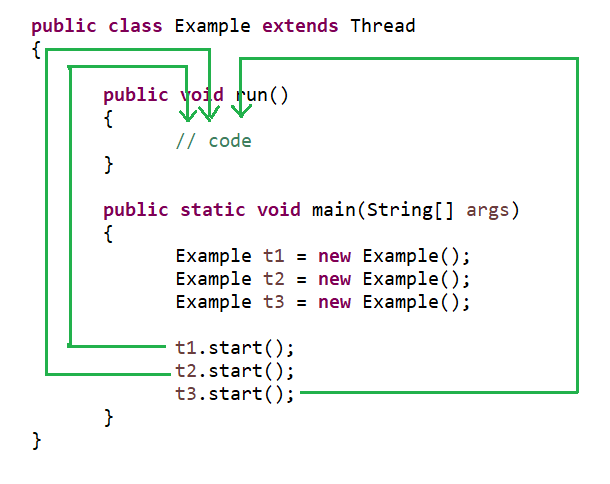
- it helps only single thread to access resource

- it can be achieved using,

**1) method synchronization**

**2) block synchronization**

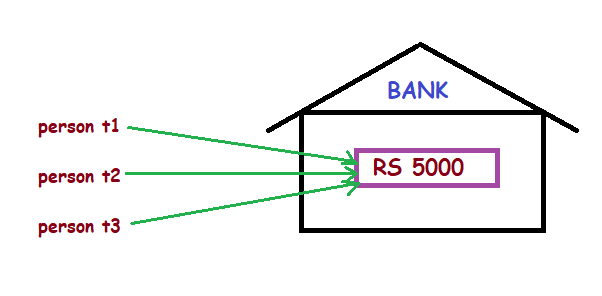
**Example : (problem without synchronization)**



**- here all threads t1, t2, t3 tries to access same resource at same time which may give wrong result**

**Real world problem :**

**method synchronization**



**- Bank has only RS 5000 left today**

**- Person t1, t2, t3 goes to bank and tried to get RS 5000 at same time.**

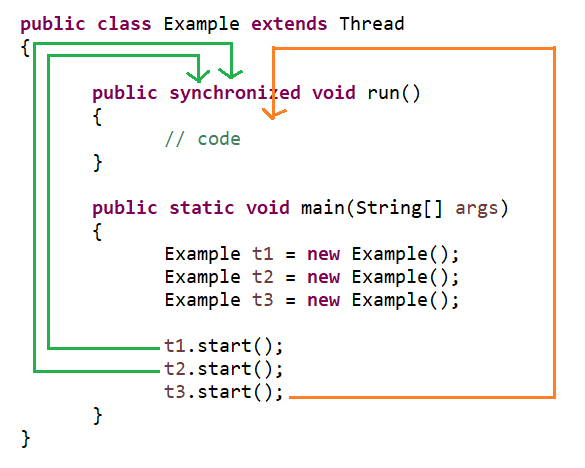
**- Practically this is impossible.**

**- only one person who goes first should get the rs 5000 and others should get message like no sufficient balance**

**- But we may get wrong result like RS 15000 retrived from bank in multithreading program as all three threads t1, t2, t3 will access same resource.**

**- to resolve this problem method synchronization is used where only one thread will have access to resource at one time**

**Example :**



**Example : (without method synchronization)**

**Example.java**

**public** **class** Example

{

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

{

Table obj = **new** Table();

Five t1 = **new** Five(obj);

Two t2 = **new** Two(obj);

t1.start();

t2.start();

}

}

**Table.java**

**public** **class** Table

{

**public** **void** printTable(**int** n)

{

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

{

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

}

}

}

**Two.java**

**public** **class** Two **extends** Thread

{

Table t;

**public** Two(Table t)

{

**this**.t=t;

}

**public** **void** run()

{

t.printTable(2);

}

}

**Five.java**

**public** **class** Five **extends** Thread

{

Table t;

**public** Five(Table t)

{

**this**.t=t;

}

**public** **void** run()

{

t.printTable(5);

}

}

**Output : (output is not as expected)**

**5**

**10**

**15**

**20**

**2**

**25**

**30**

**35**

**40**

**45**

**50**

**4**

**6**

**8**

**10**

**12**

**14**

**16**

**18**

**20**

**Example : (using method synchronization)**

**Example.java**

**public** **class** Example

{

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

{

Table obj = **new** Table();

Five t1 = **new** Five(obj);

Two t2 = **new** Two(obj);

t1.start();

t2.start();

}

}

**Table.java**

**public** **class** Table

{

**public** **synchronized** **void** printTable(**int** n)

{

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

{

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

}

}

}

**Two.java**

**public** **class** Two **extends** Thread

{

Table t;

**public** Two(Table t)

{

**this**.t=t;

}

**public** **void** run()

{

t.printTable(2);

}

}

**Five.java**

**public** **class** Five **extends** Thread

{

Table t;

**public** Five(Table t)

{

**this**.t=t;

}

**public** **void** run()

{

t.printTable(5);

}

}

**Output :**

**5**

**10**

**15**

**20**

**25**

**30**

**35**

**40**

**45**

**50**

**2**

**4**

**6**

**8**

**10**

**12**

**14**

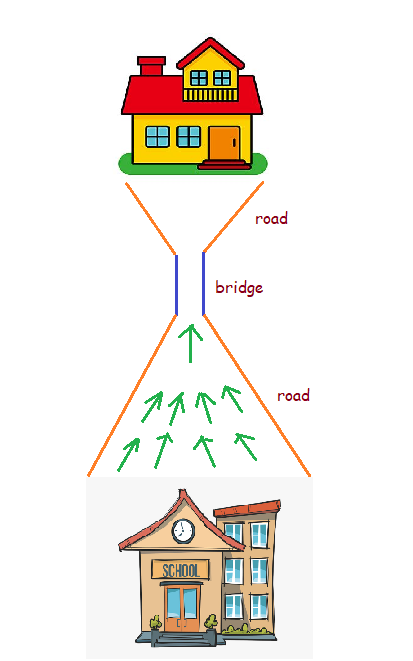
**16**

**18**

**20**

**Real world problem :**

**block synchronization**



- here **many students** walking on road to reach home

- but there is one small **bridge**.

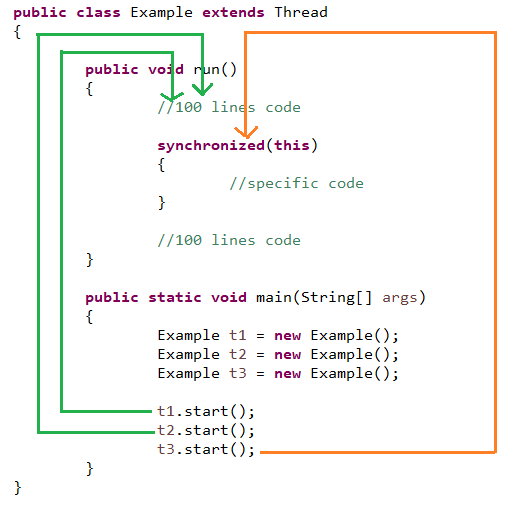
- If many students try to walk on that ridge it may cause **traffic**

- so we allow only **one student** to walt at a time on that bridge

- block synchronization is **fast** over method synchronization

- because if in above example we use method synchronization then only one student is allowed to walk on that entire road so it will take so much time to reach each student to their home

**Example :**



**Example : (using method synchronization)**

**Example.java**

**public** **class** Example

{

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

{

Table obj = **new** Table();

Five t1 = **new** Five(obj);

Two t2 = **new** Two(obj);

t1.start();

t2.start();

}

}

**Table.java**

**public** **class** Table

{

**public** **synchronized** **void** printTable(**int** n)

{

System.***out***.println("100 lines of code");

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

{

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

}

}

}

**Two.java**

**public** **class** Two **extends** Thread

{

Table t;

**public** Two(Table t)

{

**this**.t=t;

}

**public** **void** run()

{

t.printTable(2);

}

}

**Five.java**

**public** **class** Five **extends** Thread

{

Table t;

**public** Five(Table t)

{

**this**.t=t;

}

**public** **void** run()

{

t.printTable(5);

}

}

**Output : (100 lines are printing gets stuck due to method synchronization)**

**100 lines of code**

**2**

**4**

**6**

**8**

**10**

**12**

**14**

**16**

**18**

**20**

**100 lines of code**

**5**

**10**

**15**

**20**

**25**

**30**

**35**

**40**

**45**

**50**

**Example : (using block synchronization)**

**Example.java**

**public** **class** Example

{

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

{

Table obj = **new** Table();

Five t1 = **new** Five(obj);

Two t2 = **new** Two(obj);

t1.start();

t2.start();

}

}

**Table.java**

**public** **class** Table

{

**public** **void** printTable(**int** n)

{

System.***out***.println("100 lines of code");

**synchronized** (**this**)

{

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

{

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

}

}

}

}

**Two.java**

**public** **class** Two **extends** Thread

{

Table t;

**public** Two(Table t)

{

**this**.t=t;

}

**public** **void** run()

{

t.printTable(2);

}

}

**Five.java**

**public** **class** Five **extends** Thread

{

Table t;

**public** Five(Table t)

{

**this**.t=t;

}

**public** **void** run()

{

t.printTable(5);

}

}

**Output : (100 lines are printed easily due to block synchronization)**

**100 lines of code //by thread t1**

**100 lines of code //by thread t2**

**2**

**4**

**6**

**8**

**10**

**12**

**14**

**16**

**18**

**20**

**5**

**10**

**15**

**20**

**25**

**30**

**35**

**40**

**45**

**50**

**- In static synchronization thread is locked as class level**

**static synchronization**

**problem in synchronization :**

**- Till now we worked with single object as below,**

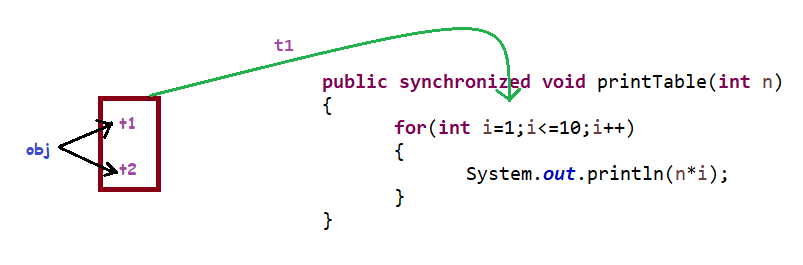
Table obj = **new** Table();

Five t1 = **new** Five(obj);

Two t2 = **new** Two(obj);

t1.start();

t2.start();



as shown above only one tread t1 is accesing **printTable method**

**- what if we have to deal with two object as below,**

Table obj1 = **new** Table();

Five t1 = **new** Five(obj1);

Two t2 = **new** Two(obj1);

Table obj2 = **new** Table();

Five t3 = **new** Five(obj2);

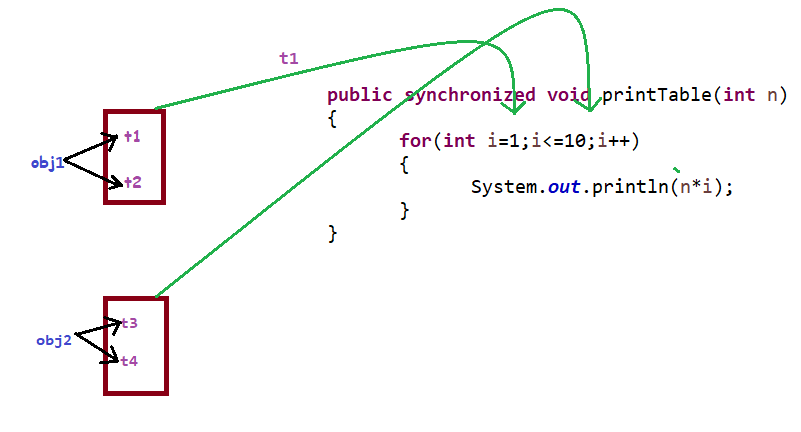
Two t4 = **new** Two(obj2);

t1.start();

t2.start();

t3.start();

t4.start();



as shown above tread t1 and tread t3 are accesing **printTable method** which will give **wrong output** **because synchronization is failing here.**

**Example : (using method synchronization)**

**Example.java**

**public** **class** Example

{

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

{

Table obj1 = **new** Table();

Five t1 = **new** Five(obj1);

Two t2 = **new** Two(obj1);

Table obj2 = **new** Table();

Five t3 = **new** Five(obj2);

Two t4 = **new** Two(obj2);

t1.start();

t2.start();

t3.start();

t4.start();

}

}

**Table.java**

**public** **class** Table

{

**public** **synchronized** **void** printTable(**int** n)

{

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

{

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

}

}

}

**Two.java**

**public** **class** Two **extends** Thread

{

Table t;

**public** Two(Table t)

{

**this**.t=t;

}

**public** **void** run()

{

t.printTable(2);

}

}

**Five.java**

**public** **class** Five **extends** Thread

{

Table t;

**public** Five(Table t)

{

**this**.t=t;

}

**public** **void** run()

{

t.printTable(5);

}

}

**Output : (gives wrong output)**

**5**

**5**

**10**

**10**

**15**

**20**

**15**

**25**

**30**

**35**

**40**

**45**

**50**

**20**

**25**

**30**

**35**

**40**

**2**

**4**

**6**

**8**

**10**

**12**

**14**

**45**

**50**

**16**

**18**

**20**

**2**

**4**

**6**

**8**

**10**

**12**

**14**

**16**

**18**

**20**

**Example : (using static synchronization)**

**Example.java**

**public** **class** Example

{

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

{

Table obj1 = **new** Table();

Five t1 = **new** Five(obj1);

Two t2 = **new** Two(obj1);

Table obj2 = **new** Table();

Five t3 = **new** Five(obj2);

Two t4 = **new** Two(obj2);

t1.start();

t2.start();

t3.start();

t4.start();

}

}

**Table.java**

**public** **class** Table

{

**public static** **synchronized** **void** printTable(**int** n)

{

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

{

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

}

}

}

**Two.java**

**public** **class** Two **extends** Thread

{

Table t;

**public** Two(Table t)

{

**this**.t=t;

}

**public** **void** run()

{

t.printTable(2);

}

}

**Five.java**

**public** **class** Five **extends** Thread

{

Table t;

**public** Five(Table t)

{

**this**.t=t;

}

**public** **void** run()

{

t.printTable(5);

}

}

**Output : (gives wrong output)**

**5**

**10**

**15**

**20**

**25**

**30**

**35**

**40**

**45**

**50**

**2**

**4**

**6**

**8**

**10**

**12**

**14**

**16**

**18**

**20**

**2**

**4**

**6**

**8**

**10**

**12**

**14**

**16**

**18**

**20**

**5**

**10**

**15**

**20**

**25**

**30**

**35**

**40**

**45**

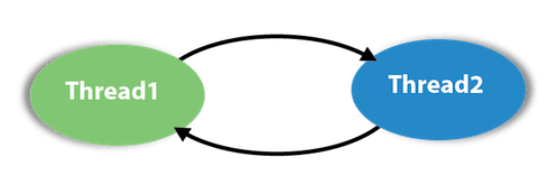
**50**

- Deadlock in Java is a part of multithreading.

**deadlock**

- Deadlock can occur in a situation when a **thread is waiting for an object lock**, that is **acquired by another thread** and **second thread is waiting for an object lock** that is **acquired by first thread**.

- Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



**Example :**

**Example.java**

**public** **class** Example

{

**static** String *s1* = "vithal";

**static** String *s2* = "krishna";

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

{

Demo1 t1 = **new** Demo1(*s1*, *s2*);

Demo2 t2 = **new** Demo2(*s1*, *s2*);

t1.start();

t2.start();

}

}

**Demo1.java**

**public** **class** Demo1 **extends** Thread

{

String lock1;

String lock2;

**public** Demo1(String lock1, String lock2)

{

**this**.lock1=lock1;

**this**.lock2=lock2;

}

**public** **void** run()

{

**synchronized** (lock1)

{

System.***out***.println("Thread 1: Holding lock 1...");

**try**

{

Thread.*sleep*(10);

} **catch** (InterruptedException e)

{

}

System.***out***.println("Thread 1: Waiting for lock 2...");

**synchronized** (lock2)

{

System.***out***.println("Thread 1: Holding lock 1 & 2...");

}

}

}

}

**Demo2.java**

**public** **class** Demo2 **extends** Thread

{

String lock1;

String lock2;

**public** Demo2(String lock1, String lock2)

{

**this**.lock1=lock1;

**this**.lock2=lock2;

}

**public** **void** run()

{

**synchronized** (lock2)

{

System.***out***.println("Thread 2: Holding lock 2...");

**try**

{

Thread.*sleep*(10);

} **catch** (InterruptedException e)

{

}

System.***out***.println("Thread 2: Waiting for lock 1...");

**synchronized** (lock1)

{

System.***out***.println("Thread 2: Holding lock 1 & 2...");

}

}

}

}

**Output : (program is not getting terminated due to deadlock)**

Thread 1: Holding lock 1...

