## **Java**



## ***Encapsulation and Abstraction***

## 

* Encapsulation is more about "How" to achieve a functionality
* Abstraction is more about "What" a class can do.

A simple example to understand this difference is a mobile phone. Where the complex logic in the circuit board is encapsulated in a touch screen, and the interface is provided to abstract it out.

## ***Memory Allocation in Java***

The JVM divided the memory into following sections.

1. Heap
2. Stack
3. Code
4. Static

This division of memory is required for its effective management.

* The **code** section contains your **bytecode**.
* The **Stack** section of memory contains **methods, local variables, and reference variables.**
* The **Heap** section contains **Objects** (may also contain reference variables).
* The **Static** section contains **Static data/methods**.

## ***Class loader and explain the types of class loader***

Class loaders are the part of the Java Runtime Environment that dynamically loads Java classes into the Java virtual machine. It is responsible for locating libraries, reading there content and loading the classes contained within the libraries. When JVM is started three class loaders are used  
  
1. **Bootstrap class loader**  
  
2. **Extensions class loader**  
  
3. **System class loader**  
**Bootstrap class loader** loads the core java libraries. It is written in native code. The bootstrap class loader is responsible for loading key java classes like java.lang.Object and other runtime code into memory. The runtime classes are packaged inside jre/lib/rt.jar file.   
  
**Extensions class loader** loads the code in the extension directories. It is implemented by ExtClassLoader class.  
  
**System class loader** the code found on the java.class.path which map to the system class path variables. It is implemented by AppClassLoader class. All user classes by default are load by the system class loader.

## ***ArrayList and LinkedList:***

ArrayList and LinkedList. Which of the two List implementations you use depends on

your specific needs. If you need to support random access, without inserting or removing elements from any place other than the end, then ArrayList offers the optimal collection.If, however, you need to frequently add and remove elements from the middle of the list and only access the list elements sequentially, then LinkedList offers the better implementation.

## *Fail-Fast Iterator:*

As the name sounds the Iterator will fail as soon as the it encounters a change in the collection. What ever the change it may be adding, update or removal of any object in the collection will throw the **ConcurrentModificationException**

## ***Fail-Safe Iterator:***

Whereas the fail-safe iterator will not throw any exception when the collection such as **CopyOnWriteArrayList and ConcurrentHashMap** is modified. As it iterates on the copy of the collection.

## ***Java ArrayList and ConcurrentModificationException:***

ArrayList is one of the basic implementations of List interface and

it’s part of Java Collections Framework. We can use iterator to traverse through ArrayList elements.

Let’s check a sample program of ArrayList.

**ConcurrentListExample.java**

package com;

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

import java.util.concurrent.CopyOnWriteArrayList;

public class **ConcurrentListExample** {

public static void main(String[] args) {

List<String> list = new ArrayList<>();

list.add("1");

list.add("2");

list.add("3");

list.add("4");

list.add("5");

// get the iterator

Iterator<String> it = list.iterator();

//manipulate list while iterating

while(it.hasNext()){

System.out.println("list is:"+list);

String str = it.next();

System.out.println(str);

**if(str.equals("2"))list.remove("5"); // throw ConcurrentModificationException**

//below code don't throw ConcurrentModificationException

//because it doesn't change modCount variable of list

if(str.equals("4")) list.set(1, "4");

}

}

}

When we run above program, we get java.util.ConcurrentModificationException as soon as the ArrayList is modified.

It happens because **ArrayList iterator is fail-fast by design**. What it means is that once the iterator is created,

if the **ArrayList** is modified, it throws **ConcurrentModificationException**.

## ***Callable Statement:***

The **CallableStatement** of JDBC API is used to call a stored procedure from Java Program.

**Stored procedure** is a set of SQL statements to be executed to perform a specific task on a database. Stored procedures are beneficial when we are dealing with multiple tables with complex scenario and rather than sending multiple queries to the database,

**JDBC CallableStatement – Stored Procedure IN parameter example**:

Code snippets to show you how to call a Oracle stored procedure via JDBC CallableStatement, and how to pass IN parameters from Java to stored procedure.

//insertDBUSER is stored procedure

String insertStoreProc = "{call insertDBUSER(?,?,?,?)}";

callableStatement = dbConnection.prepareCall(insertStoreProc);

callableStatement.setInt(1, 1000);

callableStatement.setString(2, "Sanjoy");

callableStatement.setString(3, "Network Admin");

callableStatement.setDate(4, getCurrentDate());

callableStatement.executeUpdate();

## ***Stored Procedure:***

A stored procedure in Oracle database. Later, calls it via JDBC.

CREATE OR REPLACE PROCEDURE insertDBUSER(

p\_userid IN DBUSER.USER\_ID%TYPE,

p\_username IN DBUSER.USERNAME%TYPE,

p\_createdby IN DBUSER.CREATED\_BY%TYPE,

p\_date IN DBUSER.CREATED\_DATE%TYPE)

IS

BEGIN

INSERT INTO DBUSER ("USER\_ID", "USERNAME", "CREATED\_BY", "CREATED\_DATE")

VALUES (p\_userid, p\_username,p\_createdby, p\_date);

COMMIT;

END;

/

## ***JDBC CallableStatement – Stored Procedure OUT parameter example:***

## ***Code snippets:***

//getDBUSERByUserId is a stored procedure

String getDBUSERByUserIdSql = "{call getDBUSERByUserId(?,?,?,?)}";

callableStatement = dbConnection.prepareCall(getDBUSERByUserIdSql);

callableStatement.setInt(1, 10);

callableStatement.registerOutParameter(2, java.sql.Types.VARCHAR);

callableStatement.registerOutParameter(3, java.sql.Types.VARCHAR);

callableStatement.registerOutParameter(4, java.sql.Types.DATE);

// execute getDBUSERByUserId store procedure

callableStatement.executeUpdate();

String userName = callableStatement.getString(2);

String createdBy = callableStatement.getString(3);

Date createdDate = callableStatement.getDate(4);

## ***Stored Procedure***

A stored procedure in Oracle database, with IN and OUT parameters. Later, calls it via JDBC.

CREATE OR REPLACE PROCEDURE getDBUSERByUserId(

p\_userid IN DBUSER.USER\_ID%TYPE,

o\_username OUT DBUSER.USERNAME%TYPE,

o\_createdby OUT DBUSER.CREATED\_BY%TYPE,

o\_date OUT DBUSER.CREATED\_DATE%TYPE)

IS

BEGIN

SELECT USERNAME , CREATED\_BY, CREATED\_DATE

INTO o\_username, o\_createdby, o\_date

from DBUSER WHERE USER\_ID = p\_userid;

END;

/

## ***JDBC CallableStatement – Stored Procedure CURSOR example:***

//getDBUSERCursor is a stored procedure

String getDBUSERCursorSql = "{call getDBUSERCursor(?,?)}";

callableStatement = dbConnection.prepareCall(getDBUSERCursorSql);

callableStatement.setString(1, "Sanjoy");

callableStatement.registerOutParameter(2, OracleTypes.CURSOR);

// execute getDBUSERCursor store procedure

callableStatement.executeUpdate();

// get cursor and cast it to ResultSet

rs = (ResultSet) callableStatement.getObject(2);

// loop it like normal

while (rs.next()) {

String userid = rs.getString("USER\_ID");

String userName = rs.getString("USERNAME");

}

## ***Stored Procedure***

A Oracle stored procedure, with one IN and one OUT CURSOR parameter. Later, calls it via JDBC.

CREATE OR REPLACE PROCEDURE getDBUSERCursor(

p\_username IN DBUSER.USERNAME%TYPE,

c\_dbuser OUT SYS\_REFCURSOR)

IS

BEGIN

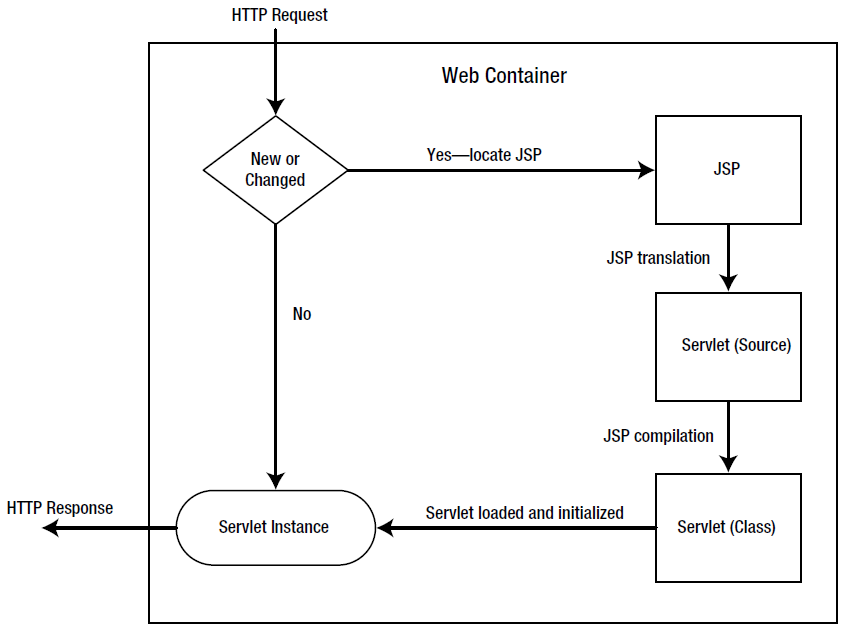
OPEN c\_dbuser FOR

SELECT \* FROM DBUSER WHERE USERNAME LIKE p\_username || '%';

END;

/

## ***JSP life cycle***



The life cycle of a JSP page can be split into four phases: **translation**, **initialization, execution, and finalization**.

## ***Translation***

* a request is first made for a JSP page
* first it checks if the JSP page is new or has changed
* if so, the JSP engine will examine the JSP file to check that it?s correctly formed and that the JSP syntax is correct
* if so, the JSP Engine will translate the JSP page into its page implementation class, which takes the form of a standard Java servlet (the servlet will extend HttpJspBase which extends HttpServlet)
* the page implementation class will be compiled into a class file by the JSP engine and will be ready for use
* If it was not changed the JSP file since its last translation, the servlet instance created before is used.

## ***Initialization***

The JSP engine loads the generated class file and creates an instance of the servlet. It is called jspInit() method which has an identical behavior to the standard servlet init() method.

jspInit() is automatically generated during the translation phase, but it's possible to override this method in the JSP page by using a declaration. The method can be used for initializing application-level variables or resources, for example:

<%! AppVar appvar = null; %>  
<%!  
  public void jspInit() {  
    try {  
      appvar = initAppVar(...);  
    } catch (Exception e){  
      //handle exception  
    }  
  }  
%>

## ***Execution***

The initial request can be serviced. For each request, the web container calls in a separate thread the \_jspService() method of the implementation servlet.

The \_jspService() method provides all the functionality for handling a request and returning a response to the client. All the scriptlets and expressions end up inside this method, in the order in which they were declared inside the JSP page.

Notice that JSP declarations and directives aren't included inside this method because they apply to the entire page, not just to a single request, and therefore exist outside the method.

The \_jspService() method may not be overridden in the JSP page.

## ***Finalization***

It is called the jspDestroy() method which has the same behavior as destroy() method found in a standard servlet. This method can be overridden in the JSP page.

For example, to release the application resource you opened inside the jspInit() method, you would use the following:

<%!  
public void jspDestroy() {  
  try {  
    appVar.release();  
  } catch (Exception e){}  
  appVar = null;  
}  
%>

## ***Comparable and Comparator***

## ***Comparable interface:***

The **Comparable** interface, in the java.lang package, is for when a class has a natural

ordering. Given a collection of objects of the same type, the interface allows you to order the

collection into that natural ordering.

The **compareTo()** method compares the current instance with an element passed in as an

argument. If the current instance comes before the argument in the ordering, a negative

value is returned. If the current instance comes after, then a positive value is returned.

Otherwise, zero is returned.

 Sorts the specified list into ascending order, according to the natural ordering of its elements.

## ***Comparator interface:***

Sorts the specified list according to the order induced by the specified comparator

The return values of the **compare()** method of Comparator are similar to the **compareTo()** method of **Comparable**

**Comparable and Comparator** in Java are very useful for sorting collection of objects. Java provides some inbuilt methods to sort primitive types array or Wrapper classes array or list.

**JavaSorting.java:**

package com.sort;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.Collections;

import java.util.List;

public class JavaSorting {

public static void main(String[] args) {

//sort primitives array like int array

int[] intArr = {5,9,1,10};

Arrays.sort(intArr);

System.out.println(Arrays.toString(intArr));

//sorting String array

String[] strArr = {"A", "C", "B", "Z", "E"};

Arrays.sort(strArr);

System.out.println(Arrays.toString(strArr));

//sorting list of objects of Wrapper classes

List<String> strList = new ArrayList<String>();

strList.add("A");

strList.add("C");

strList.add("B");

strList.add("Z");

strList.add("E");

Collections.sort(strList);

for(String str: strList) System.out.print(" "+str);

}

}

Now let’s try to sort an array of objects

**Employee.java:**

package com.sort;

public class Employee {

private int id;

private String name;

private int age;

private long salary;

public int getId() {

return id;

}

public String getName() {

return name;

}

public int getAge() {

return age;

}

public long getSalary() {

return salary;

}

public Employee(int id, String name, int age, int salary) {

this.id = id;

this.name = name;

this.age = age;

this.salary = salary;

}

@Override

//this is overridden to print the user-friendly information about the Employee

public String toString() {

return "[id=" + this.id + ", name=" + this.name + ", age=" + this.age + ", salary=" +

this.salary + "]";

}

}

**Here is the code I used to sort the array of Employee objects.**

//sorting object array

Employee[] empArr = new Employee[4];

empArr[0] = new Employee(15, " Bikash ", 20, 8000);

empArr[1] = new Employee(20, " Abshishek ", 25, 10000);

empArr[2] = new Employee(6, "Ganesh", 30, 15000);

empArr[3] = new Employee(2, "Debopriyo", 12, 40000);

//sorting employees array using Comparable interface implementation

Arrays.sort(empArr);

System.out.println("Default Sorting of Employees list:\n"+Arrays.toString(empArr));

**Throwing Error.**

Java provides **Comparable** interface which should be implemented by any custom class if we want to use Arrays or Collections sorting methods.

The **Comparable** interface has **compareTo(T obj)** method which is used by sorting methods,

After implementing Comparable [interface](https://www.journaldev.com/1601/interface-in-java) in Employee class, here is the resulting Employee class.

**Employee.java:**

package com.sort;

import java.util.Comparator;

**public class Employee implements Comparable<Employee>** {

private int id;

private String name;

private int age;

private long salary;

public int getId() {

return id;

}

public String getName() {

return name;

}

public int getAge() {

return age;

}

public long getSalary() {

return salary;

}

public Employee(int id, String name, int age, int salary) {

this.id = id;

this.name = name;

this.age = age;

this.salary = salary;

}

@Override

public int compareTo(Employee emp) {

//let's sort the employee based on an id in ascending order

//returns a negative integer, zero, or a positive integer as this employee id

//is less than, equal to, or greater than the specified object.

return (this.id - emp.id);

}

@Override

//this is required to print the user-friendly information about the Employee

public String toString() {

return "[id=" + this.id + ", name=" + this.name + ", age=" + this.age + ", salary=" +

this.salary + "]";

}

}

Now when we execute the above snippet for Arrays sorting of Employees and print it, here is the output.

**Default Sorting of Employees list:**

**Output is based on ID**.

As you can see that Employees array is sorted by id in ascending order

But, in most real-life scenarios, we want sorting based on different parameters. For example, as a CEO, I would like to sort the employees based on Salary, an HR would like to sort them based on the age. **This is the situation where we need to use Java Comparator interface** because **Comparable.compareTo(Object o) method** implementation can provide default sorting and we can’t change it dynamically. Whereas with **Comparator, we can define multiple methods with different ways of sorting and then chose the sorting method based on our requirements.**

**So if we want to sort the collections based on multiple criteria/fields then we have to use Comparator only.**

**Employee.java:**

package com;

public class Employee {

int id;

String name,address;

public Employee(int id, String name, String address) {

super();

this.id = id;

this.name = name;

this.address = address;

}

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 getAddress() {

return address;

}

public void setAddress(String address) {

this.address = address;

}

public String toString() {

return "Employee [id=" + id + ", name=" + name + ", address=" + address

+ "]";

}

}

**EmployeeByID.java:**

package com;

import java.util.Comparator;

import java.util.function.Function;

import java.util.function.ToDoubleFunction;

import java.util.function.ToIntFunction;

import java.util.function.ToLongFunction;

**public class EmployeeByID implements Comparator**{

public int **compare**(Object arg0, Object arg1) {

Employee e1=(Employee)arg0;

Employee e2=(Employee)arg1;

return e1.id-e2.id;

}

public static Comparator comparing(Function arg0) {

// TODO Auto-generated method stub

return null;