Thread 2: Holding lock 2...

**Thread 1: Waiting for lock 2...**

**Thread 2: Waiting for lock 1...**

**Example : (resolving deadlock by changing order of locks in Demo2.java)**

**Example.java**

**public** **class** Example

{

**static** String *s1* = "vithal";

**static** String *s2* = "krishna";

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

{

Demo1 t1 = **new** Demo1(*s1*, *s2*);

Demo2 t2 = **new** Demo2(*s1*, *s2*);

t1.start();

t2.start();

}

}

**Demo1.java**

**public** **class** Demo1 **extends** Thread

{

String lock1;

String lock2;

**public** Demo1(String lock1, String lock2)

{

**this**.lock1=lock1;

**this**.lock2=lock2;

}

**public** **void** run()

{

**synchronized** (lock1)

{

System.***out***.println("Thread 1: Holding lock 1...");

**try**

{

Thread.*sleep*(10);

} **catch** (InterruptedException e)

{

}

System.***out***.println("Thread 1: Waiting for lock 2...");

**synchronized** (lock2)

{

System.***out***.println("Thread 1: Holding lock 1 & 2...");

}

}

}

}

**Demo2.java**

**public** **class** Demo2 **extends** Thread

{

String lock1;

String lock2;

**public** Demo2(String lock1, String lock2)

{

**this**.lock1=lock1;

**this**.lock2=lock2;

}

**public** **void** run()

{

**synchronized** (lock1)

{

System.***out***.println("Thread 2: Holding lock 2...");

**try**

{

Thread.*sleep*(10);

} **catch** (InterruptedException e)

{

}

System.***out***.println("Thread 2: Waiting for lock 1...");

**synchronized** (lock2)

{

System.***out***.println("Thread 2: Holding lock 1 & 2...");

}

}

}

}

**Output : (now program terminated successfully)**

Thread 1: Holding lock 1...

**Thread 1: Waiting for lock 2...**

**Thread 1: Holding lock 1 & 2...**

Thread 2: Holding lock 2...

**Thread 2: Waiting for lock 1...**

**Thread 2: Holding lock 1 & 2...**

**Inter thread communication**

**1) wait() : helps in releasing lock of current resource**

**2) notify() : resumes current thread which is in waiting state**

**3) notifyAll() : resumes all threads which are in waiting state**

**Real world example :**

**1) person has RS 500 in his bank account**

**2) he go to atm and tried to get rs 1000 from bank but in this case he will get error like no sufficient balance**

**3) in this case he should call wait() current retrive cash method**

**4) and execute another method which will deposit some money in his account**

**5) then he will call notify() method which will resume retrive method again aso he can get rs 1000 in his hand**

**Read input from console**

**scanner**

**Example :**

**Test.java**

**import** java.util.Scanner;

**public** **class** Test

{

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

{

Scanner sc = **new** Scanner(System.***in***);

System.***out***.println("Enter a String : ");

String b = sc.nextLine();

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

System.***out***.println("---------------------------");

System.***out***.println("Enter an integer : ");

**int** a = sc.nextInt();

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

}

}

**Output :**

Enter a String :

bruce lee

String is : bruce lee

---------------------------

Enter an integer :

12 78

Integer is : 12

**Example :**

**BufferedReader**

**Test.java**

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**public** **class** Test

{

**public** **static** **void** main(String args[]) **throws** NumberFormatException, IOException

{

BufferedReader br = **new** BufferedReader(**new** InputStreamReader(System.***in***));

System.***out***.println("Enter a String : ");

String b = br.readLine();

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

System.***out***.println("---------------------------");

System.***out***.println("Enter an integer : ");

**int** a = Integer.*parseInt*(br.readLine());

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

}

}

**Output :**

Enter a String :

bruce lee

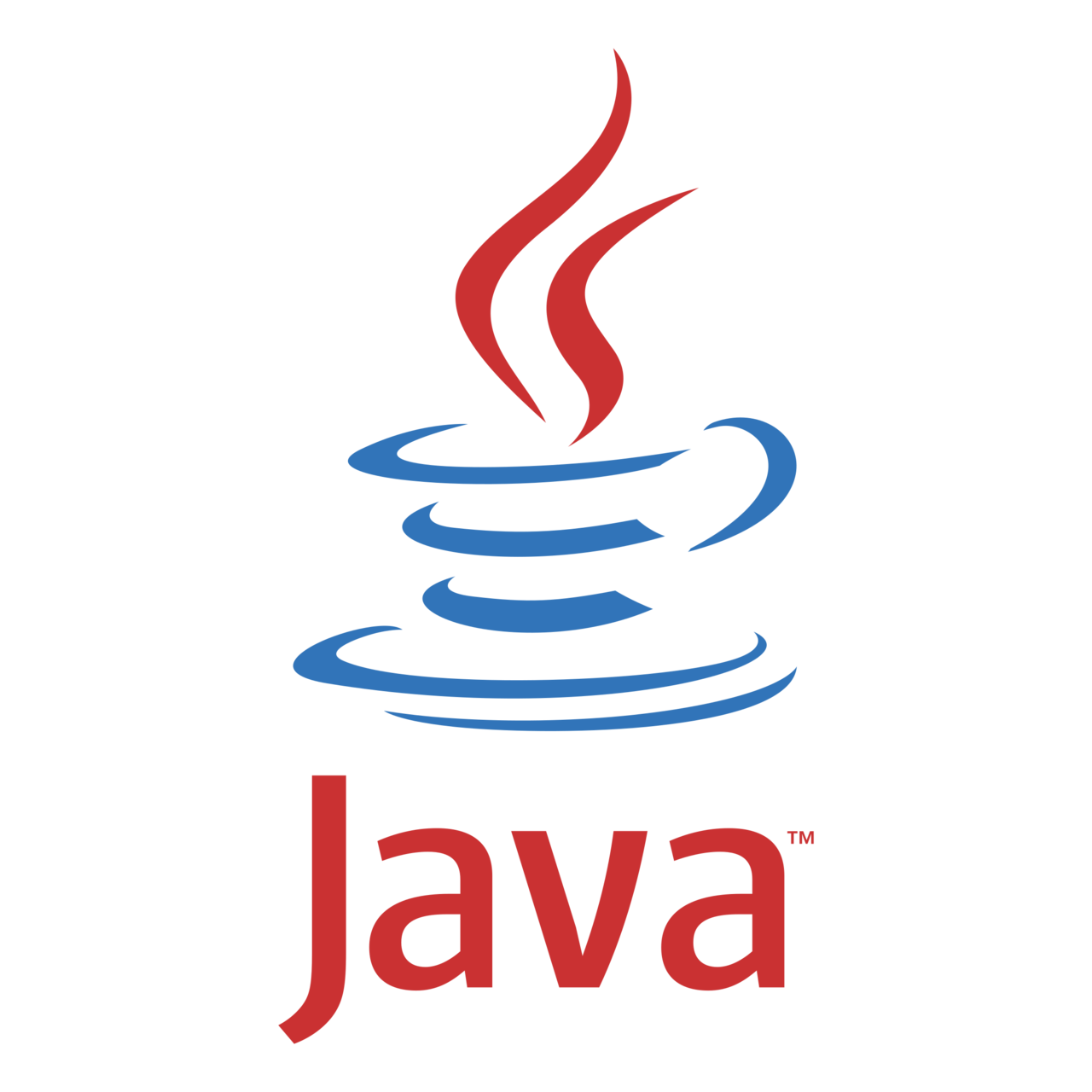
String is : bruce lee

---------------------------

Enter an integer :

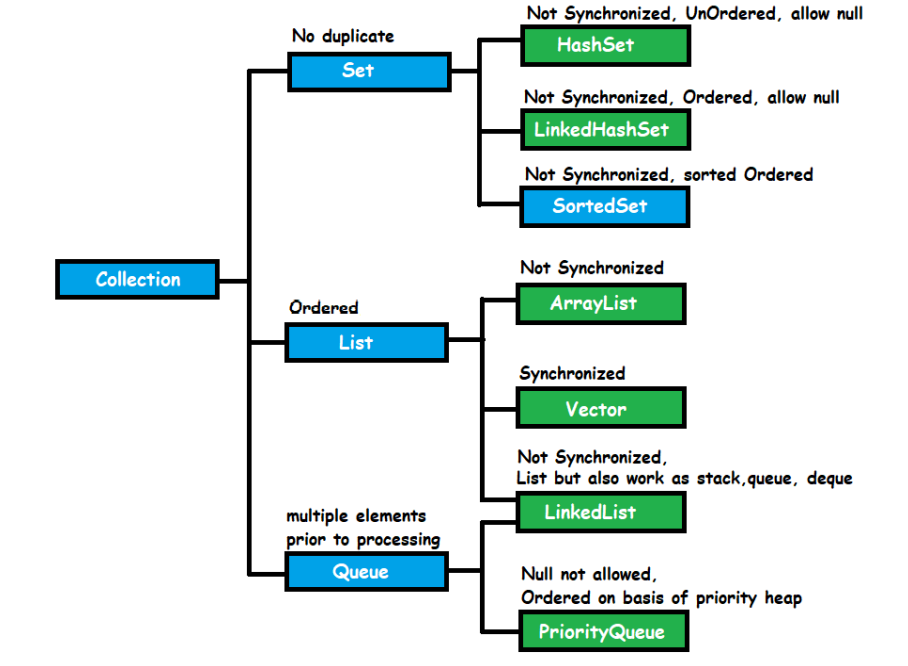
12

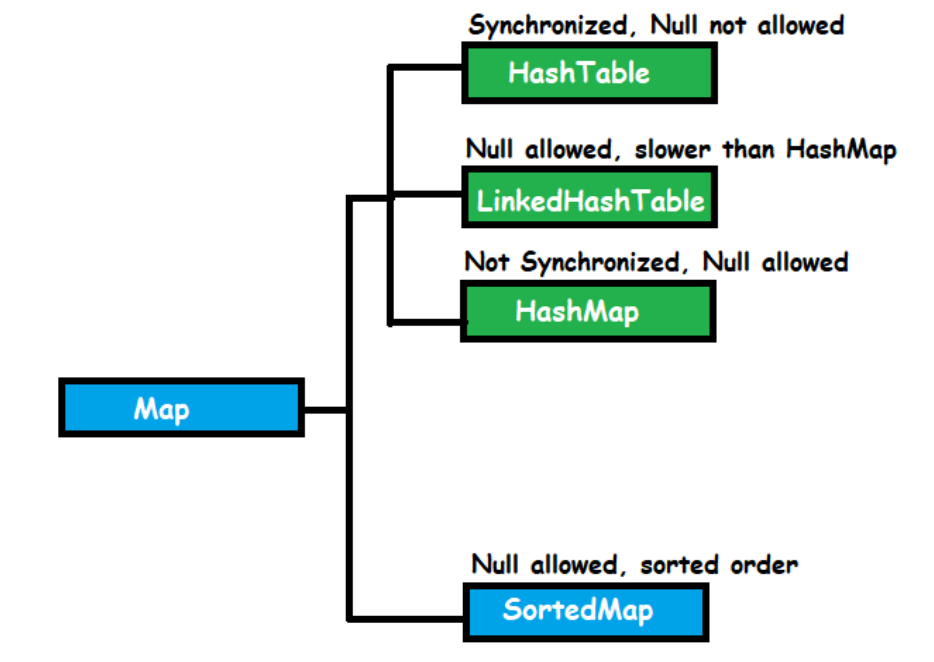
Integer is : 12



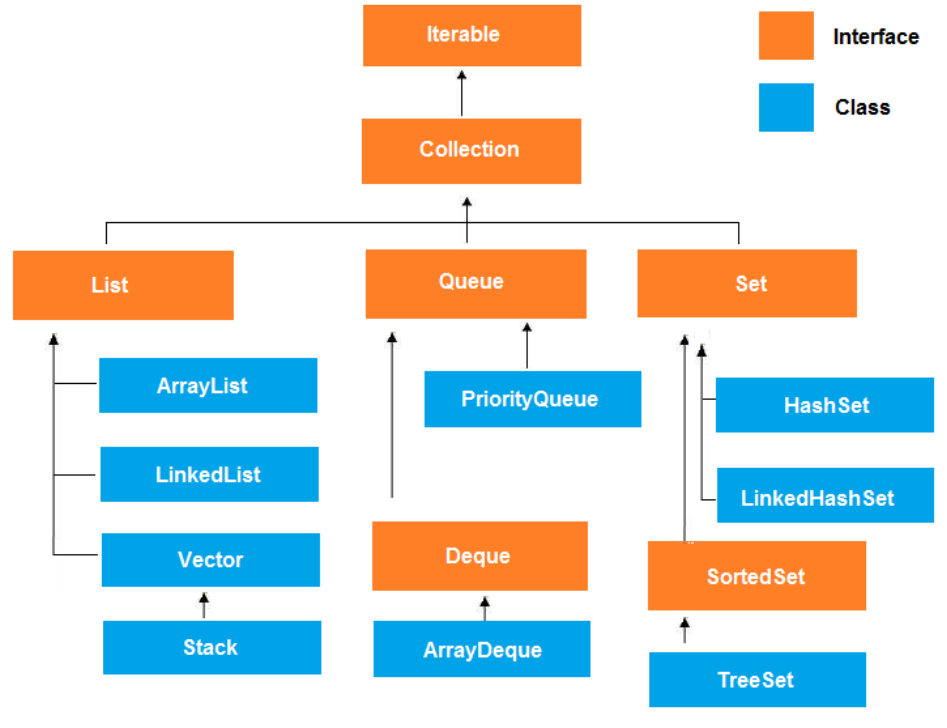
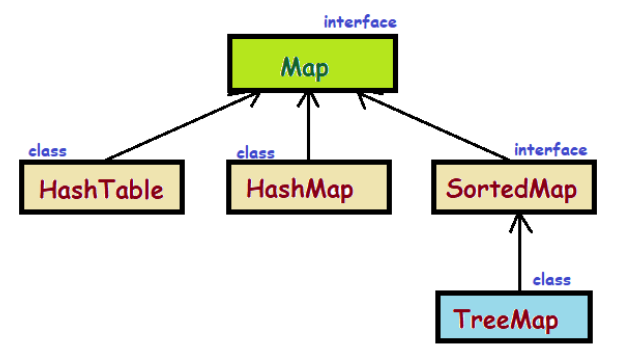
**Collection**

**collection framework**





**Hierarchy of collection and map**

**Question :**

**1) Why Map is not part of Collection framework ?**

➔ Map is a part of **Collection framework** but **not implements** **collection interface.**

Because Map stores data as **key** and **value** pairs.