}

public static Comparator comparing(Function arg0, Comparator arg1) {

// TODO Auto-generated method stub

return null;

}

public static Comparator comparingDouble(ToDoubleFunction arg0) {

// TODO Auto-generated method stub

return null;

}

public static Comparator comparingInt(ToIntFunction arg0) {

// TODO Auto-generated method stub

return null;

}

public static Comparator comparingLong(ToLongFunction arg0) {

// TODO Auto-generated method stub

return null;

}

public static Comparator naturalOrder() {

// TODO Auto-generated method stub

return null;

}

public static Comparator nullsFirst(Comparator arg0) {

// TODO Auto-generated method stub

return null;

}

public static Comparator nullsLast(Comparator arg0) {

// TODO Auto-generated method stub

return null;

}

public static Comparator reverseOrder() {

// TODO Auto-generated method stub

return null;

}

public Comparator reversed() {

// TODO Auto-generated method stub

return null;

}

public Comparator thenComparing(Comparator arg0) {

// TODO Auto-generated method stub

return null;

}

public Comparator thenComparing(Function arg0) {

// TODO Auto-generated method stub

return null;

}

public Comparator thenComparing(Function arg0, Comparator arg1) {

// TODO Auto-generated method stub

return null;

}

public Comparator thenComparingDouble(ToDoubleFunction arg0) {

// TODO Auto-generated method stub

return null;

}

public Comparator thenComparingInt(ToIntFunction arg0) {

// TODO Auto-generated method stub

return null;

}

public Comparator thenComparingLong(ToLongFunction arg0) {

// TODO Auto-generated method stub

return null;

}

}

**EmployeeByName.java**

package com;

import java.util.Comparator;

import java.util.function.Function;

import java.util.function.ToDoubleFunction;

import java.util.function.ToIntFunction;

import java.util.function.ToLongFunction;

**public class EmployeeByName implements Comparator**{

public int **compare**(Object arg0, Object arg1) {

Employee e1=(Employee)arg0;

Employee e2=(Employee)arg1;

return e1.name.compareTo(e2.name);

}

public static Comparator comparing(Function arg0) {

// TODO Auto-generated method stub

return null;

}

public static Comparator comparing(Function arg0, Comparator arg1) {

// TODO Auto-generated method stub

return null;

}

public static Comparator comparingDouble(ToDoubleFunction arg0) {

// TODO Auto-generated method stub

return null;

}

public static Comparator comparingInt(ToIntFunction arg0) {

// TODO Auto-generated method stub

return null;

}

public static Comparator comparingLong(ToLongFunction arg0) {

// TODO Auto-generated method stub

return null;

}

public static Comparator naturalOrder() {

// TODO Auto-generated method stub

return null;

}

public static Comparator nullsFirst(Comparator arg0) {

// TODO Auto-generated method stub

return null;

}

public static Comparator nullsLast(Comparator arg0) {

// TODO Auto-generated method stub

return null;

}

public static Comparator reverseOrder() {

// TODO Auto-generated method stub

return null;

}

public Comparator reversed() {

// TODO Auto-generated method stub

return null;

}

public Comparator thenComparing(Comparator arg0) {

// TODO Auto-generated method stub

return null;

}

public Comparator thenComparing(Function arg0) {

// TODO Auto-generated method stub

return null;

}

public Comparator thenComparing(Function arg0, Comparator arg1) {

// TODO Auto-generated method stub

return null;

}

public Comparator thenComparingDouble(ToDoubleFunction arg0) {

// TODO Auto-generated method stub

return null;

}

public Comparator thenComparingInt(ToIntFunction arg0) {

// TODO Auto-generated method stub

return null;

}

public Comparator thenComparingLong(ToLongFunction arg0) {

// TODO Auto-generated method stub

return null;

}

}

**EmpMain.java:**

package com;

import java.util.\*;

public class EmpMain {

public static void main(String[] args) {

Employee e1=new Employee(1,"Shanka","Tollygunge");

Employee e2=new Employee(2,"Bapan","Selimpur");

Employee e3=new Employee(3,"Asim","Jadavpur");

Employee e4=new Employee(4,"Pritam","Dumdum");

ArrayList<Employee> list=new ArrayList<Employee>();

list.add(e1);

list.add(e4);

list.add(e2);

list.add(e3);

System.out.println("Before sorting the List based on Id::"+list);

Collections.sort(list,new EmployeeByID());

System.out.println("After sorting the List based on Id::"+list);

list.clear();

list.add(e1);

list.add(e4);

list.add(e2);

list.add(e3);

System.out.println("Before sorting the List based on Name::"+list);

Collections.sort(list,new EmployeeByName());

System.out.println("After sorting the List based on Name::"+list);

}

}

**OutPut:**

**Before sorting the List based on Id::[Employee [id=1, name=Shanka, address=Tollygunge], Employee [id=4, name=Pritam, address=Dumdum], Employee [id=2, name=Bapan, address=Selimpur], Employee [id=3, name=Asim, address=Jadavpur]]**

**After sorting the List based on Id::[Employee [id=1, name=Shanka, address=Tollygunge], Employee [id=2, name=Bapan, address=Selimpur], Employee [id=3, name=Asim, address=Jadavpur], Employee [id=4, name=Pritam, address=Dumdum]]**

**Before sorting the List based on Name::[Employee [id=1, name=Shanka, address=Tollygunge], Employee [id=4, name=Pritam, address=Dumdum], Employee [id=2, name=Bapan, address=Selimpur], Employee [id=3, name=Asim, address=Jadavpur]]**

**After sorting the List based on Name::[Employee [id=3, name=Asim, address=Jadavpur], Employee [id=2, name=Bapan, address=Selimpur], Employee [id=4, name=Pritam, address=Dumdum], Employee [id=1, name=Shanka, address=Tollygunge]]**

## ***Object level Locking vs. Class level Locking in Java***

In java there are two types of locks:

**Class Level**

Class level locking prevents multiple threads to enter in synchronized block in any of all available instances on runtime. This means if in runtime there are 100 instances of DemoClass, then only one thread will be able to execute demoMethod() in any one of instance at a time, and all other instances will be locked for other threads.

**Object Level**

In case of Static methods the lock is always checked on class but in case of instance methods the lock is always checked on object.

Object level locking is mechanism when you want to synchronize a non-static method or non-static code block such that only one thread will be able to execute the code block on given instance of the class. This should always be done to make instance level data thread safe

**Example**:

show1() is non static and show() is static. Now, show() is called by class name (or by object) and show1() is called by object, then both methods can accessed simultaneously by two threads.

class Shared{

static int x;

static synchronized void show(String s,int a){

x=a;

System.out.println("Starting in method "+s+" "+x);

try{

Thread.sleep(2000);

}

catch(Exception e){ }

System.out.println("Ending from method "+s+" "+x);

}

synchronized void show1(String s,int a){

x=a;

System.out.println("Starting show1 "+s);

try{

Thread.sleep(2000);

}

catch(Exception e){ }

System.out.println("Ending from show1 "+s);

}

}

class CustomThread extends Thread{

Shared s;

public CustomThread(Shared s,String str){

super(str);

this.s=s;

start();

}

public void run(){

Shared.show(Thread.currentThread().getName(),10);

}

}

class CustomThread1 extends Thread{

Shared s;

public CustomThread1(Shared s,String str){

super(str);

this.s=s;

start();

}

public void run(){

s.show1(Thread.currentThread().getName(),20);

}

}

public class RunSync {

public static void main(String[] args) {

Shared sh=new Shared();

CustomThread t1=new CustomThread(sh,"one");

CustomThread1 t2=new CustomThread1(sh,"two");

}

}

Output:

Starting in method one 10

Starting show1 two

Ending from method one 20

Ending from show1 two

**Synchronization** refers to multi-threading. A synchronized block of code can only be executed by one thread at a time.

Java supports multiple threads to be executed. This may cause two or more threads to access the same fields or objects. Synchronization is a process which keeps all concurrent threads in execution to be in synch. Synchronization avoids memory consistence errors caused due to inconsistent view of shared memory. When a method is declared as synchronized; the thread holds the monitor for that method’s object If another thread is executing the synchronized method, your thread is blocked until that thread releases the monitor.

Synchronization in java is achieved using **synchronized** keyword. **You can use synchronized keyword in your class on defined methods or blocks. Keyword can not be used with variables or attributes in class definition.**

**Object level locking**

**Object level locking** is mechanism when you want to synchronize a non-static method or non-static code block such that only one thread will be able to execute the code block on given instance of the class. This should always be done to make instance level data thread safe. This can be done as below :

|  |
| --- |
| public class DemoClass  {      public synchronized void demoMethod(){}  }    or    public class DemoClass  {      public void demoMethod(){          synchronized (this)          {              //other thread safe code          }      }  }    or    public class DemoClass  {      private final Object lock = new Object();      public void demoMethod(){          synchronized (lock)          {              //other thread safe code          }      }  } |

**Class level locking**

**Class level locking** prevents multiple threads to enter in synchronized block in any of all available instances on runtime. This means if in runtime there are 100 instances of  DemoClass, then only one thread will be able to execute demoMethod() in any one of instance at a time, and all other instances will be locked for other threads. This should always be done to make static data thread safe.

|  |
| --- |
| public class DemoClass  {      public synchronized static void demoMethod(){}  }    or    public class DemoClass  {      public void demoMethod(){          synchronized (DemoClass.class)          {              //other thread safe code          }      }  }    or    public class DemoClass  {      private final static Object lock = new Object();      public void demoMethod(){          synchronized (lock)          {              //other thread safe code          }      }  } |

**Difference between synchronized keyword and synchronized block**

When we use **synchronized** **keyword with a method**, it acquires a lock in the object for the whole method. It means that no other thread can use any synchronized method until the current thread, which has invoked it's synchronized method, has finished its execution.

**synchronized block** acquires a lock in the object only between parentheses after the synchronized keyword. This means that no other thread can acquire a lock on the locked object until the synchronized block exits. But other threads can access the rest of the code of the method.

## ***String Concept:***

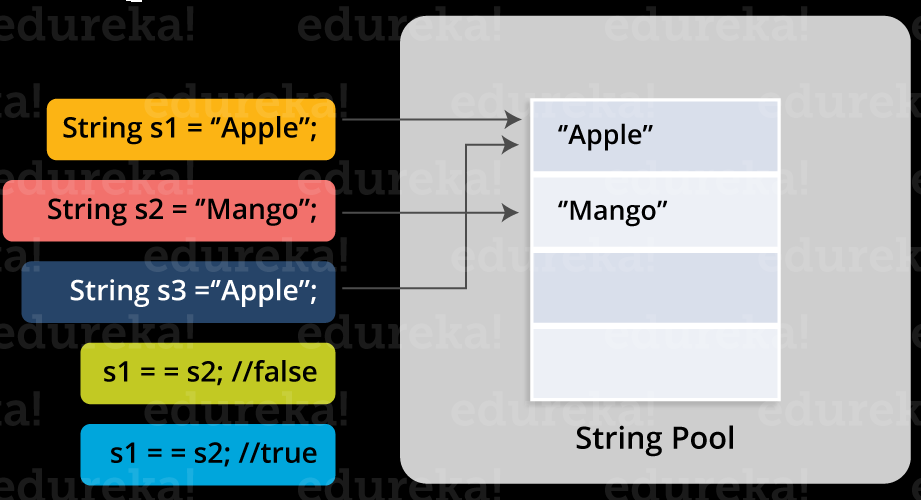
There are two ways to create a String object:

* **By string literal :** Java String literal is created by using double quotes.  
  For Example: String s=“Welcome”;
* **By new keyword :** Java String is created by using a keyword “new”.  
  For example: String s=new String(“Welcome”);    
  It creates two objects (in String pool and in heap) and one reference variable where the variable ‘s’ will refer to the object in the heap.

Now, let us understand the concept of Java String pool.

## ***Java String Pool:***

Java String pool refers to collection of Strings which are stored in heap memory. In this, whenever a new object is created, String pool first checks whether the object is already present in the pool or not. If it is present, then same reference is returned to the variable else new object will be created in the String pool and the respective reference will be returned. Refer to the diagrammatic representation for better understanding:



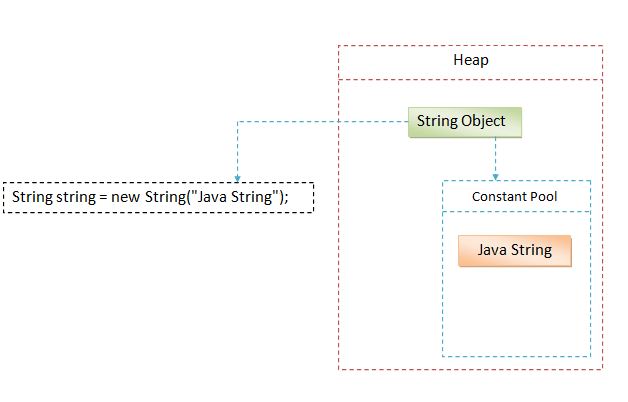
In the above image, two Strings are created using literal i.e “Apple” and “Mango”. Now, when third String is created with the value “Apple”, instead of creating a new object, the already present object reference is returned. That’s the reason Java String pool came into the picture.

Before we go ahead, One key point I would like to add that unlike other data types in Java, Strings are immutable. By immutable, we mean that Strings are constant, their values cannot be changed after they are created. Because String objects are immutable, they can be shared.

String is immutable in java programming. Once you crated an String object, you can not modify it’s content again on the same object.  
  
Let’s understand it by creating a simple String object.

**String string = new String("Java String");**

Let’s discuss what’s happening inside when we are creating a new string in java and it’s memory structure.

[](http://www.ashtpoint.com/wp-content/uploads/2017/07/what-is-string-in-java.jpg)

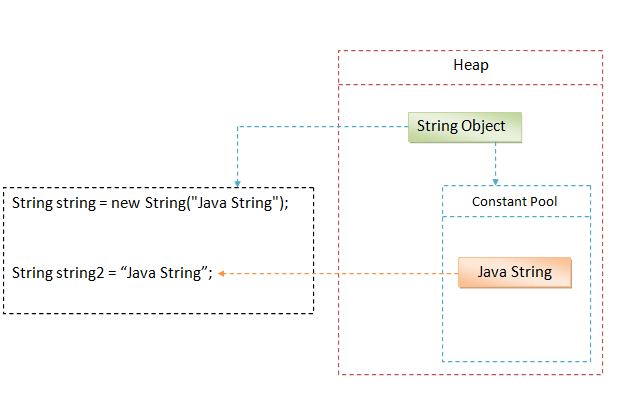
Look at the above diagram how String object is being created in Heap area.  
  
Generally we create lot of string objects in our project or program during run time. So to make this process faster, Java though let’s create a separate area where we can create String objects faster to make our program response time faster. This separate space is called “**String Constant Pool**” or we can say “**String Literal Pool**”.  
  
**String constant pool basically stores the String literals which can be reused by other String objects**.  
  
Let’s take the above syntax to understand how String is created.

* First, we created a reference variable of type String which is “string”. This object is empty for now.
* Then we created a new String by simple using the “new” keyword.
* We passed the string “Java String” to the constructor of String class.
* Now JVM, first will look for the existing String “Java String” in constant pool. JVM finds that there no such string created before in Constant Pool.
* So Java will create a new String in normal Heap (Non pool) memory area and reference of it will be passed to reference variable of String type.
* And at the same time, String literal “Java String” will be placed in the String constant pool.
* So basically two objects will be created. First it will create a string in constant pool and then other will be created in the Heap area.
* After that reference of the newly created String will be passed to the String reference variable.
* Now let’s modify the above example and create another String object and see what’s happens now.

**String string = new String("Java String");**

**String string2 = "Java String";**

Look at the below diagram what’s happening internally.

[](http://www.ashtpoint.com/wp-content/uploads/2017/07/String-in-java.jpg)

What’s happening here.

1. Now, JVM will first look for the String literal “Java String” in string constant pool.
2. So there will no new object will be created for the “string2”.
3. JVM will simple pass the reference of the String literal already created in the String constant pool.
4. So there will still be only two objects in memory.

If String is immutable then what’s happening in below example.

**String string = "Hello ";**

**string = string + "World";**

**System.out.println(string); // Hello World**

We were saying that String in immutable in Java, once crated can not be modified. But as we can see above, we have modified the string which is giving us the concatenated string.  
  
Remember, What you are seeing above is actually not happening internally. We are adding another string to the existing string. So JVM will take both of the String and will create a new String. And the reference of the newly created String will be given to the existing String reference variable. So it will output the combined string and existing String literals will be removed from the memory.    
  
So “Hello ” will be removed from the memory and now String constant pool will contain only “Hello World”.

## ***Difference Between String , StringBuilder And StringBuffer***

## ***String***

**String** is **immutable** ( once created can not be changed )object . The object created as a String is stored in the **Constant String Pool**. Every immutable object in Java is **thread safe** ,that implies String is also thread safe . String can not be used by two threads simultaneously. String once assigned can not be changed.