**2) List vs Set ?**

➔

|  |  |
| --- | --- |
| **List** | **Set** |
| **It has index** | **No index** |
| **Order of elements preserved** | **Order of elements not preserved** |
| **Duplicates allowed** | **Duplicates not allowed** |

**3) What is** **package** **name of collection framework ?**

➔ **java.util.Collection**

**java.util.Map**

**Internal working**

**ArrayList**

**1) Default size :**

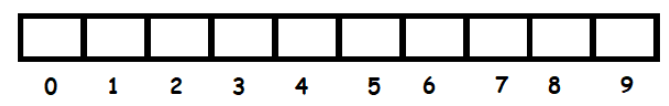
**- ArrayList default size is 0 at below line,**

**ArrayList list=new ArrayList();**

**- when we add element its size becomes is 10**

**list.add("hitesh");**

**- It creates array of size 10 internally as below,**



**Example :**

**Test.java**

**import** java.util.ArrayList;

**public** **class** Test

{

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

{

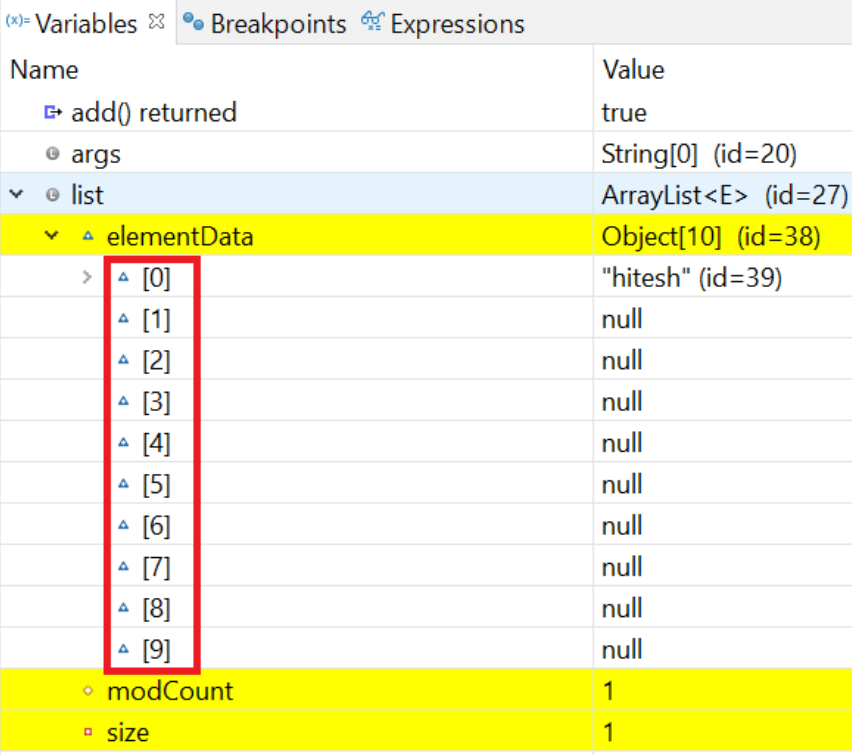
ArrayList list = **new** ArrayList();

list.add("hitesh");

}

}

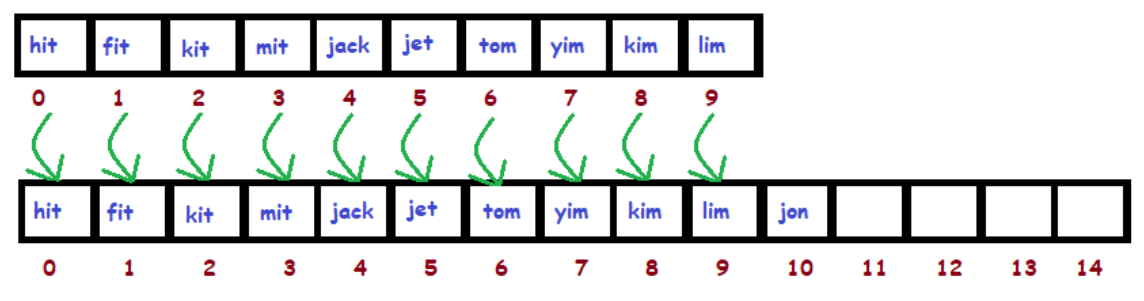
**Debug mode :**



**2) size after 10th element :**

**- when array size gets full till 10**

**- then it increases array size dynamically by 50%**



**- as shown old array elements gets copied to new array**

**- then old array is eligible for garbage collection**

**Question :**

**1) why insert / remove elements from ArrayList is time consuming?**

➔ Because when we add/ remove any element from arraylist **Shifting of elements** is done

So if we there are 1000 elements present in arraylist and we do add/remove element then need to shift those 1000 elements Which **takes more time**

**2*)* why read elements from ArrayList is faster?**

➔ Because ArrayList uses **RandomeAccess interface**

This is marker interface which contains no method

This hepls to search elements more faster We want 1st element or 1000th elements it will take same time to fetch

**Example :**

**List**

**Test.java**

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Test

{

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

{

List<Integer> list = **new** ArrayList<Integer>();

list.add(20);

list.add(30);

list.add(10);

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

}

}

**Output :**

[20, 30, 10]

**Example : (with Arrays.asList())**

**ArrayList initialization**

**Test.java**

**import** java.util.ArrayList;

**import** java.util.Arrays;

**public** **class** Test

{

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

{

ArrayList<String> list = **new** ArrayList<String>(Arrays.*asList*("Pratap", "Peter", "Harsh"));

System.***out***.println("Elements are : " + list);

}

}

**Output :**

Elements are : [Pratap, Peter, Harsh]

**Example : (with normal initialization)**

**Test.java**

**import** java.util.ArrayList;

**public** **class** Test

{

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

{

ArrayList<String> list = **new** ArrayList<String>();

list.add("python");

list.add("java");

list.add("sql");

System.***out***.println("Elements are : " + list);

}

}

**Output :**

Elements are : [python, java, sql]

**Example : (with for loop)**

**Reading elements**

**Test.java**

**import** java.util.ArrayList;

**public** **class** Test

{

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

{

ArrayList<Integer> list = **new** ArrayList<Integer>();

list.add(14);

list.add(7);

**for** (**int** counter = 0; counter < list.size(); counter++)

{

System.***out***.println(list.get(counter));

}

}

}

**Output :**

14

7

**Example : (with for each loop)**

**Test.java**

**import** java.util.ArrayList;

**public** **class** Test

{

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

{

ArrayList<Integer> list = **new** ArrayList<Integer>();

list.add(14);

list.add(7);

**for** (Integer num : list)

{

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

}

}

}

**Output :**

14

7

**Example : add(**element**);**

**add elements**

**Test.java**

**import** java.util.ArrayList;

**public** **class** Test

{

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

{

ArrayList<String> list = **new** ArrayList<String>();

list.add("Ram");

list.add("Shyam");

System.***out***.println("Elements are : " + list);

}

}

**Output :**

Elements are : [Ram, Shyam]

**Example : add(**index , element**);**

**Test.java**

**import** java.util.ArrayList;

**public** **class** Test

{

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

{

ArrayList<String> al = **new** ArrayList<String>();

al.add("Hi");

al.add("hello");

al.add("yo");

al.add("Test");

System.***out***.println("Elements are :" + al);

// adding string to 3rd position

al.add(3, "Howdy");

System.***out***.println("Elements after adding string Howdy :" + al);

// adding string to 1st position

al.add(0, "Bye");

System.***out***.println("Elements after adding string bye :" + al);

}

}

**Output :**

Elements are :[Hi, hello, yo, Test]

Elements after adding string Howdy :[Hi, hello, yo, **Howdy**, Test]

Elements after adding string bye :[**Bye**, Hi, hello, yo, Howdy, Test]

**Example : addAll(**list**);**

**Test.java**

import java.util.ArrayList;

public class Test

{

public static void main(String args[])

{

ArrayList<String> al = new ArrayList<String>();

al.add("Hi");

al.add("hello");

System.*out*.println("ArrayList1 elements are :" + al);

// ArrayList2 of String Type

ArrayList<String> al2 = new ArrayList<String>();

al2.add("Text1");

al2.add("Text2");

// Adding ArrayList2 into ArrayList1

al.addAll(al2);

System.*out*.println("ArrayList1 after addAll :" + al);

}

}

**Output :**

ArrayList1 elements are :[Hi, hello]

ArrayList1 after addAll :[Hi, hello, **Text1**, **Text2**]

**Example : addAll(**index , list**);**

**Test.java**

**import** java.util.ArrayList;

**public** **class** Test

{

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

{

ArrayList<String> al = **new** ArrayList<String>();

al.add("Hi");

al.add("hello");

System.***out***.println("ArrayList1 elements are :" + al);

// ArrayList2 of String Type

ArrayList<String> al2 = **new** ArrayList<String>();

al2.add("Text1");

al2.add("Text2");

// Adding ArrayList2 into ArrayList1

al.addAll(1,al2);

System.***out***.println("ArrayList1 after addAll :" + al);

}

}

**Output :**

ArrayList1 elements are :[Hi, hello]

ArrayList1 after addAll :[Hi, **Text1**, **Text2**, hello]

**Example : remove(**index**);**

**remove elements**

**Test.java**

**import** java.util.ArrayList;

**public** **class** Test

{

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

{

ArrayList<String> al = **new** ArrayList<String>();

al.add("hitesh");

al.add("ramesh");

al.add("suresh");

System.***out***.println("ArrayList elements : " + al);

al.remove(2);

System.***out***.println("ArrayList After removing element : " + al);

}

}

**Output :**

ArrayList elements : [hitesh, ramesh, **suresh**]

ArrayList After removing element : [hitesh, ramesh]

**Example : remove(**elements**);**

**Test.java**

**import** java.util.ArrayList;

**public** **class** Test

{

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

{

ArrayList<String> al = **new** ArrayList<String>();

al.add("hitesh");

al.add("ramesh");

al.add("suresh");

System.***out***.println("ArrayList elements : " + al);

al.remove("ramesh");

System.***out***.println("ArrayList After removing element : " + al);

}

}

**Output :**

ArrayList elements : [hitesh, **ramesh**, suresh]

ArrayList After removing element : [hitesh, suresh]

**Example : removeAll(**list**);**

**Test.java**

**import** java.util.ArrayList;

**public** **class** Test

{

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

{

ArrayList<String> al = **new** ArrayList<String>();

al.add("hitesh");

al.add("ramesh");

al.add("suresh");

System.***out***.println("ArrayList elements : " + al);

al.removeAll(al);

System.***out***.println("ArrayList After removing elements : " + al);

}

}

**Output :**

ArrayList elements : [hitesh, ramesh, suresh]

ArrayList After removing elements : []

**Example : set(**index**);**

**update elements**

**Test.java**

**import** java.util.ArrayList;

**public** **class** Test

{

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

{

ArrayList<String> al = **new** ArrayList<String>();

al.add("hitesh");

al.add("ramesh");

al.add("suresh");

System.***out***.println("ArrayList elements : " + al);

al.set(1,"panduranga");

System.***out***.println("ArrayList After update elements : " + al);

}

}

**Output :**

ArrayList elements : [hitesh, **ramesh**, suresh]

ArrayList After update elements : [hitesh, **panduranga**, suresh]

**get elements**

**Example : get(**index**);**

**Test.java**

**import** java.util.ArrayList;

**public** **class** Test

{

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

{

ArrayList<String> al = **new** ArrayList<String>();

al.add("hitesh");

al.add("ramesh");

al.add("suresh");

System.***out***.println("ArrayList elements : " + al);

System.***out***.println("get 2nd element: " + al.get(1));

}

}

**Output :**

ArrayList elements : [hitesh, ramesh, suresh]

get 2nd element: **ramesh**

**Example : size();**

**size**

**Test.java**

**import** java.util.ArrayList;

**public** **class** Test

{

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

{

ArrayList<Integer> al = **new** ArrayList<Integer>();

System.***out***.println("Initial size: " + al.size());

al.add(1);

al.add(13);

al.add(45);

System.***out***.println("Size after few additions: " + al.size());

}

}

**Output :**

Initial size: 0

Size after few additions: **3**

**Example : clone();**

**clone**

**Test.java**

**import** java.util.ArrayList;

**public** **class** Test

{

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

{

ArrayList<String> al = **new** ArrayList<String>();

// Adding elements to the ArrayList

al.add("Apple");

al.add("Orange");

System.***out***.println("ArrayList: " + al);

ArrayList<String> a2 = (ArrayList<String>) al.clone();

System.***out***.println("cloned (copy) ArrayList: " + a2);

}

}

**Output :**

ArrayList: [Apple, Orange]

cloned (copy) ArrayList: [**Apple**, **Orange**]

**Example : indexOf(**element**);**

**get index**

**Test.java**

**import** java.util.ArrayList;

**public** **class** Test

{

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

{

ArrayList<String> al = **new** ArrayList<String>();

al.add("AB");

al.add("CD");

al.add("EF");

al.add("EF");

System.***out***.println("Index of 'EF': " + al.indexOf("EF"));

System.***out***.println("Index of ‘hitesh’ : " + al.indexOf("hitesh"));

}

}

**Output :**

Index of 'EF': **2**

Index of ‘hitesh’ : **-1**

**Example : lastIndexOf(**element**);**

**Test.java**

**import** java.util.ArrayList;

**public** **class** Test

{

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

{

ArrayList<String> al = **new** ArrayList<String>();

al.add("AB");

al.add("CD");

al.add("EF");

al.add("EF");

System.***out***.println("Last occurrence of element is : " + al.lastIndexOf("EF"));

}

}

**Output :**

Last occurrence of element is : **3**

**Example : clear(**list**);**

**clear**

**Test.java**

**import** java.util.ArrayList;

**public** **class** Test

{

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

{

ArrayList<String> al1 = **new** ArrayList<String>();

al1.add("abc");

al1.add("xyz");

System.***out***.println("ArrayList contents : " + al1);

al1.clear();

System.***out***.println("ArrayList after clear : " + al1);

}

}

**Output :**

ArrayList contents : [abc, xyz]

ArrayList after clear : []

**Example : isEmpty();**

**check ArrayList empty**

**Test.java**

**import** java.util.ArrayList;

**public** **class** Test

{

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

{

ArrayList<String> al = **new** ArrayList<String>();

System.***out***.println("Is ArrayList Empty: " + al.isEmpty());

al.add("abc");

al.add("xyz");

System.***out***.println("Is ArrayList Empty: " + al.isEmpty());

}

}

**Output :**

Is ArrayList Empty: true

Is ArrayList Empty: false

**Example : contains(**element**);**

**contains**

**Test.java**

**import** java.util.ArrayList;

**public** **class** Test

{

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

{

ArrayList<String> al = **new** ArrayList<String>();

al.add("pen");

al.add("pencil");

al.add("ink");

al.add("notebook");

System.***out***.println("ArrayList contains 'ink pen': " + al.contains("ink pen"));

System.***out***.println("ArrayList contains 'pen': " + al.contains("pen"));

System.***out***.println("ArrayList contains 'book': " + al.contains("book"));

}

}

**Output :**

ArrayList contains 'ink pen': false

ArrayList contains 'pen': true

ArrayList contains 'book': false

**subList**

**Example : subList(**startIndex, endIndex**);**

**Test.java**

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Test

{

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

{

ArrayList<String> al = **new** ArrayList<String>();

al.add("Steve");

al.add("Justin");

al.add("Ajeet");

al.add("John");

System.***out***.println("Original ArrayList Content: " + al);

ArrayList<String> al2 = **new** ArrayList<String>(al.subList(1, 3));

System.***out***.println("SubList stored in ArrayList: " + al2);

List<String> list = al.subList(1, 3);

System.***out***.println("SubList stored in List: " + list);

}

}

**Output :**

Original ArrayList Content: [Steve, Justin, Ajeet, John]

SubList stored in ArrayList: [**Justin**, **Ajeet**]

SubList stored in List: [**Justin**, **Ajeet**]

**Example : Collections.swap(**list, index1, index2**);**

**swap**

**Test.java**

**import** java.util.ArrayList;

**import** java.util.Collections;

**public** **class** Test

{

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

{

ArrayList<String> al = **new** ArrayList<String>();

al.add("Sachin");

al.add("Rahul");

al.add("Saurav");

System.***out***.println("ArrayList contents : " + al);

Collections.*swap*(al, 1, 2);

System.***out***.println("ArrayList after swap : " + al);

}

}

**Output :**

ArrayList contents : [Sachin, Rahul, Saurav]

ArrayList after swap : [Sachin, **Saurav**, **Rahul**]

**Example : Collections.sort(**list**);**

**sort**

**Test.java**

**import** java.util.ArrayList;

**import** java.util.Collections;

**public** **class** Test

{

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

{

ArrayList<String> al = **new** ArrayList<String>();

al.add("hitesh");

al.add("abhijit");

al.add("rahul");

System.***out***.println("ArrayList contents : " + al);

Collections.*sort*(al);

System.***out***.println("ArrayList after sort : " + al);

}

}

**Output :**

ArrayList contents : [hitesh, abhijit, rahul]

ArrayList after sort : [**abhijit**, **hitesh**, **rahul**]

**- compares single element**

**comparable interface**

**- uses compareTo() method**

**Example : (sorting int values)**

**Test.java**

**import** java.util.ArrayList;

**import** java.util.Collections;

**public** **class** Test

{

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

{

ArrayList<Laptop> list = **new** ArrayList<>();

list.add(**new** Laptop("lenovo", 8, 50000));

list.add(**new** Laptop("dell", 4, 30000));

list.add(**new** Laptop("apple", 16, 80000));

Collections.*sort*(list);

System.***out***.println("\nList after sorting : ");

**for** (Laptop l : list)

{

System.***out***.println("list is : " + l);

}

}

}

**Laptop.java**

**public** **class** Laptop **implements** Comparable<Laptop>

{

**private** String brand;

**private** **int** ram;

**private** **int** price;

**public** Laptop(String brand, **int** ram, **int** price)

{

**this**.brand = brand;

**this**.ram = ram;

**this**.price = price;

}

**public** String getBrand()

{

**return** brand;

}

**public** **void** setBrand(String brand)

{

**this**.brand = brand;

}

**public** **int** getRam()

{

**return** ram;

}

**public** **void** setRam(**int** ram)

{

**this**.ram = ram;

}

**public** **int** getPrice()

{

**return** price;

}

**public** **void** setPrice(**int** price)

{

**this**.price = price;

}

@Override

**public** String toString()

{

**return** "Laptop [brand=" + brand + ", ram=" + ram + ", price=" + price + "]";

}

@Override

**public** **int** compareTo(Laptop lap2)

{

**if** (**this**.getPrice() < lap2.getPrice())

{

**return** -3;

}

**if** (**this**.getPrice() == lap2.getPrice())

{

**return** 0;

}

**return** 2;

}

}

**Output :**

List after sorting :

list is : Laptop [brand=dell, ram=4, price=**30000**]

list is : Laptop [brand=lenovo, ram=8, price=**50000**]

list is : Laptop [brand=apple, ram=16, price=**80000**]

**Example : (sorting String values)**

**Test.java**

**import** java.util.ArrayList;

**import** java.util.Collections;

**public** **class** Test

{

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

{

ArrayList<Laptop> list = **new** ArrayList<>();

list.add(**new** Laptop("lenovo", 8, 50000));

list.add(**new** Laptop("dell", 4, 30000));

list.add(**new** Laptop("apple", 16, 80000));