## ***StringBuffer***

## 

**StringBuffer** is mutable means one can change the value of the object . The object created through StringBuffer is stored in the heap. StringBuffer has the same methods as the StringBuilder , but each method in StringBuffer is synchronized that is **StringBuffer is thread safe** .

Due to this it does not allow two threads to simultaneously access the same method . Each method can be accessed by one thread at a time . But being thread safe has disadvantages too as the performance of the StringBuffer hits due to thread safe property . Thus StringBuilder is faster than the StringBuffer when calling the same methods of each class.

String Buffer can be converted to the string by using toString() method. StringBuffer demo1 = new StringBuffer("Hello") ; // The above object stored in heap and its value can be changed .

demo1=new StringBuffer("Bye"); // Above statement is right as it modifies the value which is allowed in the StringBuffer

## ***StringBuilder***

**StringBuilder** is same as the **StringBuffer**, that is it stores the object in heap and it can also be modified . The main difference between the StringBuffer and StringBuilder is that **StringBuilder is also not thread safe**. StringBuilder is fast as it is not thread safe .

StringBuilder demo2= new StringBuilder("Hello"); // The above object too

is stored in the heap and its value can be modified

demo2=new StringBuilder("Bye");

// Above statement is right as it modifies the value which is allowed in the StringBuilder

## ***Check the performance***

public class CompareString

{

public static void main(String[] args)

{

long startTime = System.currentTimeMillis();

String ss = new String("Java");

for (int i = 0; i < 10000; i++) {

ss = ss + "Test";

}

System.out.println("Time taken by String : "

+ (System.currentTimeMillis() - startTime) + "ms");

startTime = System.currentTimeMillis();

StringBuffer sb = new StringBuffer("Java");

for (int i = 0; i < 10000; i++) {

sb.append("Test ");

}

System.out.println("Time taken by StringBuffer: "

+ (System.currentTimeMillis() - startTime) + "ms");

startTime = System.currentTimeMillis();

StringBuilder sb2 = new StringBuilder("Java");

for (int i = 0; i < 10000; i++) {

sb2.append("Test ");

}

System.out.println("Time taken by StringBuilder: "

+ (System.currentTimeMillis() - startTime) + "ms");

}

}

***Exception***



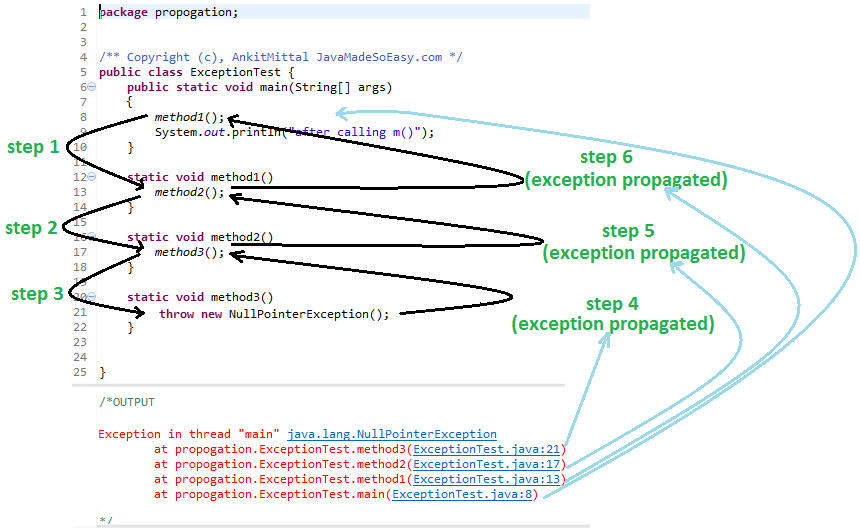
## ***Exception Propagation***

Whenever methods are called [stack](http://javamadesoeasy.com/2015/01/stacks.html) is formed and an exception is first thrown from the top of the stack and if it is not caught, it starts coming down the stack to previous methods until it is not caught.

If exception remains uncaught even after reaching bottom of the stack it is propagated to JVM and program is terminated.

## ***Propagating*** [***unchecked***](http://www.javamadesoeasy.com/2015/05/checked-compile-time-exceptions-and.html) ***exception (NullPointerException)***

unchecked exceptions are automatically propagated in java.

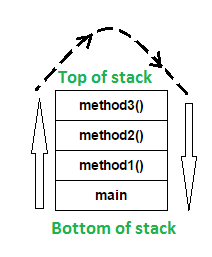


***Now, i’ll be explaining you how unchecked exception*** *was* ***propagated.***

***Let’s see step by step what happened in above program***

* ***JVM*** *called main method*
* ***step 1*** *- main called method1()*
* ***step 2*** *- method1 called method2()*
* ***step 3*** *- method2 called method3()*
* ***step 4*** *- method3* ***automatically propagated exception*** *to method2() [because, unchecked exceptions are propagated* ***automatically****]*
* ***step 5*** *- method2* ***automatically propagated exception*** *to method1() [because, unchecked exceptions are propagated* ***automatically****]*
* ***step 6*** *- method2* ***automatically propagated exception*** *to main() [because, unchecked exceptions are propagated* ***automatically****]*
* *main()* ***automatically propagated exception*** *to* ***JVM*** *[because, unchecked exceptions are propagated* ***automatically****]*

## ***Let's see how*** [***stack***](http://javamadesoeasy.com/2015/01/stacks.html) ***of methods is formed***

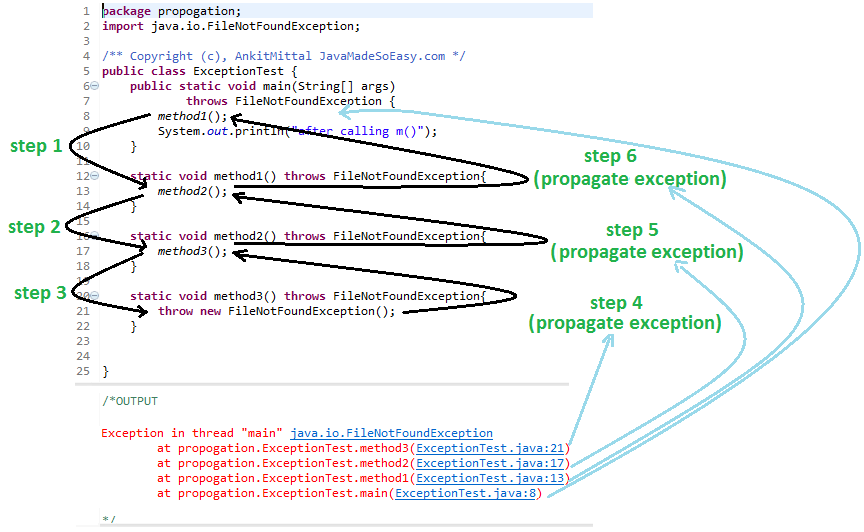


In the above program, stack is formed and an exception is first thrown from the top of the stack [ method3() ] and it remains uncaught there, and starts coming down the stack to previous methods to method2(), then to method1(), than to main() and it remains uncaught throughout.

exception remains uncaught even after reaching bottom of the stack [ main() ] so it is propagated to JVM and ultimately program is terminated by throwing exception [ as shown in output ].

## ***Propagating*** [***checked***](http://www.javamadesoeasy.com/2015/05/checked-compile-time-exceptions-and.html) ***exception (FileNotFoundException) using throws keyword***

For **propagating checked** exceptions method must throw exception by using [**throws**](http://www.javamadesoeasy.com/2015/05/throws-exception-in-java.html) keyword.



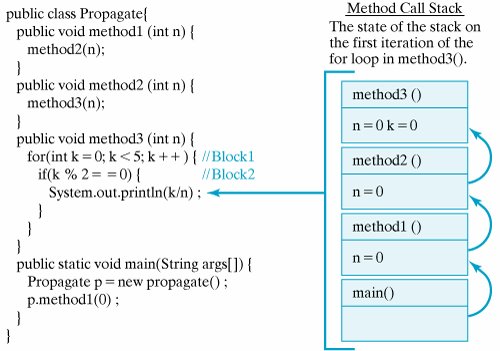
***Now, i’ll be explaining you how checked exception*** *was* ***propagated****.*

***Let’s see step by step what happened in above program***

* ***JVM*** *called main method*
* ***step 1*** *- main called method1()*
* ***step 2*** *- method1 called method2()*
* ***step 3*** *- method2 called method3()*
* ***step 4*** *- method3* ***propagated exception*** *to method2() using* ***throws*** *keyword****.****[because, checked exceptions are not propagated* ***automatically****]*
* ***step 5*** *- method2* ***propagated exception*** *to method1() using* ***throws*** *keyword****.****[because, checked exceptions are not propagated* ***automatically****]*
* ***step 6*** *- method2* ***propagated exception*** *to main() using* ***throws*** *keyword****.****[because, checked exceptions are not propagated* ***automatically****]*
* *main()* ***propagated exception*** *to* ***JVM*** *using* ***throws*** *keyword****.****[because, checked exceptions are not propagated* ***automatically****]*

## ***Exception Propagation: Searching for a Catch Block***

The method call stack for the Propagate program. The curved arrows give a trace of the method calls leading to the program's present state.



## ***Naming Conventions***

It's common to use a variable naming convention to distinguish between fields, arguments, and local variables

All the classes, interfaces, packages, methods and fields of java programming language are given according to java naming convention.

|  |  |
| --- | --- |
| **Name** | **Convention** |
| **class name** | should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc. |
| **interface name** | should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc. |
| **method name** | Methods in Java also follow the same lowerCamelCase convention like Objects and variables.should start with lowercase letter and be a verb e.g. actionPerformed(), main(), print(), println() etc. |
| **variable name** | should start with lowercase letter e.g. firstName, orderNumber etc. |
| **package name** | should be in lowercase letter e.g. java, lang, sql, util etc. |
| **constants name** | should be in uppercase letter. e.g. RED, YELLOW, MAX\_PRIORITY etc. |

| Java Bitwise and Bit Shift Operators | |
| --- | --- |
| Operator | Description |
| | | [Bitwise OR](https://www.programiz.com/java-programming/bitwise-operators#or) |
| & | [Bitwise AND](https://www.programiz.com/java-programming/bitwise-operators#and) |
| ~ | [Bitwise Complement](https://www.programiz.com/java-programming/bitwise-operators#complement) |
| ^ | [Bitwise XOR](https://www.programiz.com/java-programming/bitwise-operators#xor) |
| << | [Left Shift](https://www.programiz.com/java-programming/bitwise-operators#left-shift) |
| >> | [Right Shift](https://www.programiz.com/java-programming/bitwise-operators#right-shift) |
| >>> | [Unsigned Right Shift](https://www.programiz.com/java-programming/bitwise-operators#unsigned-right-shift) |

**Bitwise OR**

Bitwise OR is a binary operator (operates on two operands). It's denoted by |.

The | operator compares corresponding bits of two operands. If either of the bits is 1, it gives 1. If not, it gives 0. For example,

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bitwise OR Operation of 12 and 25

00001100

| 00011001

\_\_\_\_\_\_\_\_

00011101 = 29 (In decimal)

**Example 1: Bitwise OR**

class BitwiseOR {

public static void main(String[] args) {

int number1 = 12, number2 = 25, result;

result = number1 | number2;

System.out.println(result);

}

}

When you run the program, the output will be:

29

**Bitwise AND**

Bitwise AND is a binary operator (operates on two operands). It's denoted by &.

The & operator compares corresponding bits of two operands. If both bits are 1, it gives 1. If either of the bits is not 1, it gives 0. For example,

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bit Operation of 12 and 25

00001100

& 00011001

\_\_\_\_\_\_\_\_

00001000 = 8 (In decimal)

**Example 2: Bitwise AND**

class BitwiseAND {

public static void main(String[] args) {

int number1 = 12, number2 = 25, result;

result = number1 & number2;

System.out.println(result);

}

}

When you run the program, the output will be:

8

**Bitwise Complement**

Bitwise complement is an unary operator (works on only one operand). It is denoted by ~.

The ~ operator inverts the bit pattern. It makes every 0 to 1, and every 1 to 0.

35 = 00100011 (In Binary)

Bitwise complement Operation of 35

~ 00100011

\_\_\_\_\_\_\_\_

11011100 = 220 (In decimal)

**Example 3: Bitwise Complement**

class Complement {

public static void main(String[] args) {

int number = 35, result;

result = ~number;

System.out.println(result);

}

}

When you run the program, the output will be:

-36

**Bitwise XOR**

Bitwise XOR is a binary operator (operates on two operands). It's denoted by ^.

The ^ operator compares corresponding bits of two operands. If corresponding bits are different, it gives 1. If corresponding bits are same, it gives 0. For example,

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bitwise XOR Operation of 12 and 25

00001100

| 00011001

\_\_\_\_\_\_\_\_

00010101 = 21 (In decimal)

**Example 4: Bitwise XOR**

class Xor {

public static void main(String[] args) {

int number1 = 12, number2 = 25, result;

result = number1 ^ number2;

System.out.println(result);

}

}

When you run the program, the output will be:

21

**Signed Left Shift**

The left shift operator << shifts a bit pattern to the left by certain number of specified bits, and zero bits are shifted into the low-order positions.

212 (In binary: 11010100)

212 << 1 evaluates to 424 (In binary: 110101000)

212 << 0 evaluates to 212 (In binary: 11010100)

212 << 4 evaluates to 3392 (In binary: 110101000000)

**Example 5: Signed Left Shift**

class LeftShift {

public static void main(String[] args) {

int number = 212, result;

System.out.println(number << 1);

System.out.println(number << 0);

System.out.println(number << 4);

}

}

When you run the program, the output will be:

424

212

3392

**Signed Right Shift**

**The right shift operator >> shifts a bit pattern to the right by certain number of specified bits.**

212 (In binary: 11010100)

212 >> 1 evaluates to 106 (In binary: 01101010)

212 >> 0 evaluates to 212 (In binary: 11010100)

212 >> 8 evaluates to 0 (In binary: 00000000)

If the number is a 2's complement signed number, the sign bit is shifted into the high-order positions.

**Example 6: Signed Right Shift**

class RightShift {

public static void main(String[] args) {

int number = 212, result;

System.out.println(number >> 1);

System.out.println(number >> 0);

System.out.println(number >> 8);

}

}

When you run the program, the output will be:

106

212

0

**Unsigned Right Shift**

The unsigned right shift operator << shifts zero into the leftmost position.

**Example 7: Signed and UnSigned Right Shift**

class RightShift {

public static void main(String[] args) {

int number1 = 5, number2 = -5;

// Signed right shift

System.out.println(number1 >> 1);

// Unsigned right shift

System.out.println(number1 >>> 1);

// Signed right shift

System.out.println(number2 >> 1);

// Unsigned right shift

System.out.println(number2 >>> 1);

}

}

When you run the program, the output will be:

2

2

-3

2147483645

Notice, how signed and unsigned right shift works differently for 2's complement.

The 2's complement of 2147483645 is 3.

## ***Method overloading:***

Method overloading is the way of implementing static/compile time polymorphism in java. Method overloading means more than one methods in a class with same name but different parameters. Parameters can be differing in types, numbers or order. **Compiler resolve method call by matching method signature at compile time, that’s why it is known as static or compile time polymorphism**. It is also known as static binding.

## **Ways to implement method overloading in java:**

* Parameters differ in types.
* Parameters differ in number.
* Parameters differ in order.

## ***Aggregation:***

Aggregation is a type of HAS-A relationship. Aggregation represents a type of relationship between two objects in which one contain the other’s reference. Two objects can exist independently. If one is deleted other can still exist.

## ***Association:***

Association is a way of defining a relationship between classes of objects.

## ***Dynamic method dispatch:***

Dynamic method dispatch is a mechanism to resolve overridden method call at run time instead of compile time. It is based on the concept of up-casting (A super class reference variable can refer subclass object.).

**Example:**

class Student {

public void show(){

System.out.println("Student details.");

}

}

public class CollegeStudent extends Student {

public void show(){

System.out.println("College Student details.");

}

public static void main(String args[]){

//Super class can contain subclass object.

Student obj = new CollegeStudent();

//method call resolved at runtime

obj.show();

}

}

## **Marker/Tagging Interfaces:**

An interface with no methods is known as marker or tagged interface.

## ***Why marker interface used:***

It provides some useful information to JVM/compiler so that JVM/compiler performs some special operations on it. It is used for better readability of code.  Example: Serializable, Clonnable etc.

## **Map.Entry interface:**

**Map.Entry** interface provides the facility to work with a map entry. The **entrySet( )** method of the Map interface returns the set of map entries i.e. set of **Map.Entry** objects.

**Example:**

public class HashMapTest {

public static void main(String args[]){

//Create HashMap object.