Collections.*sort*(list);

System.***out***.println("\nList after sorting : ");

**for** (Laptop l : list)

{

System.***out***.println("list is : " + l);

}

}

}

**Laptop.java**

**public** **class** Laptop **implements** Comparable<Laptop>

{

**private** String brand;

**private** **int** ram;

**private** **int** price;

**public** Laptop(String brand, **int** ram, **int** price)

{

**this**.brand = brand;

**this**.ram = ram;

**this**.price = price;

}

**public** String getBrand()

{

**return** brand;

}

**public** **void** setBrand(String brand)

{

**this**.brand = brand;

}

**public** **int** getRam()

{

**return** ram;

}

**public** **void** setRam(**int** ram)

{

**this**.ram = ram;

}

**public** **int** getPrice()

{

**return** price;

}

**public** **void** setPrice(**int** price)

{

**this**.price = price;

}

@Override

**public** String toString()

{

**return** "Laptop [brand=" + brand + ", ram=" + ram + ", price=" + price + "]";

}

@Override

**public** **int** compareTo(Laptop lap2)

{

**return** **this**.getBrand().compareTo(lap2.getBrand()); // this compareTo is String method

}

}

**Output :**

List after sorting :

list is : Laptop [brand=**apple**, ram=16, price=80000]

list is : Laptop [brand=**dell**, ram=4, price=30000]

list is : Laptop [brand=**lenovo**, ram=8, price=50000]

**- compares multiple elements**

**comparator interface**

**- uses compare() method**

**Example : (sorting int and String values together)**

**Test.java**

**import** java.util.ArrayList;

**import** java.util.Collections;

**public** **class** Test

{

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

{

ArrayList<Laptop> list = **new** ArrayList<>();

list.add(**new** Laptop("lenovo", 8, 50000));

list.add(**new** Laptop("dell", 4, 30000));

list.add(**new** Laptop("apple", 16, 80000));

Collections.*sort*(list, **new** Price());

System.***out***.println("\nList after sorting as per price : ");

**for** (Laptop l : list)

{

System.***out***.println("list is : " + l);

}

System.***out***.println("----------------------");

Collections.*sort*(list, **new** Brand());

System.***out***.println("\nList after sorting as per Brand : ");

**for** (Laptop l : list)

{

System.***out***.println("list is : " + l);

}

}

}

**Price.java**

**import** java.util.Comparator;

**public** **class** Price **implements** Comparator<Laptop>

{

**public** **int** compare(Laptop l1, Laptop l2)

{

**if** (l1.getPrice() < l2.getPrice())

{

**return** -3;

}

**if** (l1.getPrice() == l2.getPrice())

{

**return** 0;

}

**return** 2;

}

}

**Brand.java**

**import** java.util.Comparator;

**public** **class** Brand **implements** Comparator<Laptop>

{

**public** **int** compare(Laptop l1, Laptop l2)

{

**return** l1.getBrand().compareTo(l2.getBrand());

}

}

**Laptop.java**

**public** **class** Laptop

{

**private** String brand;

**private** **int** ram;

**private** **int** price;

**public** Laptop(String brand, **int** ram, **int** price)

{

**this**.brand = brand;

**this**.ram = ram;

**this**.price = price;

}

**public** String getBrand()

{

**return** brand;

}

**public** **void** setBrand(String brand)

{

**this**.brand = brand;

}

**public** **int** getRam()

{

**return** ram;

}

**public** **void** setRam(**int** ram)

{

**this**.ram = ram;

}

**public** **int** getPrice()

{

**return** price;

}

**public** **void** setPrice(**int** price)

{

**this**.price = price;

}

@Override

**public** String toString()

{

**return** "Laptop [brand=" + brand + ", ram=" + ram + ", price=" + price + "]";

}

}

**Output :**

List after sorting as per price :

list is : Laptop [brand=dell, ram=4, price=**30000**]

list is : Laptop [brand=lenovo, ram=8, price=**50000**]

list is : Laptop [brand=apple, ram=16, price=**80000**]

**----------------------**

List after sorting as per Brand :

list is : Laptop [brand=**apple**, ram=16, price=80000]

list is : Laptop [brand=**dell**, ram=4, price=30000]

list is : Laptop [brand=**lenovo**, ram=8, price=50000]

**Internal working**

**vector**

**1) Default size :**

**- Vector default size is 0**

**2) size after 10th element :**

**- when array size gets full till 10**

**- then it increases array size dynamically by 100%**

**Question :**

1. **ArrayList vs Vector?**



|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| ArrayList next capacity is **increased by 50%** | Vector next capacity is **increased by 100%** |
| methods are **not synchronized** | methods are **synchronized** |

**LinkedList**

- LinkedList is implemented using the **doubly linked list** data structure

- In doubly LinkedList there are next and previous pointers so we can **traverse in both direction**

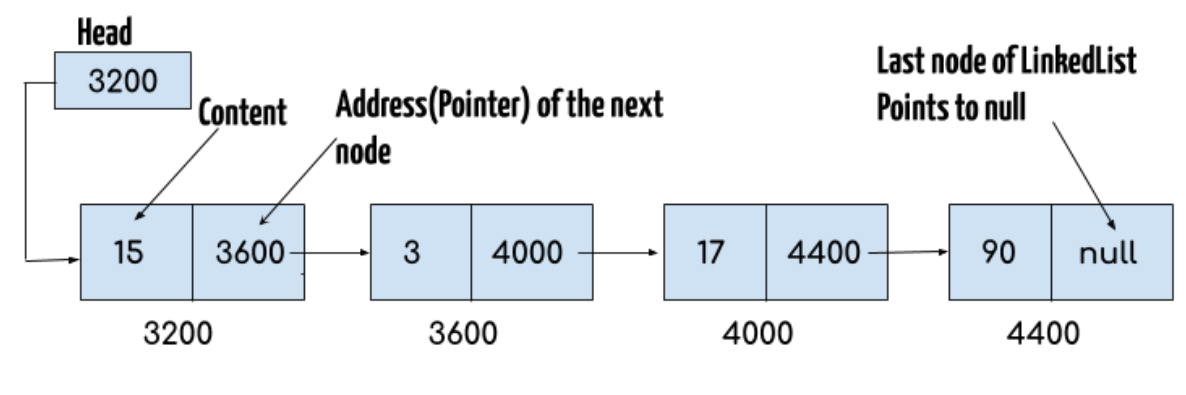
- It also has few disadvantages like the nodes cannot be accessed directly instead we need to start from the head and follow through the link to reach to a node we wish to access (in short **performance of searching is bad** as compared to ArrayList)

- In LinkedList if we want to add,change,delete an element then instead of shifting position of elements it will just **change pointer of nodes**

**Note :**

All methods in LinkedList are same as in ArrayList because they both implements same interface List But few methods are different we will discuss them.

**singly linkedList**



- **Head** contains address of **first node**

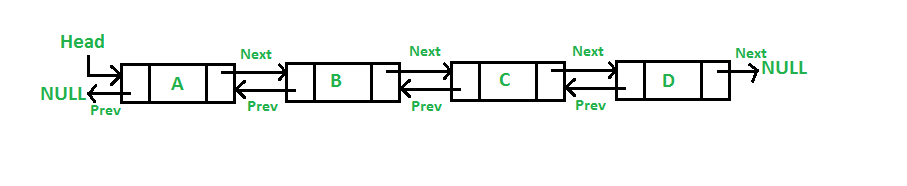
- **Each node** contains **data** and **address** of **next node**

- **Last node** contains **data** and **address** **null** as it is not pointing to any node

- traverse in **both direction**

**doubly linkedList**

- LinkedList class uses Doubly LinkedList **internally**



- **Head** contains **address** of **first node**

- **Each node** contains **data**, **address** of **previous node**, **address** of **next node**

- **Last node** contains **data**, **address** of **previous node**, **address** of **next node** as **null** as **next is not pointing to any node**

**Example :**

**Test.java**

**import** java.util.LinkedList;

**public** **class** Test

{

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

{

LinkedList<String> list = **new** LinkedList<String>();

list.add("Apple");

list.add("Orange");

list.add("Mango");

**for** (String a : list)

{

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

}

}

}

**Output :**

Apple

Orange

Mango

**Question :**

1. **ArrayList vs LinkedList?**



|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| Uses **dynamic array** internally | Uses **doubly linkedList** internally |
| **Add/remove** elemetns is **slow** | **Add/remove** elemetns is **fast** |
| **Search** element **faster** | **Search** element **slower** |
| **Memory** **consumption** is **less** | **Memory consumption** is **more** |
| Works as **List** | Works as **List** and **queue** |

1. Can ArrayList and LinkedList store **null** values ?

* **YES**

**HashSet**

- HashSet implements **Set** interface

- It internally uses **HashMap**

**Internal working**

**- HashSet internally uses HashMap**

**- when we use below line,**

hset.add("Apple");

- HashMap uses **key** = “**ram**” and **value** = **some predefined Object**

- Refer HashMap chapter for HashMap internal working

**Example :**

**Test.java**

**import** java.util.HashSet;

**public** **class** Test

{

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

{

HashSet hset = **new** HashSet();

hset.add("ram");

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

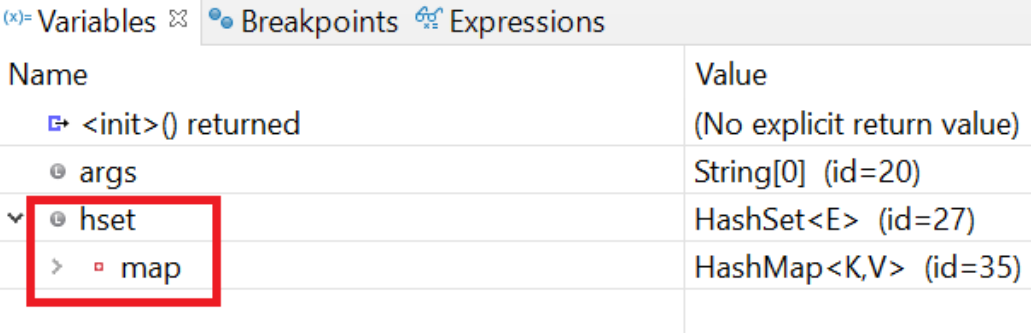
}

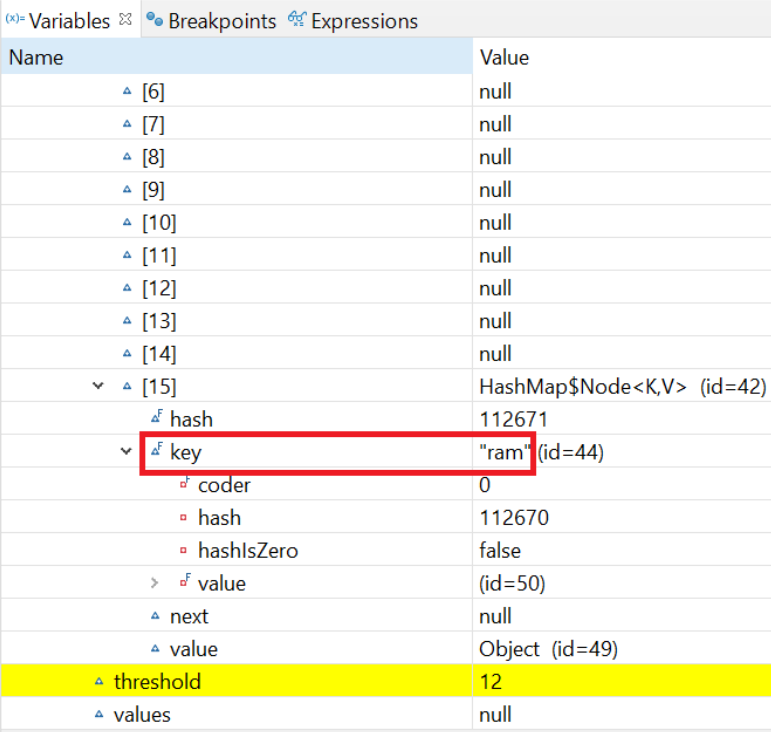
}

**Output :**

[ram]

**Debug mode :**



As shown below “**ram**” is getting stored as a **key** in **HashMap**,

**1) order of elements not preseved**

**features**

**2) elements are not duplicate**

**3) null values allowed**

**but if you enter more than one null value it will display only one**

**4) elements are not synchronized**

**Example : (elements not ordered)**

**Test.java**

**import** java.util.HashSet;

**public** **class** Test

{

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

{

HashSet<String> hset = **new** HashSet<String>();

hset.add("Apple");

hset.add("Mango");

hset.add("Grapes");

hset.add("Orange");

hset.add("Fig");

System.***out***.println("hashset contents : " + hset);

}

}

**Output :**

hashset contents : [Apple, Grapes, Fig, Mango, Orange]

**Example : (elements not duplicate)**

**Test.java**

**import** java.util.HashSet;

**public** **class** Test

{

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

{

HashSet<String> hset = **new** HashSet<String>();

hset.add("Apple");

hset.add("Apple");

System.***out***.println("hashset contents : " + hset);

}

}

**Output :**

hashset contents : [Apple]

**Example : (elements can be null)**

**Example : (elements can be null)**

**Test.java**

**import** java.util.HashSet;

**public** **class** Test

{

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

{

HashSet<String> hset = **new** HashSet<String>();

hset.add("Apple");

hset.add(**null**);

hset.add(**null**);

System.***out***.println("hashset contents : " + hset);

}

}

**Output :**

hashset contents : [null, Apple]

- HashMap implements **Map** interface

**HashMap**

- Data stored in <**key**, **value**> format

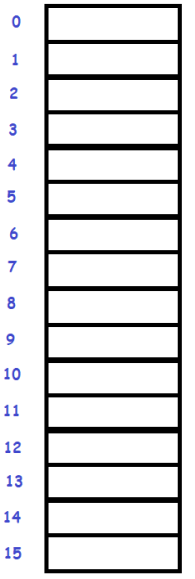
**Internal working**

**1) Buckets :**

**- HashMap data stored inside buckets**

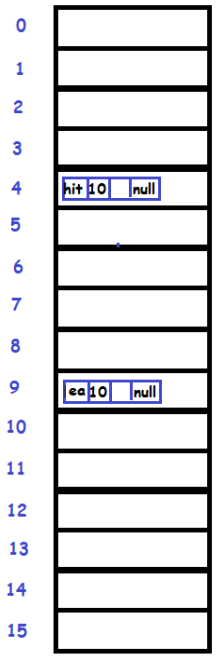
**- these buckets are stored in Heap memory**

**- default size of buckets is 16**



**2) LinkedList :**

**- data inside buckets are stored as LinkedList**



**3) Loading factor :**

**- default size of loading factor is 0.75**

**- as default bucket size is 16**

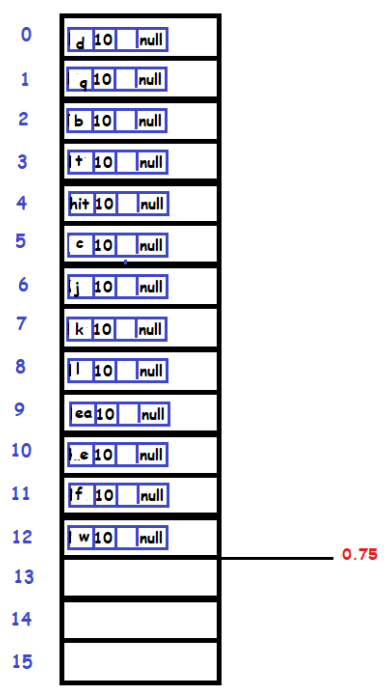
**- threshold = 0.75 x 16 = 12 (i.e. 75% of 16)**

**4) Rehashing :**

**- when 12 buckets get filled up**

**- then jvm increases size of bucket to its** **double**

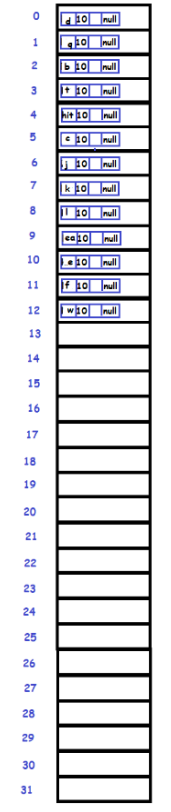
**- here it will become 32 (i.e.16 \* 2)**