Map hashMap = new HashMap();

//Add objects to the HashSet.

hashMap.put(4, "Roxy");

hashMap.put(2, "Sunil");

hashMap.put(5, "Sandy");

hashMap.put(1, "Munish");

hashMap.put(3, "Pardeep");

//Print the HashMap object.

System.out.println("HashMap elements:");

System.out.println(hashMap);

//Get iterator

Set set=hashMap.entrySet();

Iterator iterator=set.iterator();

//Print the HashMap elements using iterator.

System.out.println("HashMap elements using iterator:");

while(iterator.hasNext()){

Map.Entry mapEntry=(Map.Entry)iterator.next();

System.out.println("Key: " + mapEntry.getKey() + ", " +

"Value: " + mapEntry.getValue());

}

}

}

**JVM Architecture – Understanding JVM Internals**

Every Java developer knows that bytecode will be executed by **JRE** (Java Runtime Environment). But many doesn’t know the fact that **JRE** is the implementation of **Java Virtual Machine** (JVM), which analyzes the bytecode, interprets the code and executes it. It is very important as a developer we should know the Architecture of JVM, this enables us to write code more efficiently. In this JVM architecture in Java with diagram article, we will learn more deeply about **JVM architecture** in Java and **different components** of a JVM.

**What is a JVM in Java ?**

A **Virtual Machine** is a Software implementation of a Physical Machine, Java was developed with the concept of **WORA** **( *Write Once Run Anywhere*)**which runs on a **VM**. **The compiler** will be compiling the java file into a java **.class** file.  The **.class** file is input to JVM which Loads and executes the class file. Below goes the Architecture of JVM.

**JVM Architecture Diagram**

**[](https://javainterviewpoint-7ac9.kxcdn.com/wp-content/uploads/2016/01/JVM-Architecture.png)**

**How JVM works in Java ?**

As shown in the above architecture diagram JVM is divided into three main subsystems

1. **Class Loader Subsystem**
2. **Runtime Data Area**
3. **Execution Engine**

**1. Class Loader Subsystem**

Java’s dynamic class loading functionality is handled by the class loader subsystem. It loads, links and initializes the class when it refers to a class for the first time at **runtime**, not at **compile-time.**It performs three major functionality such as Loading, Linking, and Initialization.

**1.1 Loading**

Classes will be loaded by this component. BootStrap ClassLoader, Extension ClassLoader, Application ClassLoader are the three class loader which will help in achieving it.

1. **BootStrap ClassLoader** – Responsible for loading classes from the bootstrap classpath, nothing but **rt.jar.**Highest priority will be given to this loader.
2. **Extension ClassLoader** – Responsible for loading classes which are inside **ext** folder **(jre\lib)**
3. **Application ClassLoader** –Responsible for loading **Application Level Classpath**, path mentioned Environment Variable etc.

The above **Class Loaders** will follow **Delegation Hierarchy Algorithm**while loading the class files.

**1.2 Linking**

1. **Verify** – Bytecode verifier will verify whether the generated bytecode is proper or not if verification fails we will get **verification error**
2. **Prepare** – For all static variables memory will be allocated and assigned with **default values.**
3. **Resolve** – All **symbolic memory references** are replaced with the **original references** from **Method Area**.

**1.3 Initialization**

This is the final phase of Class Loading, here all [**static variable**](https://www.javainterviewpoint.com/use-of-static-keyword-in-java/)will be assigned with the original values and [**static block**](https://www.javainterviewpoint.com/java-static-import/) will be executed.

**2. Runtime Data Area**

Runtime Data Area is divided into 5 major components

1. **Method Area** – All the **Class level data** will be stored here including **static variables**. **Method Area** is **one per JVM** and it is a shared resource.
2. **Heap Area** – All the **Objects** and its corresponding**instance variables** and **arrays** will be stored here. **Heap Area** is also **one per JVM**since **Method area** and **Heap area** shares memory for multiple threads the data stored is **not thread safe.**
3. **Stack Area** – For every thread, a separate **runtime stack** will be created. For every **method call**, one entry will be made in the stack memory which is called as **Stack Frame**. All **local variables** will be created in the stack memory. Stack area is thread safe since it is not a shared resource. Stack Frame is divided into three sub-entities such as
   1. **Local Variable Array** – Related to the method how many **local variables** are involved and the corresponding values will be stored here.
   2. **Operand stack** – If any intermediate operation is required to perform, **operand stack** act as runtime workspace to perform the operation.
   3. **Frame data** – All symbols corresponding to the method is stored here. In the case of any **exception**, the catch block information will be maintained in the frame data.
4. **PC Registers** – Each thread will have separate**PC Registers,** to hold address of **current executing instruction** once the instruction is executed the PC register will be **updated** with the next instruction
5. **Native Method stacks** – Native Method Stack holds native method information. For every thread, separate native method stack will be created.

**3. Execution Engine**

The bytecode which is assigned to the **Runtime Data Area** will be executed by the Execution Engine. The Execution Engine reads the byte code and executes one by one.

1. **Interpreter** – Reads the bytecode, interprets it and executes it one by one. The interpreter interprets the bytecode faster but executes slowly. The disadvantage of the interpreter is that when one method called multiple times, every time interpretation is required.
2. **JIT Compiler** – JIT Compiler neutralizes the disadvantage of the Interpreter ( a single method called multiple times, each time interpretation is required ), The Execution Engine will be using the help of Interpreter in converting but when it found repeated code it uses JIT compiler which compiles the entire bytecode and changes it to native code.  This native code will be used directly for repeated method calls which improve the performance of the system.
   1. **Intermediate Code generator** – produces intermediate code
   2. **Code Optimizer** – Code Optimizer is responsible for optimizing the intermediate code generated above
   3. **Target Code Generator** – Target Code Generator is responsible for Generating Machine Code/ Native Code
   4. **Profiler** – **Profiler** is a special component, it is responsible for finding the hotspots (i.e) Used to identify whether the method is called multiple time or not.
3. **Garbage Collector** : Garbage Collector is a part of Execution Engine, it collects/removes the unreferenced objects. Garbage Collection can be triggered by calling ***“System.gc()”***, but the execution is not guaranteed. Garbage collector of JVM collects only those objects that are created by **new** keyword. So if you have created any object without **new**, you can use **finalize method** to perform cleanup.

**Java Native Interface (JNI)**:  **JNI** will be interacting with the **Native Method Libraries** and provides the Native Libraries required for the Execution Engine.

**Native Method Libraries :**It is a Collection of the Native Libraries which is required for the Execution Engine.

**Hashing:**

**Hashing** is the process of generating a key value (in this case, typically a 32 or 64 bit integer) from a piece of data.This hash value then becomes a basis for organizing and sorting the data. The hash value might be the first n bits of data, the last n bits of data, a modulus of the value, or in some cases, a more complicated function. Using the hash

value, different "hash buckets" can be set up to store data. If the hash values are distributed evenly (which is the case for an ideal hash algorithm), then the buckets will tend to fill up evenly, and in many cases, most buckets will have no

more than one or only a few objects in them. This makes the search even faster.

**Connection Pool:**

A **connection pool** operates by performing the work of creating connections ahead of time, In the case of a JDBC connection pool, a pool of Connection objects is created at the time the application server (or some other server) starts. These objects are then managed by a pool manager that disperses connections as they are requested by clients and returns them to the pool when it determines the client is finished with the Connection object. A great deal of housekeeping is involved in managing these connections.

When the connection pool server starts, it creates a predetermined number of Connection objects. A client application would then perform a JNDI lookup to retrieve a reference to a DataSource object that implements the ConnectionPoolDataSource interface. The client application would not need make any special provisions to use

the pooled data source;the code would be no different from code written for a nonpooled DataSource.

## ***Apache Camel:***

**Apache Camel** is a rule-based routing engine that provides a Java object-based implementation of the Enterprise Integration Patterns using

an API (or declarative Java Domain Specific Language) to configure from which source to accept message and determine how to process and

send those messages to the other destinations.

Apache Camel uses URIs to work directly with any kind of Transport or messaging model such as HTTP, ActiveMQ, JMS etc., but on the outset, the API remains same

regardless of the transport protocol the systems are using.

**Camel has two main ways of defining routing rules:**

* **the Java-based domain specific language (DSL)**
* **and the Spring XML configuration format.**

I will show you an example

for each case.

**CamelContext:**

CamelContext is at heart of all camel application and it represents Camel run time system.

**Create camelcontext**.

**Add endpoints or components**.

**Add Routes to connect the endpoints**.

**Invoke camelcontext.start() – This starts all the camel-internal threads which are responsible for receiving, sending and processing messages in the endpoints.**

**Lastly invoking camelcontext.stop() when all the messages are exchanged and processed. This will gracefully stop all the camel-internal threads and endpoints.**

**RouterBuilder** can be created by extending org.apache.camel.builder.**RouterBuilder** class and overriding configure() method.