**As it reached to 12 th index**

**Now it will increase size to double i.e. 32**

**As shown in below image,**



**5) Hash collision :**

**- as hashCode of key is genereated in hashmap**

**- so if any keys having same hashcodes**

**- then its called as Hash collision**

**- because as hashcode is same bucket index will also same**

**- e.g. “FB”.hashCode() = 1256**

**“EA”.hashCose() = 1256**

**So the bucket index will also be same say 4 for both of them**

**- here JVM might get confused to which data store at index 4 ?**

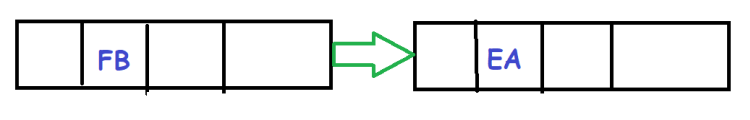
**6) equals() method :**

**- equals() method is the solution for hash collision**

**- e.g. “FB”.equals(“EA”);**

**As FB and EA are not same values**

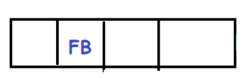
**So both of them will get stored in bucket at index 4 In linkedList manner**



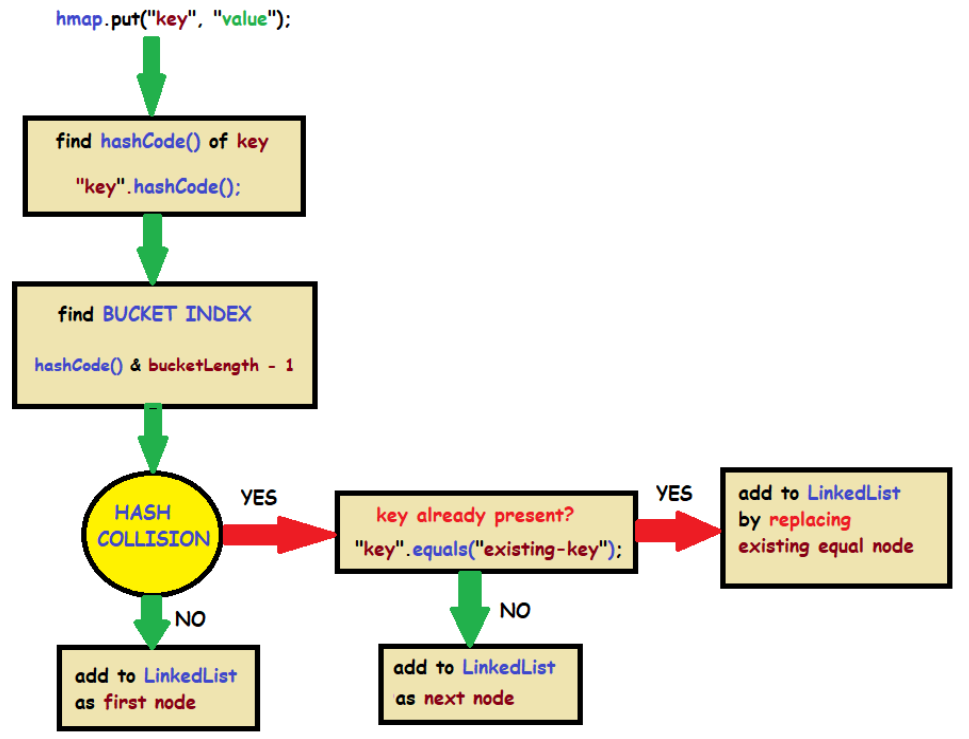
**- e.g. “FB”.equals(“FB”);**

**As FB and FB are same values**

**So only of them which latest value gets replaced in linkedList manner**



**Diagram of HashMap working :**



**Example 1 : (without hash collision)**

**Test.java**

**import** java.util.HashMap;

**public** **class** Test

{

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

{

HashMap<String, String> hmap = **new** HashMap<>();

hmap.put("FB", "hitesh");

}

}

**Internal working :**

**Step 1: generate hashCode()**

**hashcode = “FB”.hashCode(); //key.hashcode();**

**hashcode = 1256**

**Step 2: generate bucket index**

**Bucket index = “FB”.hashCode() & (16 – 1) //key.hashcode() & (bucketLength - 1)**

**Bucket index = 1256 & (16 – 1)**

**Bucket index = 4**

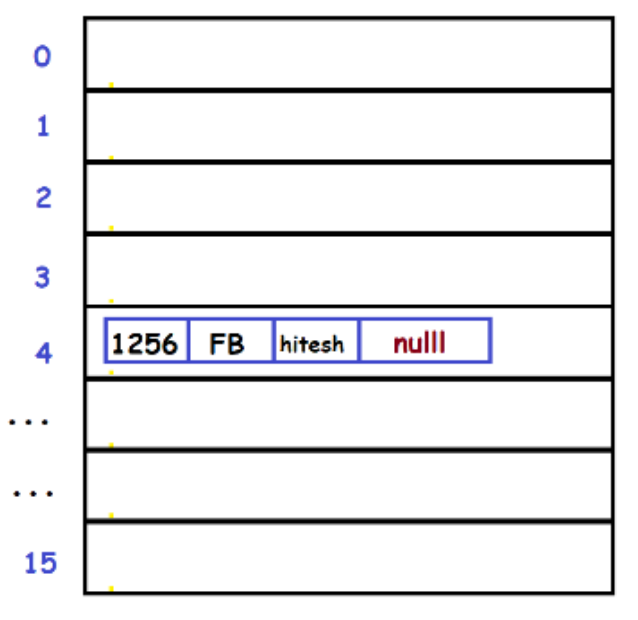
**Step 3: hash collision occurred?**

**NO**

**Step 4: insert data in LinkedList as first node**

**Data will get insert at index 4 of bucket**

**as shown below,**



**Example 2 : (with hash collision but without equals())**

**Test.java**

**import** java.util.HashMap;

**public** **class** Test

{

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

{

HashMap<String, String> hmap = **new** HashMap<>();

hmap.put("FB", "hitesh");

hmap.put("LD", "manish");

hmap.put("EA", "ramesh");

}

}

**Internal working :**

**Step 1: generate hashCode()**

**hashcode = “FB”.hashCode(); //key.hashcode();**

**hashcode = 1256**

**------------------------------**

**hashcode = “LD”.hashCode(); //key.hashcode();**

**hashcode = 2106**

**------------------------------**

**hashcode = “EA”.hashCode(); //key.hashcode();**

**hashcode = 1256**

**Step 2: generate bucket index**

**Bucket index = “FB”.hashCode() & (16 – 1) //key.hashcode() & (bucketLength - 1)**

**Bucket index = 1256 & (16 – 1)**

**Bucket index = 4**

**------------------------------**

**Bucket index = “LD”.hashCode() & (16 – 1) //key.hashcode() & (bucketLength - 1)**

**Bucket index = 2106 & (16 – 1)**

**Bucket index = 7**

**------------------------------**

**Bucket index = “EA”.hashCode() & (16 – 1) //key.hashcode() & (bucketLength - 1)**

**Bucket index = 1256 & (16 – 1)**

**Bucket index = 4**

**Step 3: hash collision occurred?**

**YES (as hashCode of FB and EA is same i.e. 1256)**

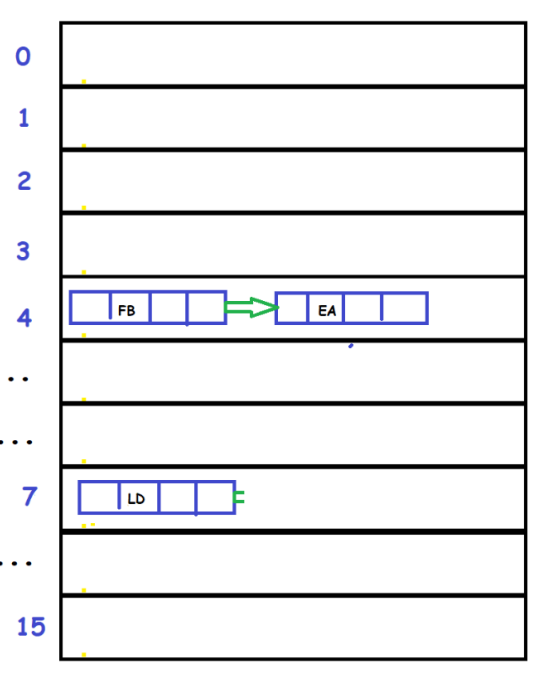
**Step 4: check key equals() or not?**

**NO**

**Step 4: insert data in LinkedList as next node**

**FB and EA get insert at index 4**

**& LD get insert at index 7 of bucket as shown below,**



**Example 3 : (with hash collision and with equals())**

**Test.java**

**import** java.util.HashMap;

**public** **class** Test

{

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

{

HashMap<String, String> hmap = **new** HashMap<>();

hmap.put("FB", "hitesh");

hmap.put("LD", "manish");

hmap.put("FB", "ramesh");

}

}

**Internal working :**

**Step 1: generate hashCode()**

**hashcode = “FB”.hashCode(); //key.hashcode();**

**hashcode = 1256**

**------------------------------**

**hashcode = “LD”.hashCode(); //key.hashcode();**

**hashcode = 2106**

**------------------------------**

**hashcode = “FB”.hashCode(); //key.hashcode();**

**hashcode = 1256**

**Step 2: generate bucket index**

**Bucket index = “FB”.hashCode() & (16 – 1) //key.hashcode() & (bucketLength - 1)**

**Bucket index = 1256 & (16 – 1)**

**Bucket index = 4**

**------------------------------**

**Bucket index = “LD”.hashCode() & (16 – 1) //key.hashcode() & (bucketLength - 1)**

**Bucket index = 2106 & (16 – 1)**

**Bucket index = 7**

**------------------------------**

**Bucket index = “FB”.hashCode() & (16 – 1) //key.hashcode() & (bucketLength - 1)**

**Bucket index = 1256 & (16 – 1)**

**Bucket index = 4**

**Step 3: hash collision occurred?**

**YES (as hashCode of FB and FB is same i.e. 1256)**

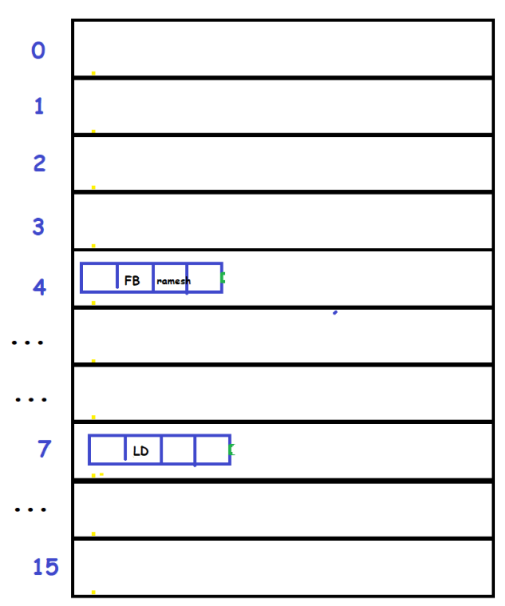
**Step 4: check key equals() or not?**

**YES (as FB came twice so “FB”.equals(“FB”) = true)**

**Step 4: insert data in LinkedList as existing node**

**FB get replaced at index 4 with value = ramesh**

**& LD get insert at index 7 of bucket as shown below,**



**Debug hashmap**

**Example 1 : (keys with different hashcode)**

**Test.java**

**import** java.util.HashMap;

**public** **class** Test

{

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

{

HashMap hmap = **new** HashMap();

hmap.put(101, "hit");

hmap.put(102, "man");

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

}

}

**Output :**

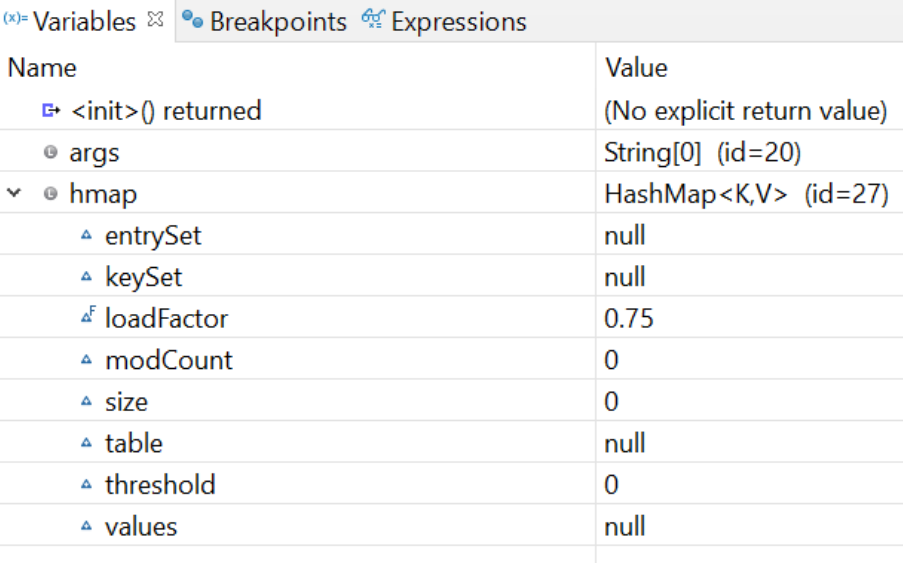
{101=hit, 102=man}

**Example : (elements**

**Debug mode :**

**1) creating hashmap :**

**HashMap hmap = new HashMap();**

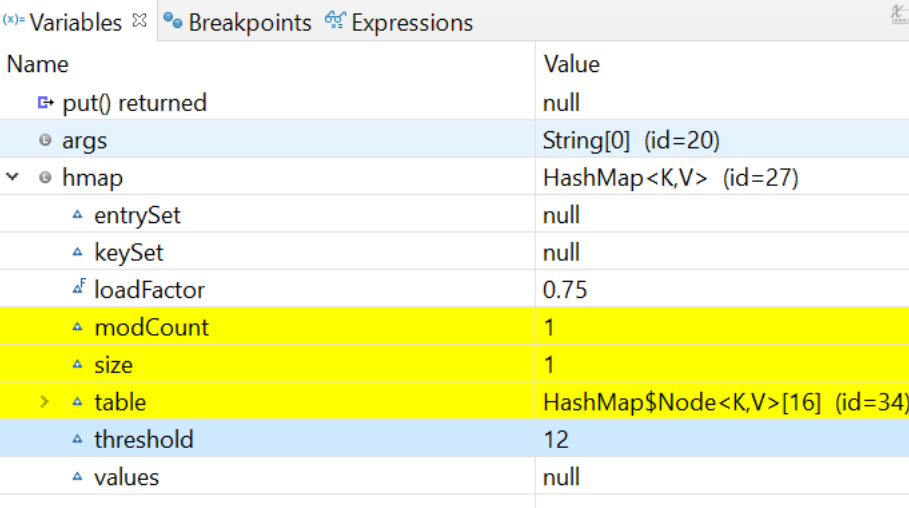


**loadFactor is 0.75**

**threshold is 0**

**2) creating hashmap :**

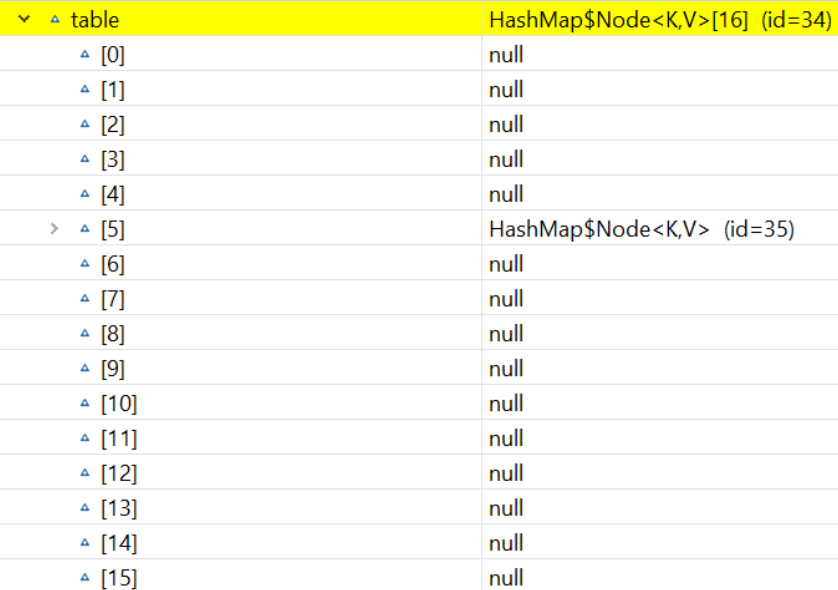
**hmap.put(101, "hit");**



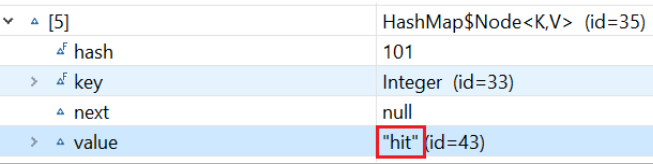
**loadFactor is 0.75**

**threshold is 12 (i.e. 75% of 16)**

**just open table, (16 buckets created)**

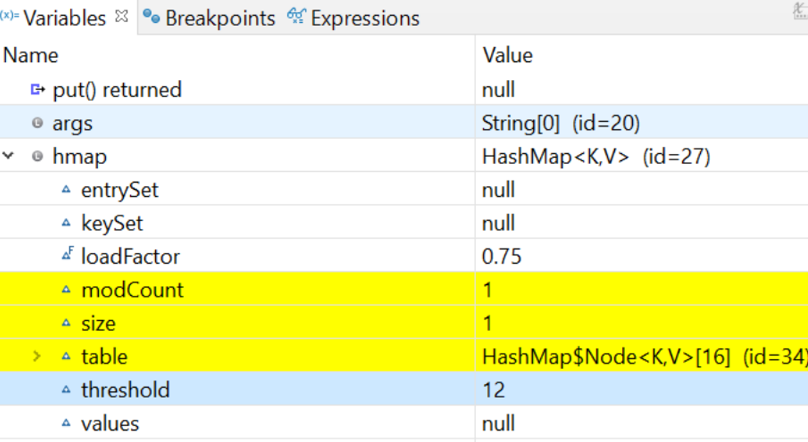


**Value is inserted at bucket index 5,**



**3) creating hashmap :**

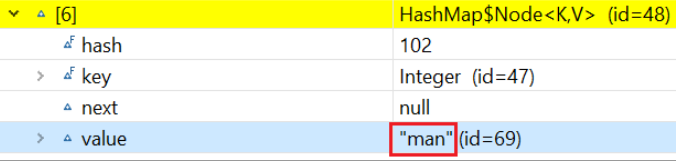
**hmap.put(102, "man");**



**loadFactor is 0.75**

**threshold is 12 (i.e. 75% of 16)**

**Value is inserted at bucket index 6,**



**Example 2 : (keys with same hashcode)**

**Test.java**

**import** java.util.HashMap;

**public** **class** Test

{

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

{

HashMap hmap = **new** HashMap();

hmap.put("FB", "hit");

hmap.put("EA", "man");

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

}

}

**Output :**

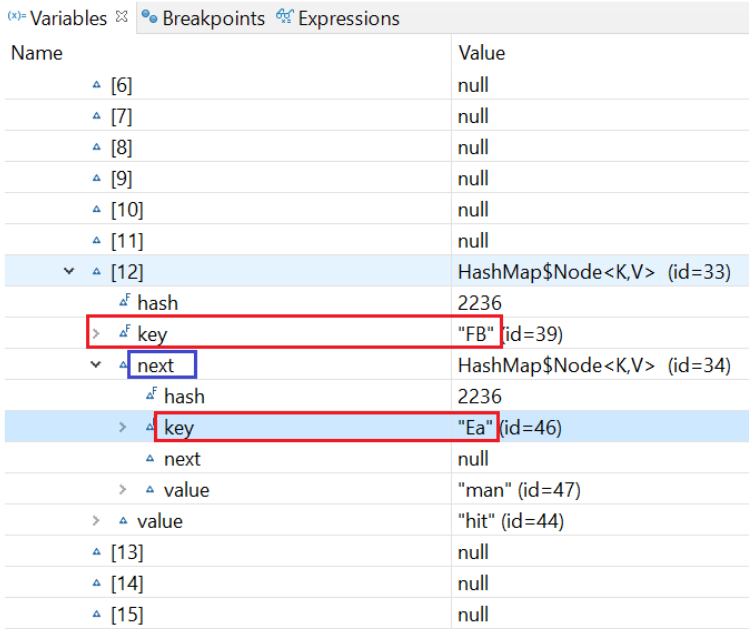
{FB=hit, EA=man}

**Example : (elements**

**Debug mode :**

**- as key FB and Ea has same hashCode**

**- gets stored in same bucket index as next node of LinkedList**



**Example 3 : (bucket size increases when reaches 0.75 of bucket)**

**Test.java**

**import** java.util.HashMap;

**public** **class** Test

{

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

{

HashMap hmap = **new** HashMap();

hmap.put(101, "hit");

hmap.put(102, "man");

hmap.put(103, "tom");

hmap.put(104, "jam");

hmap.put(105, "yep");

hmap.put(106, "chao");

hmap.put(107, "kim");

hmap.put(108, "bruce");

hmap.put(113, "jackie");

hmap.put(115, "jet");

hmap.put(116, "scot");

hmap.put(125, "tony");

hmap.put(168, "kavya");

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

}

}

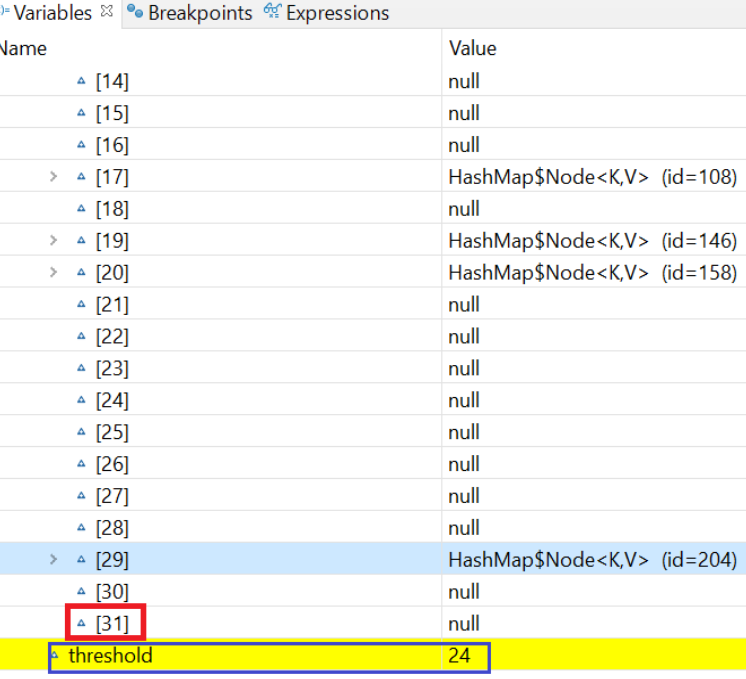
**Output :**

{101=hit, 102=man, 103=tom, 104=jam, 168=kavya, 105=yep, 106=chao, 107=kim, 108=bruce, 113=jackie, 115=jet, 116=scot, 125=tony}

**Example : (elements**

**Debug mode :**

**- Here size of bucket gets double 16 \*2 = 32 and threshold = 24 (i.e. 12 x 2 =24)**



**- HashTable implements Map interface**

**HashTable**

**- Same has HashMap but its methods are synchronized**

**- Data stored in <key, value> format**

**1) order of elements not preseved**

**features**

**2) keys are not duplicate but values can be duplicate**

**3) null key and values not allowed**

**4) elements are synchronized**

**Example : (elements not ordered)**

**Test.java**

**import** java.util.\*;

**public** **class** Test

{

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

{

Hashtable<Integer, String> ht = **new** Hashtable<Integer, String>();

ht.put(101, " ajay");

ht.put(102, "Rahul");

ht.put(106, "Rahul");

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

}

}

**Output :**

{106=Rahul, 102=Rahul, 101= ajay}

**Example : (keys not duplicate)**

**Test.java**

**import** java.util.\*;

**public** **class** Test

{

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

{

Hashtable<Integer, String> ht = **new** Hashtable<Integer, String>();

ht.put(101, " ajay");

ht.put(101, "Rahul");

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

}

}

**Output :**

{101=Rahul}

**Example : (values can be duplicate)**

**Test.java**

**import** java.util.\*;

**public** **class** Test

{

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

{

Hashtable<Integer, String> ht = **new** Hashtable<Integer, String>();

ht.put(101, " ajay");

ht.put(102, "ajay");

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

}

}

**Output :**

{102=ajay, 101= ajay}

**Example : (null key and value not allowed)**

**Test.java**

**import** java.util.\*;

**public** **class** Test

{

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

{

Hashtable<Integer, String> ht = **new** Hashtable<Integer, String>();

ht.put(**null**, "ajay");

ht.put(106, **null**);

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

}

}

**Output :**

Exception in thread "main" java.lang.NullPointerException: Cannot invoke "Object.hashCode()" because "key" is null

at java.base/java.util.Hashtable.put(Hashtable.java:481)

at Test.main(Test.java:8)

**Question :**

**1) HashMap vs HashSet ?**



|  |  |
| --- | --- |
| **HashMap** | **HashTable** |
| **order of elements not preseved** | **order of elements not preseved** |
| **keys are not duplicate but**  **values can be duplicate** | **keys are not duplicate but**  **values can be duplicate** |
| **null key and values allowed** | **null key and values not allowed** |
| **elements are not synchronized** | **elements are synchronized** |

**TreeSet**

**- Elements are already sorted in TreeSet**

**Example :**

**Test.java**

**import** java.util.\*;

**public** **class** Test

{

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

{

TreeSet ts = **new** TreeSet();

// Add elements to the tree set

ts.add("narsimha");

ts.add("vithal");

ts.add("pandurang");

ts.add("krishna");

ts.add("shripad");

ts.add("yogananda");

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

}

}

**Output :**

[krishna, narsimha, pandurang, shripad, vithal, yogananda]

**TreeMap**

**- key/values are already sorted in TreeMap**

**Example :**

**Test.java**

**import** java.util.\*;

**public** **class** Test

{

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

{

TreeMap tm = **new** TreeMap();

// Put elements to the map

tm.put("Zara", **new** ~~Double~~(3434.34));

tm.put("Mahnaz", **new** ~~Double~~(123.22));

tm.put("Ayan", **new** ~~Double~~(1378.00));

tm.put("Daisy", **new** ~~Double~~(99.22));

tm.put("Qadir", **new** ~~Double~~(-19.08));

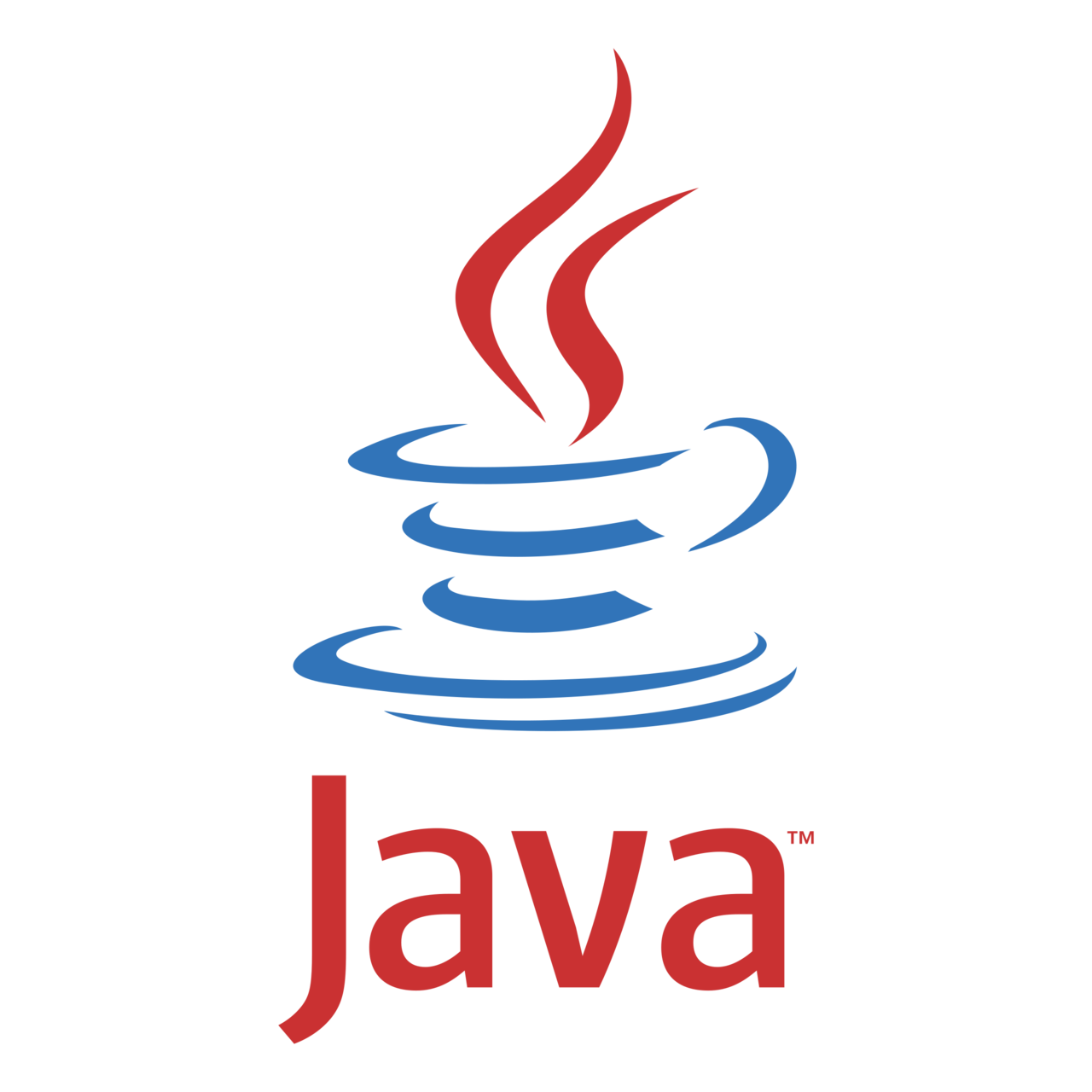
System.***out***.println(tm);

}

}

**Output :**

{Ayan=1378.0, Daisy=99.22, Mahnaz=123.22, Qadir=-19.08, Zara=3434.34}



**1.8**

**Functional interface**

**- In functional interface only single abstract method is allowed**

**- It can have static and normal methods too**

**- Helps to achieve functional programming approach**

**- Helps to use lambda expressions**

**Example : (normal interface)**

**Test.java**

**public** **class** Test **implements** Sample

{

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

{

Test t = **new** Test();

t.m1();

t.m2();

}

@Override

**public** **void** m1()

{

System.***out***.println("executing m1...");

}

@Override

**public** String m2()

{

System.***out***.println("executing m2...");

**return** "done";

}

}

**Sample.java**

**public** **interface** Sample

{

**public** **void** m1();

**public** String m2();

}

**Output :**

executing m1...

executing m2...

**Example : (functional interface)**

**Test.java**

**public** **class** Test **implements** Sample

{

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

{

Test t = **new** Test();

t.m1();

}

@Override

**public** **void** m1()

{

System.***out***.println("executing m1...");

}

}

**Sample.java**

**public** **interface** Sample

{

**public** **void** m1();

}

**Output :**

executing m1...

**- @FunctionalInterface will give compile time error if we add more than one abstract method**

**@FunctionalInterface**

**- It can have static and normal methods too**

**Example : (gives error when adding more than one abstract method)**

**Test.java**

**public** **class** Test **implements** Sample

{

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

{

Test t = **new** Test();

t.m1();

}

@Override

**public** **void** m1()

{

System.***out***.println("executing m1...");

}

}

**Sample.java**

@FunctionalInterface

**public** **interface** Sample

{

**public** **void** m1();

**public** **void** m2(); //compile time error

}

**Output :**

**Compile time error!**

**Example : (can have static/default method)**

**Test.java**

**public** **class** Test **implements** Sample

{

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

{

Test t = **new** Test();

t.m1();

t.m3();

}

@Override

**public** **void** m1()

{

System.***out***.println("executing m1...");

}

}

**Sample.java**

@FunctionalInterface

**public** **interface** Sample

{

**public** **void** m1();

// public void m2(); //compile time error

**public** **default** **void** m3()

{

System.***out***.println("executing default m3...");

}

}

**Output :**

executing m1...

executing default m3...

**Question :**

**1) can Functional interface extend another normal interface or functional interface ?**

**➔ NO (as it allows only one abstract method)**

**2) why Functional interface allows only one abstract method?**

**➔ Otherwise lambda expression will get confused which method to call. (we will learn this in next chapter)**

**Lambda expression**

**- It is an enhanced version of Anonymous class**

**- works only with functional interface**

**Example : (normal class)**

**Test.java**

**public** **class** Test **implements** Sample

{

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

{

Test t = **new** Test();

t.show();

}

@Override

**public** **void** show()

{

System.***out***.println("executing show...");

}

}

**Sample.java**

**public** **interface** Sample

{

**public** **void** show();

}

**Output :**

executing show...

**Example : (Anonymous class)**

**Test.java**

**public** **class** Test

{

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

{

Sample obj = **new** Sample()

{

@Override

**public** **void** show()

{

System.***out***.println("executing show...");

}

};

obj.show();

}

}

**Sample.java**

**public** **interface** Sample

{

**public** **void** show();

}

**Output :**

executing show...

**Example : (Anonymous class)**

**Test.java**

**public** **class** Test

{

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

{

Sample obj = **new** Sample()

{

**public** **void** show()

{

System.***out***.println("executing show...");

}

**public** **void** print(**int** x)

{

System.***out***.println("executing print... "+x);

}

};

obj.show();

obj.print(14);

}

}

**Sample.java**

**public** **interface** Sample

{

**public** **void** show();

**public** **void** print(**int** x);

}

**Output :**

executing show...

executing print... 14

**How to create lambda expression ?**

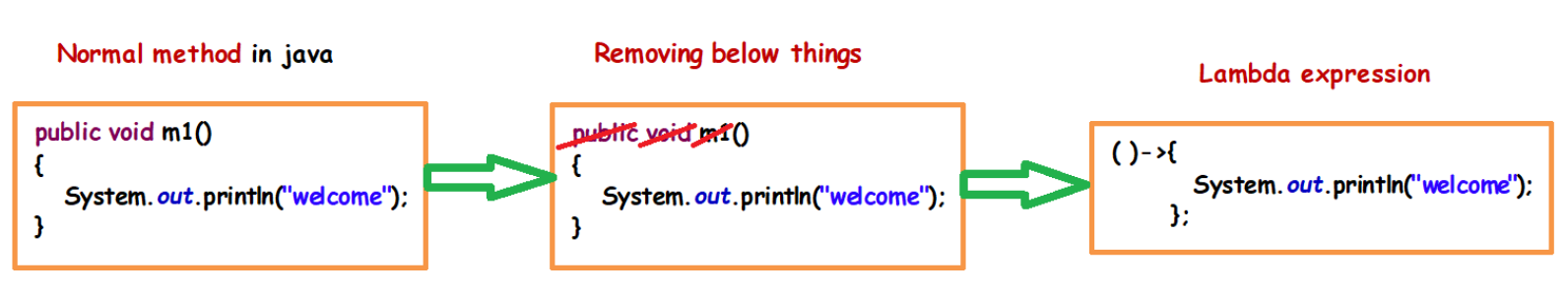
**1) Remove access-modifier**

**2) Remove return type**

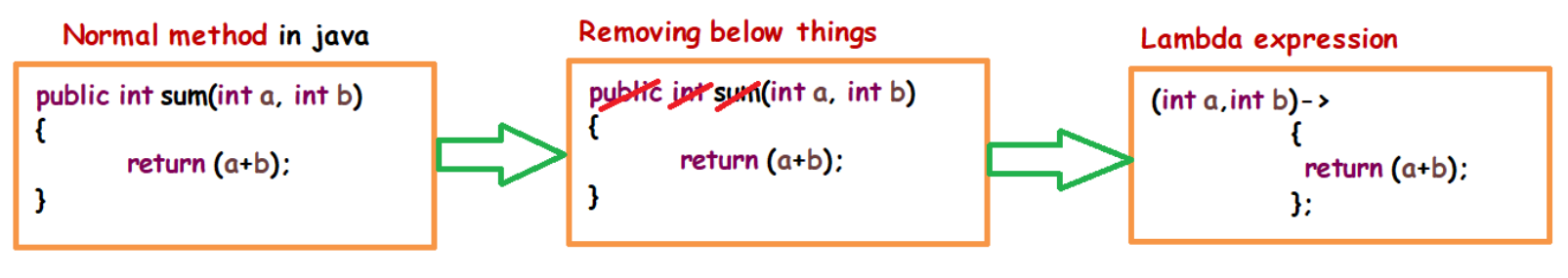
**3) Remove method name**

**4) add ->**

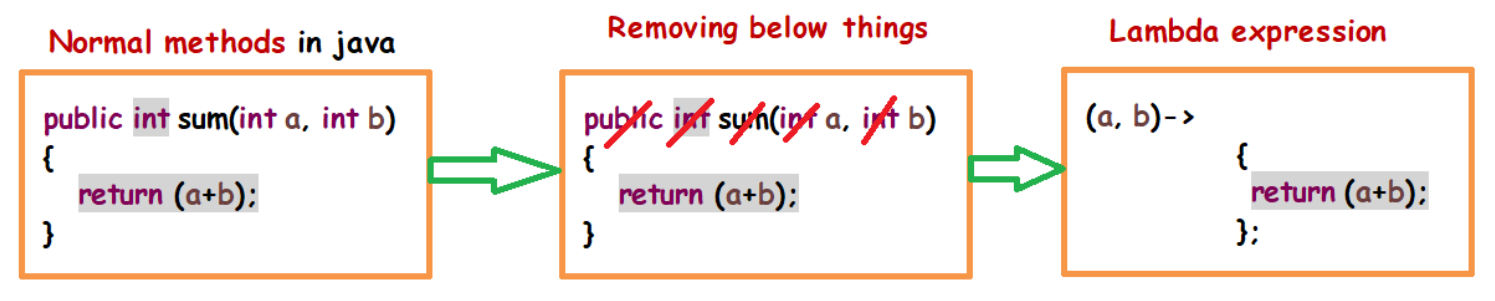
**Example 1 : (no parameter)**



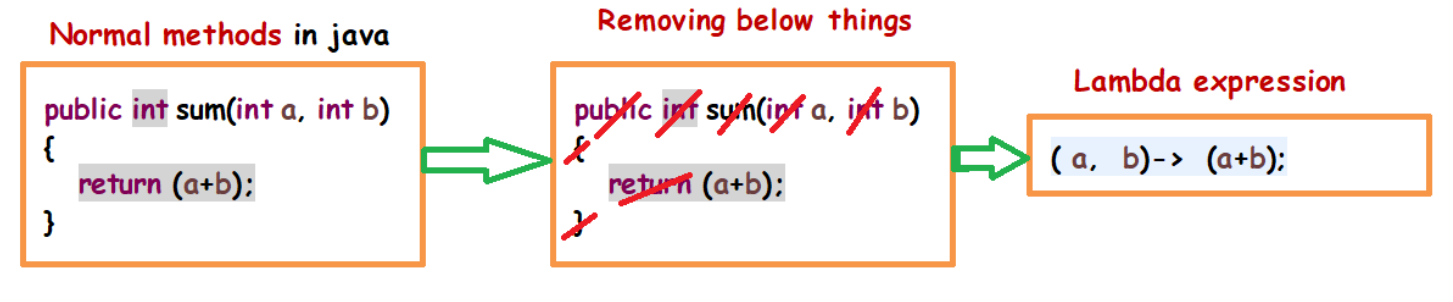
**Example 2 : (with parameter)**



**Example 3 : (with parameter)**



**Example 4 : (we can remove curly braces {} and return keyword if single statement is used)**



**Example : (Lambda expression with no parameter method)**

**Test.java**

**public** **class** Test

{

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

{

Sample obj = () ->

{

System.***out***.println("executing show...");

};

obj.show();

}

}

**Sample.java**

**public** **interface** Sample

{

**public** **void** show();

}

**Output :**

executing show...

**Example : (Lambda expression with parameterized method)**

**Test.java**

**public** **class** Test

{

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

{

Sample obj = (a, b) ->

{

**return** a + b;

};

System.***out***.println("sum is : " + obj.sum(10, 20));

}

}

**Sample.java**

**public** **interface** Sample

{

**public** **int** sum(**int** a,**int** b);

}

**Output :**

sum is : 30

**Example : (Lambda expression with parameterized method)**

**Test.java**

**public** **class** Test

{

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

{

Sample s = (a, b) -> a + b;

System.***out***.println("sum is : " + s.sum(10, 20));

}

}

**Sample.java**

**public** **interface** Sample

{

**public** **int** sum(**int** a,**int** b);

}

**Output :**

sum is : 30

**Example : (Lambda expression with parameterized method)**

**Test.java**

**public** **class** Test

{

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

{

Sample s = (a) -> a.length();

System.***out***.println("length is : " + s.getData("hello"));

}

}

**Sample.java**

**public** **interface** Sample

{

**int** getData(String str);

}

**Output :**

length is : 5

**Example : (Lambda expression with parameterized method)**

**Test.java**

**public** **class** Test

{

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

{

Sample person = (message) ->

{

String str1 = "I would like to say, ";

String str2 = str1 + message;

**return** str2;

};

System.***out***.println(person.getData("time is precious."));

}

}

**Sample.java**

**public** **interface** Sample

{

String getData(String str);

}

**Output :**

I would like to say, time is precious.

**Example : (Lambda expression with forEach method)**

**Test.java**

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Test

{

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

{

List<String> list = **new** ArrayList<String>();

list.add("ankit");

list.add("mayank");

list.forEach((n) -> System.***out***.println(n));

}

}

**Output :**

ankit

mayank

**Example : (Lambda expression with multi-threading)**

**Test.java**

**public** **class** Test

{

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

{

Runnable r1 = () ->

{

System.***out***.println("Thread 1 is running...");

};

Thread t1 = **new** Thread(r1);

t1.start();

Runnable r2 = () ->

{

System.***out***.println("Thread 2 is running...");

};

Thread t2 = **new** Thread(r2);

t2.start();

}

}

**Output :**

Thread 1 is running...

Thread 2 is running...

**forEach() method**

**- helps to iterate elements**

**Example :**

**Test.java**

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Test

{

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

{

List<String> list = **new** ArrayList<String>();

list.add("krishna");

list.add("pandurang");

**for**(String str:list)

{

System.***out***.println("element is : "+str);

}

}

}

**Output :**

element is : krishna

element is : pandurang

**Example : (Lambda expression with forEach method)**

**Test.java**

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Test

{

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

{

List<String> list = **new** ArrayList<String>();

list.add("krishna");

list.add("pandurang");

list.forEach((str) -> System.***out***.println("element is : "+str));

}

}

**Output :**

element is : krishna

element is : pandurang

**Example : (method reference with forEach method)**

**Test.java**

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Test

{

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

{

List<String> list = **new** ArrayList<String>();

list.add("Krishna");

list.add("Pandurang");

list.forEach(System.***out***::println);

}

}

**Output :**

Krishna

Pandurang

**method reference**

**- method reference is used to refer method of functional interface**

**- It is compact and easy form of lambda expression**

**- Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference**

**Syntax : (static method)**

**Interface obj=Class::method**

**Syntax : (instance method)**

**Class obj1 = new Class();**

**Interface obj2= obj1::method**

**Syntax : (constructor)**

**Interface obj1** = **Class**::**new**;

**Class** **obj2**= **obj1**.**method**

**Example : (method reference for static method)**

**Test.java**

**public** **class** Test

{

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

{

Sample sample = Test::*display*;

sample.show();

}

**public** **static** **void** display() // instance method

{

System.***out***.println("displaying...");

}

}

**Sample.java**

**public** **interface** Sample

{

**public** **void** show();

}

**Output :**

displaying...

**Example : (method reference for instance method)**

**Test.java**

**public** **class** Test

{

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

{

Test test = **new** Test();

Sample sample = test::display;

sample.show();

}

**public** **void** display() // instance method

{

System.***out***.println("displaying...");

}

}

**Sample.java**

**public** **interface** Sample

{

**public** **void** show();

}

**Output :**

displaying...

**Example : (constructor reference)**

**Test.java**

**public** **class** Test

{

**public** Test()

{

System.***out***.println("this is constructor...");

}

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

{

Sample sample = Test::**new**;

Test test = sample.show();

test.display();

}

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

{

System.***out***.println("displaying...");

}

}

**Sample.java**

**public** **interface** Sample

{

**public** Test show();

}

**Output :**

this is constructor...

displaying...

**Optional class**

**- It is used to handle NullPointerException**

**Example : (without optional class)**

**Test.java**

**public** **class** Test

{

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

{

String str=**null**;

System.***out***.print(str.toUpperCase());

}

}

**Output :**

Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.toUpperCase()" because "str" is null

at Test.main(Test.java:7)

**Example : (using optional class)**

**Test.java**

**import** java.util.Optional;

**public** **class** Test

{

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

{

String str = **null**;

Optional<String> checkNull = Optional.*ofNullable*(str);

**if** (checkNull.isPresent())

{

String word = str.toUpperCase();

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

} **else**

{

System.***out***.println("word is null");

}

}

}

**Output :**

word is null

**Example : (using optional class)**

**Test.java**

**import** java.util.Optional;

**public** **class** Test

{

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

{

String str = **null**;

Optional<String> checkNull = *getData*(str);

**if** (checkNull.isPresent())

{

String word = str.toUpperCase();

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

} **else**

{

System.***out***.println("word is null");

}

}

**public** **static** Optional<String> getData(String str)

{

**return** Optional.*ofNullable*(str);

}

}

**Output :**

word is null

**- Streams read input from collections and perform various operations on it**

**Stream API**

**forEach()**

**- helps to iterate list**

**Example : (normal for each loop arrayList)**

**Test.java**

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Test

{

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

{

List<String> list = **new** ArrayList();

list.add("vithal");

list.add("krishna");

**for** (String s : list)

{

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

}

}

}

**Output :**

vithal

krishna

**Example : (using stream forEach arrayList)**

**Test.java**

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Test

{

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

{

List<String> list = **new** ArrayList();

list.add("vithal");

list.add("krishna");

list.stream()

.forEach(s->System.***out***.println(s));

}

}

**Output :**

vithal

krishna

**Example : (normal for each loop hashmap)**

**Test.java**

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** Test

{

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

{

Map<String, String> hmap = **new** HashMap();

hmap.put("vithal", "pandharpur");

hmap.put("krishna", "mathura");

hmap.put("vishnu", "vaikuntha");

**for** (Map.Entry<String, String> entry : hmap.entrySet())

{

System.***out***.println("Key = " + entry.getKey() + ", Value = " + entry.getValue());

}

}

}

**Output :**

Key = vishnu, Value = vaikuntha

Key = krishna, Value = mathura

Key = vithal, Value = pandharpur

**Example : (using stream forEach hashmap)**

**Test.java**

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** Test

{

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

{

Map<String, String> hmap = **new** HashMap();

hmap.put("vithal", "pandharpur");

hmap.put("krishna", "mathura");

hmap.put("vishnu", "vaikuntha");

hmap.entrySet()

.stream()

.forEach(s->System.***out***.println(s));

}

}

**Output :**

vishnu=vaikuntha

krishna=mathura

vithal=pandharpur

**- replacement for if else condition**

**filter()**

**Example : (normal if else)**

**Test.java**

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Test

{

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

{

List<String> list = **new** ArrayList();

list.add("vithal");

list.add("krishna");

list.add("vishnu");

**for** (String s : list)

{

**if**(s.startsWith("v"))

{

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

}

}

}

}

**Output :**

vithal

vishnu

**Example : (using filter)**

**Test.java**

import java.util.ArrayList;

import java.util.List;

public class Test

{

public static void main(String[] args)

{

List<String> list = new ArrayList();

list.add("vithal");

list.add("krishna");

list.add("vishnu");

list.stream()

.filter(s->s.startsWith("v"))

.forEach(s->System.*out*.println(s));

}

}

**Output :**

vithal

vishnu

**Example : (find values less than 20)**

**Test.java**

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.stream.Collectors;

**public** **class** Test

{

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

{

List<Integer> list1 = **new** ArrayList<>();

list1.add(12);

list1.add(20);

list1.add(6);

list1.add(82);

list1.stream()

.filter(i -> i < 20)

.collect(Collectors.*toList*())

.forEach(s->System.***out***.println(s));;

}

}

**Output :**

12

6

**Example : (real world example)**

**- find out employees who are eligible for paying tax**

**i..e employess having salary greater than 5 lac**

**- find employee whose dept is civil**

**Employee.java**

**public** **class** Employee

{

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

**private** String name;

**private** String dept;

**private** **long** salary;

**public** Employee(**int** id, String name, String dept, **long** salary)

{

**super**();

**this**.id = id;

**this**.name = name;

**this**.dept = dept;

**this**.salary = salary;

}

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

{

**return** id;

}

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

{

**this**.id = id;

}

**public** String getName()

{

**return** name;

}

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

{

**this**.name = name;

}

**public** String getDept()

{

**return** dept;

}

**public** **void** setDept(String dept)

{

**this**.dept = dept;

}

**public** **long** getSalary()

{

**return** salary;

}

**public** **void** setSalary(**long** salary)

{

**this**.salary = salary;

}

}

**Test.java**

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Collectors;

public class Main

{

public static void main(String[] args)

{

List<Employee> list = new ArrayList();

list.add(new Employee(176,"vithal","IT",600000));

list.add(new Employee(456,"krishna","civil",900000));

list.add(new Employee(268,"panduranga","civil",400000));

list.stream()

.filter(emp->emp.getSalary()>500000) //if(emp.getSalary()>500000)

.collect(Collectors.*toList*())

.forEach(s->System.*out*.println(s.getName()+" - "+s.getSalary()));

System.*out*.println("------------------------");

list.stream()

.filter(emp->emp.getDept().equals("civil")) //if(emp.getDept=civil)

.collect(Collectors.*toList*())

.forEach(s->System.*out*.println(s.getName()+" - "+s.getDept()));

}

}

**Output :**

vithal - 600000

krishna - 900000

------------------------

krishna - civil

panduranga - civil

**- map() is used for data transformation**

**map()**

**Example : (find sqaure of each element)**

**Test.java**

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Test

{

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

{

List<Integer> list1 = **new** ArrayList<>();

list1.add(12);

list1.add(20);

list1.stream()

.map(i -> i \* i)

.forEach(i -> System.***out***.println("square of element is : " + i));

}

}

**Output :**

square of element is : **144**

square of element is : **400**

**- flatMap() is used for data transformation and fatten array**

**flatMap()**

**Example : (find sqaure of each element)**

**Test.java**

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.stream.Stream;

**public** **class** Test

{

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

{

List<Integer> list1 = **new** ArrayList<>();

list1.add(12);

list1.add(6);

list1.stream()

.flatMap(i->Stream.*of*(i\*i))

.forEach(i->System.***out***.println("square of element is : "+i));

}

}

**Output :**

square of element is : **144**

square of element is : **400**

**Example : (flatten array)**

**Test.java**

**import** java.util.Arrays;

**import** java.util.List;

**import** java.util.stream.Collectors;

**public** **class** Test

{

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

{

List<Integer> list1 = Arrays.*asList*(1, 2, 3);

List<Integer> list2 = Arrays.*asList*(4, 5, 6);

List<Integer> list3 = Arrays.*asList*(7, 8, 9);

List<List<Integer>> listOfLists1 = Arrays.*asList*(list1, list2, list3);

System.***out***.println("Before flatten : " + listOfLists1);

List<Integer> newList1 = listOfLists1.stream().flatMap(x -> x.stream()).collect(Collectors.*toList*());

System.***out***.println("After flatten : " + newList1);

}

}

**Output :**

**Before flatten :** [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

**After flatten :** [**1, 2, 3, 4, 5, 6, 7, 8, 9**]

**- returns data as a single value**

**reduce()**

**Example : (find sqaure of each element)**

**Test.java**

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Test

{

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

{

List<Integer> list1 = **new** ArrayList<>();

list1.add(12);

list1.add(20);

**int** sum = list1.stream().reduce(0, (ans, i) -> ans + i);

System.***out***.println("sum of elements is : " + sum);

}

}

**Output :**

sum of elements is : **32**

**Explanation :**

**reduce(0, (ans, i) -> ans + i); //here 0 is the initial value**

**0+12 = 12**

**12+20 = 32**

**- returns data as a distinct value**

**distinct()**

**Example :**

**Test.java**

**import** java.util.\*;

**import** java.util.stream.Collectors;

**public** **class** Test

{

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

{

List<Integer> list = Arrays.*asList*(1, 2, 3, 4, 1, 2, 3, 4, 4);

List<Integer> distintList =

list.stream()

.distinct()

.collect(Collectors.*toList*());

System.***out***.println("distinct values are : " + distintList);

}

}

**Output :**

distinct values are : [1, 2, 3, 4]

**- returns first value**

**findFirst()**

**Example :**

**Test.java**

**import** java.util.\*;

**public** **class** Main

{

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

{

List<String> list = Arrays.*asList*("apple", "banana", "watermelon");

Optional<String> fruit =

list.stream()

.findFirst();

**if** (fruit.isPresent())

{

System.***out***.println(fruit.get());

} **else**

{

System.***out***.println("no value");

}

}

}

**Output :**

apple

**- returns any value but mostly returns first value**

**findAny()**

**- It is designed for parallel stream**

**Example :**

**Test.java**

**import** java.util.\*;

**public** **class** Main

{

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

{

List<String> list = Arrays.*asList*("apple", "banana", "watermelon");

Optional<String> fruit = list.stream().findAny();

**if** (fruit.isPresent())

{

System.***out***.println(fruit.get());

} **else**

{

System.***out***.println("no value");

}

}

}

**Output :**

apple

**Example : (with parallel stream)**

**Test.java**

**import** java.util.\*;

**public** **class** Main

{

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

{

List<String> list = Arrays.*asList*("apple", "banana", "watermelon");

Optional<String> fruit = list.stream().parallel().findAny();

**if** (fruit.isPresent())

{

System.***out***.println(fruit.get());

} **else**

{

System.***out***.println("no value");

}

}

}

**Output :**

banana

**- returns boolean value if condition satisfied with any of the element**

**anyMatch()**

**Example :**

**Test.java**

**import** java.util.\*;

**public** **class** Main

{

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

{

List<String> list = Arrays.*asList*("apple", "banana", "watermelon");

**boolean** fruit = list.stream().anyMatch(e->e.contains("banana"));

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

}

}

**Output :**

true

**- returns boolean value if condition satisfied with all of the element**

**allMatch()**

**Example :**

**Test.java**

**import** java.util.\*;

**public** **class** Main

{

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

{

List<String> list = Arrays.*asList*("apple", "banana", "watermelon");

**boolean** fruit = list.stream().allMatch(e->e.contains("banana"));

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

}

}

**Output :**

false

**Example : (find min and max element)**

**max()**

**min()**

**Test.java**

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Test

{

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

{

List<Integer> list1 = **new** ArrayList<>();

list1.add(12);

list1.add(20);

list1.add(6);

list1.add(82);

list1.add(46);

**int** minValue = list1.stream().min((x, y) -> x.compareTo(y)).get();

System.***out***.println("min value is : " + minValue);

**int** maxValue = list1.stream().max((x, y) -> x.compareTo(y)).get();

System.***out***.println("max value is : " + maxValue);

}

}

**Output :**

**min** value is : **6**

**max** value is : **82**

**- returns data as a sorted**

**sorted()**

**Example : (sort arrayList)**

**Test.java**

**import** java.util.Arrays;

**import** java.util.List;

**import** java.util.stream.Collectors;

**public** **class** Test

{

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

{

Integer[] arr = **new** Integer[] { 11, 2, 5, 3, 2, 55, 32, 34 };

List<Integer> list = Arrays.*asList*(arr);

List<Integer> sortedList = list.stream()

.sorted()

.collect(Collectors.*toList*());

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

}

}

**Output :**

[2, 2, 3, 5, 11, 32, 34, 55]

**Example : (sort arrayList of Employee)**

**Employee.java**

**public** **class** Employee

{

**private** String name;

**private** **int** salary;

**public** Employee(String name,**int** salary)

{

**this**.name = name;

**this**.salary = salary;

}

**public** String getName()

{

**return** name;

}

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

{

**this**.name = name;

}

**public** **int** getSalary()

{

**return** salary;

}

**public** **void** setSalary(**int** salary)

{

**this**.salary = salary;

}

@Override

**public** String toString()

{

**return** "Employee [name=" + name + ", salary=" + salary + "]";

}

}

**Test.java**

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.Comparator;

**import** java.util.List;

**import** java.util.stream.Collectors;

**public** **class** Test

{

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

{

List<Employee> list = **new** ArrayList<Employee>();

list.add(**new** Employee("John", 4500));

list.add(**new** Employee("naresh", 2000));

list.add(**new** Employee("rohan", 1000));

List<Employee> sortedByName = list.stream()

.sorted(Comparator.*comparing*(Employee::getName))

.collect(Collectors.*toList*());

System.***out***.println("sorted by name : ");

sortedByName.forEach(s -> System.***out***.println(s));

System.***out***.println("----------------------");

List<Employee> sortedBySalary = list.stream()

.sorted(Comparator.*comparingInt*(Employee::getSalary))

.collect(Collectors.*toList*());

System.***out***.println("sorted by salary : ");

sortedBySalary.forEach(s -> System.***out***.println(s));

}

}

**Output :**

sorted by name :

Employee [name=John, salary=4500]

Employee [name=naresh, salary=2000]

Employee [name=rohan, salary=1000]

----------------------

sorted by salary :

Employee [name=rohan, salary=1000]

Employee [name=naresh, salary=2000]

Employee [name=John, salary=4500]

**Example : (sort HashMap by key)**

**Test.java**

**import** java.util.HashMap;

**import** java.util.LinkedHashMap;

**import** java.util.Map;

**import** java.util.Map.Entry;

**import** java.util.stream.Collectors;

**public** **class** Test

{

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

{

Map<String,Integer> hmap = **new** HashMap<>();

hmap.put("Tony", 87);

hmap.put("Bruce", 68);

hmap.put("Jackie", 98);

Map<String,Integer> sortedMap =

hmap.entrySet()

.stream()

.sorted(Entry.*comparingByKey*())

.collect(Collectors.*toMap*(

Entry::getKey,

Entry::getValue,

(e1,e2) -> e2,

**LinkedHashMap**::**new**

)

);

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

}

}

**Output :**

{Bruce=68, Jackie=98, Tony=87}

**Example : (sort HashMap by value)**

**Test.java**

**import** java.util.HashMap;

**import** java.util.LinkedHashMap;

**import** java.util.Map;

**import** java.util.Map.Entry;

**import** java.util.stream.Collectors;

**public** **class** Test

{

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

{

Map<String,Integer> hmap = **new** HashMap<>();

hmap.put("Tony", 87);

hmap.put("Bruce", 68);

hmap.put("Jackie", 98);

Map<String,Integer> sortedMap =

hmap.entrySet()

.stream()

.sorted(Entry.*comparingByValue*())

.collect(Collectors.*toMap*(

Entry::getKey,

Entry::getValue,

(e1,e2) -> e2,

**LinkedHashMap**::**new**

)

);

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

}

}

**Output :**

{Bruce=68, Tony=87, Jackie=98}

**combination of methods**

**Example : (find sum of sqaure of elements less than 20)**

**Test.java**

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Test

{

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

{

List<Integer> list1 = **new** ArrayList<>();

list1.add(12);

list1.add(20);

list1.add(6);

list1.add(82);

list1.add(46);

**int** sum = list1.stream().filter(i -> i < 20).map(i -> i \* i).reduce(0, (ans, i) -> ans + i);

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

}

}

**Output :**

sum is : **180** //(12²)+(6²)=180

**Collectors class**

**- helps to save result data into collections like list, set**

**Example : (find values less than 20)**

**Test.java**

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.stream.Collectors;

**public** **class** Test

{

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

{

List<Integer> list1 = **new** ArrayList<>();

list1.add(12);

list1.add(20);

list1.add(6);

List<Integer> list2 = list1.stream().filter(i -> i < 20).collect(Collectors.*toList*());

System.***out***.println("elements less than 20 : " + list2);

}

}

**Output :**

elements less than 20 are : [**12**, **6**]

**groupingBy()**

**Example : (find group of employees by pincode / salary/ salary and name)**

**Employee.java**

**public** **class** Employee

{

**private** String name;

**private** **int** salary;

**private** Address address;

**public** Employee(String name, **int** salary, Address address)

{

**super**();

**this**.name = name;

**this**.salary = salary;

**this**.address = address;

}

**public** String getName()

{

**return** name;

}

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

{

**this**.name = name;

}

**public** **int** getSalary()

{

**return** salary;

}

**public** **void** setSalary(**int** salary)

{

**this**.salary = salary;

}

**public** Address getAddress()

{

**return** address;

}

**public** **void** setAddress(Address address)

{

**this**.address = address;

}

}

**Address.java**

**public** **class** Address

{

**private** String country;

**private** **int** pincode;

**public** Address(String country, **int** pincode)

{

**super**();

**this**.country = country;

**this**.pincode = pincode;

}

**public** String getCountry()

{

**return** country;

}

**public** **void** setCountry(String country)

{

**this**.country = country;

}

**public** **int** getPincode()

{

**return** pincode;

}

**public** **void** setPincode(**int** pincode)

{

**this**.pincode = pincode;

}

}

**Main.java**

**import** java.util.\*;

**import** java.util.stream.Collectors;

**public** **class** Main

{

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

{

Employee e1=**new** Employee("hitesh",20000,**new** Address("India",421));

Employee e2=**new** Employee("mahesh",10000,**new** Address("China",320));

Employee e3=**new** Employee("ramesh",30000,**new** Address("Japan",280));

List<Employee> list=**new** ArrayList<>();

list.add(e1);

list.add(e2);

list.add(e3);

Map<Integer, List<Employee>> empByPincode=

list.stream()

.collect(Collectors.*groupingBy*(a->a.getAddress().getPincode()));

System.***out***.println("employees by pincode : "+empByPincode);

Map<Integer, List<Employee>> empBySalary=

list.stream()

.collect(Collectors.*groupingBy*(a->a.getSalary()));

System.***out***.println("employees by salary : "+empBySalary);

Map<Integer, Map<String, List<Employee>>> empBySalaryAndName=

list.stream()

.collect(Collectors.*groupingBy*(a->a.getSalary(),Collectors.*groupingBy*(a->a.getName())));

System.***out***.println("employees by salary and name : "+empBySalaryAndName);

}

}

**Output :**

**employees by pincode :** {**320**=[Employee@1761e840], **421**=[Employee@6c629d6e], **280**=[Employee@5ecddf8f]}

**employees by salary :** {**30000**=[Employee@5ecddf8f], **10000**=[Employee@1761e840], **20000**=[Employee@6c629d6e]}

**employees by salary and name :** {**30000**={**ramesh**=[Employee@5ecddf8f]}, **10000**={**mahesh**=[Employee@1761e840]}, **20000**={**hitesh**=[Employee@6c629d6e]}}

**static and default methods**

**- static and default methods were not allowed before java 8 in interface**

**- but now it is allowed**

**Example :**

**Test.java**

**public** **class** Test **implements** Sample

{

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

{

Test t = **new** Test();

t.m1("welcome");

t.m2();

Sample.*m3*();

}

@Override

**public** **void** m1(String msg)

{

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

}

}

**Sample.java**

**public** **interface** Sample

{

**void** m1(String msg);

**default** **void** m2()

{

System.***out***.println("this is default method");

}

**static** **void** m3()

{

System.***out***.println("this is static method");

}

}

**Output :**

welcome

**this is default method**

**this is static method**

**Question :**

1. **Why we need default and static methods in interface?**

* **We have abstract methods in interface**
* **But those methods we must override in implementation class**

**Example :**

**Main.java**

**public** **class** Main

{

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

{

Developer webDev = **new** WebDeveloper();

webDev.developer();

Developer androidDev = **new** AndroidDeveloper();

androidDev.developer();

}

}

**Developer.java**

**public** **interface** Developer

{

**void** developer();

}

**WebDeveloper.java**

**public** **class** WebDeveloper **implements** Developer

{

@Override

**public** **void** developer()

{

System.***out***.println("I am web developer");

}

}

**AndroidDeveloper.java**

**public** **class** AndroidDeveloper **implements** Developer

{

@Override

**public** **void** developer()

{

System.***out***.println("I am android developer");

}

}

**Output :**

I am web developer

I am android developer

* **So In below example webDeveloper wants callDatabase() method but androidDeveloper don’t want it (don’t want to ovverride like normal abstract methodes we do)**
* **In this case we can use default methods**

**Example :**

**Main.java**

**public** **class** Main

{

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

{

Developer webDev = **new** WebDeveloper();

webDev.developer();

webDev.callDatabase();

Developer androidDev = **new** AndroidDeveloper();

androidDev.developer();

}

}

**Developer.java**

**public** **interface** Developer

{

**void** developer();

**default** **void** callDatabase()

{

System.***out***.println("calling database");

}

}

**WebDeveloper.java**

**public** **class** WebDeveloper **implements** Developer

{

@Override

**public** **void** developer()

{

System.***out***.println("I am web developer");

}

}

**AndroidDeveloper.java**

**public** **class** AndroidDeveloper **implements** Developer

{

@Override

**public** **void** developer()

{

System.***out***.println("I am android developer");

}

}

**Output :**

I am web developer

calling database

I am android developer

* **default method we can override also in both WebDeveloper and AndroidDeveloper class**

**Example :**

**Main.java**

**public** **class** Main

{

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

{

Developer webDev = **new** WebDeveloper();

webDev.developer();

webDev.callDatabase();

Developer androidDev = **new** AndroidDeveloper();

androidDev.developer();

}

}

**Developer.java**

**public** **interface** Developer

{

**void** developer();

**default** **void** callDatabase()

{

System.***out***.println("calling database");

}

}

**WebDeveloper.java**

**public** **class** WebDeveloper **implements** Developer

{

**public** **void** developer()

{

System.***out***.println("I am web developer");

}

**public** **void** callDatabase()

{

System.***out***.println("calling database web");

}

}

**AndroidDeveloper.java**

**public** **class** AndroidDeveloper **implements** Developer

{

**public** **void** developer()

{

System.***out***.println("I am android developer");

}

}

**Output :**

I am web developer

calling database web

I am android developer

**- But if we don’t want to override callDatabase() method at all then use static method**

**Example :**

**Main.java**

**public** **class** Main

{

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

{

Developer webDev = **new** WebDeveloper();

webDev.developer();

Developer.*callDatabase*();

Developer androidDev = **new** AndroidDeveloper();

androidDev.developer();

}

}

**Developer.java**

**public** **interface** Developer

{

**void** developer();

**static** **void** callDatabase()

{

System.***out***.println("calling database");

}

}

**WebDeveloper.java**

**public** **class** WebDeveloper **implements** Developer

{

**public** **void** developer()

{

System.***out***.println("I am web developer");

}

**public** **void** callDatabase()

{

System.***out***.println("calling database web");

}

}

**AndroidDeveloper.java**

**public** **class** AndroidDeveloper **implements** Developer

{

**public** **void** developer()

{

System.***out***.println("I am android developer");

}

}

**Output :**

I am web developer

calling database

I am android developer