## ***Agile:***

**With Agile development methodology –**

•In the Agile methodology, each project is broken up into several ‘Iterations’.

•All Iterations should be of the same time duration (between 2 to 8 weeks).

•At the end of each iteration, a working product should be delivered.

•In simple terms, in the Agile approach the project will be broken up into 10 releases (assuming each iteration is set to last 4 weeks).

•Rather than spending 1.5 months on requirements gathering, in Agile software development, the team will decide the basic core features that are required in the

product and decide which of these features can be developed in the first iteration.

•Any remaining features that cannot be delivered in the first iteration will be taken up in the next iteration or subsequent iterations, based on priority.

•At the end of the first iterations, the team will deliver a working software with the features that were finalized for that iteration.

•There will be 10 iterations and at the end of each iteration the customer is delivered a working software that is incrementally enhanced and updated with the features that were shortlisted for that iteration.

**Advantages of Agile Methodology:**

•In Agile methodology the delivery of software is unremitting.

•The customers are satisfied because after every Sprint working feature of the software is delivered to them.

•Customers can have a look of the working feature which fulfilled their expectations.

•If the customers has any feedback or any change in the feature then it can be accommodated in the current release of the product.

•In Agile methodology the daily interactions are required between the business people and the developers.

•In this methodology attention is paid to the good design of the product.

•Changes in the requirements are accepted even in the later stages of the development.

**Maven:**

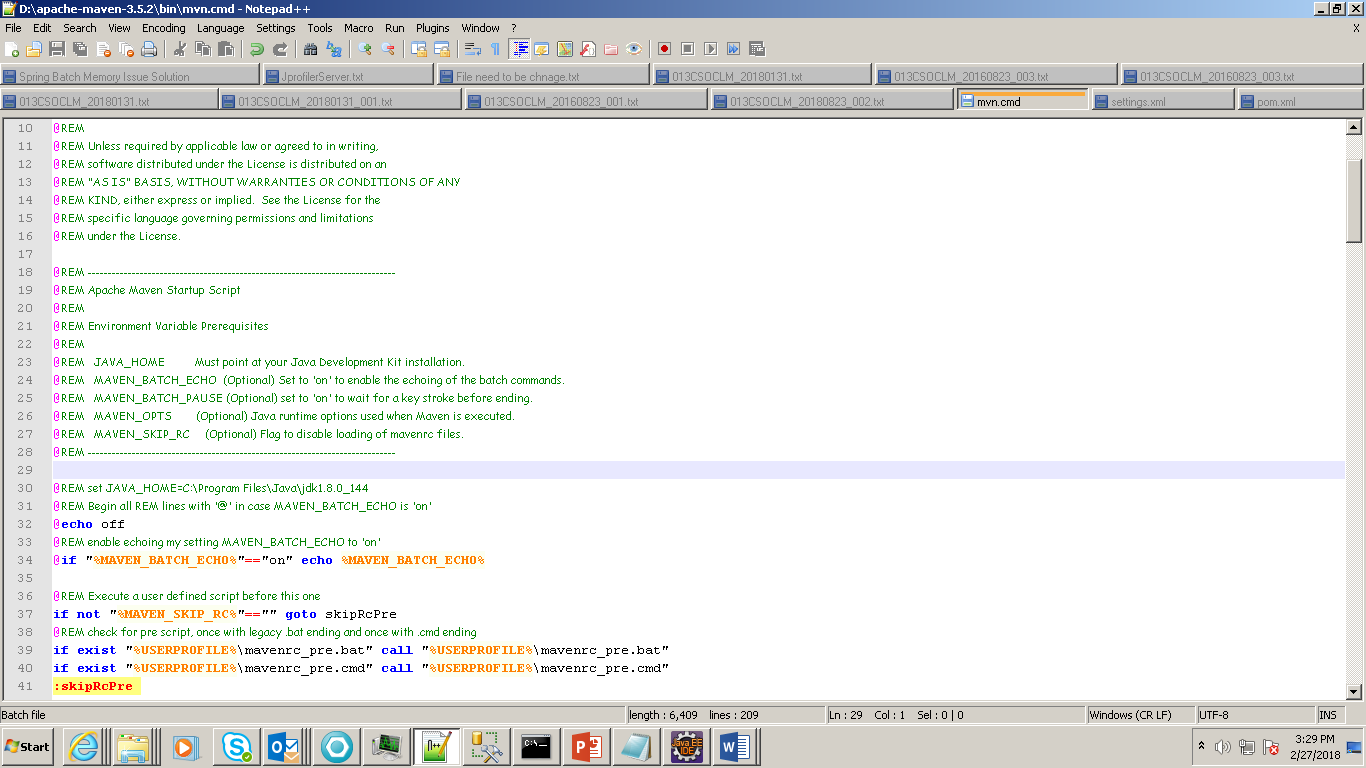
Set **JAVA\_HOME** and **M2\_HOME** and **MAVEN\_HOME**

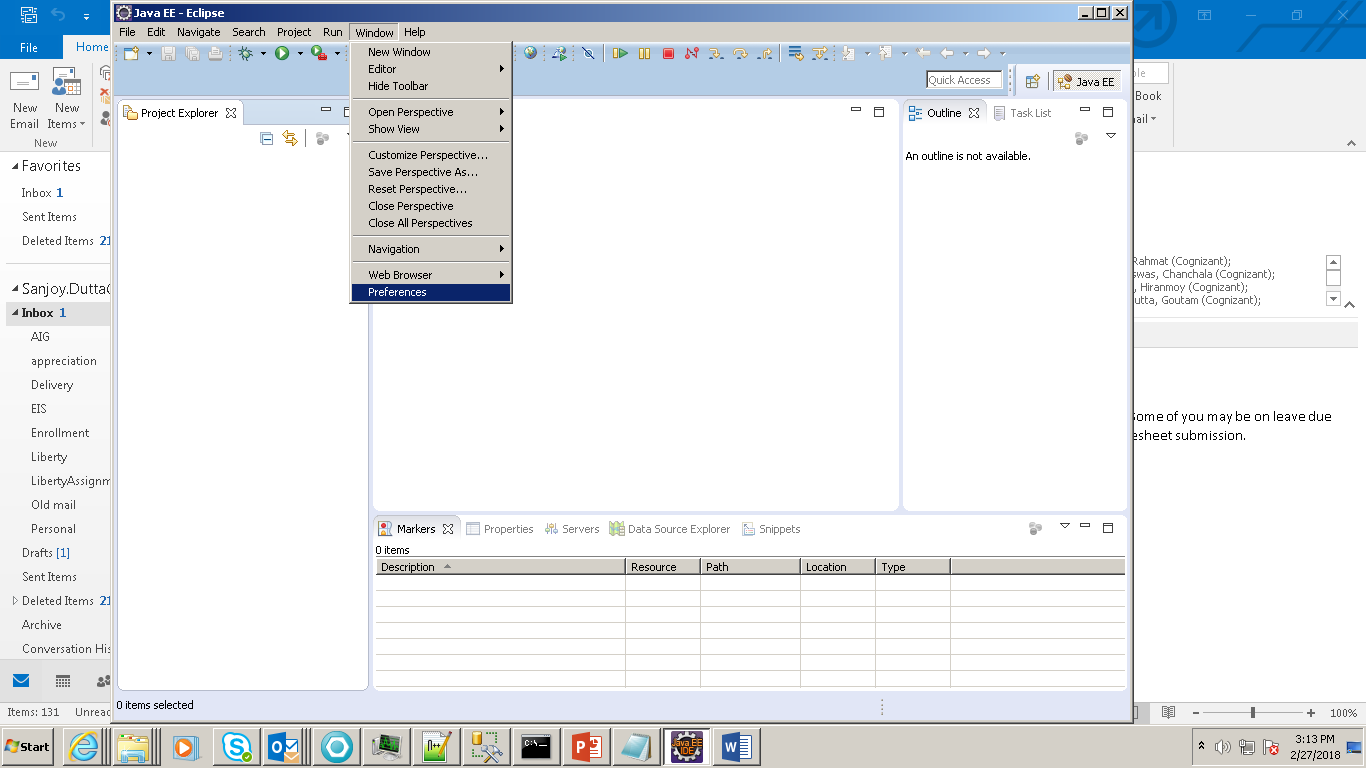
And run in command line to see the every thing works fine or not:

**Java –version**

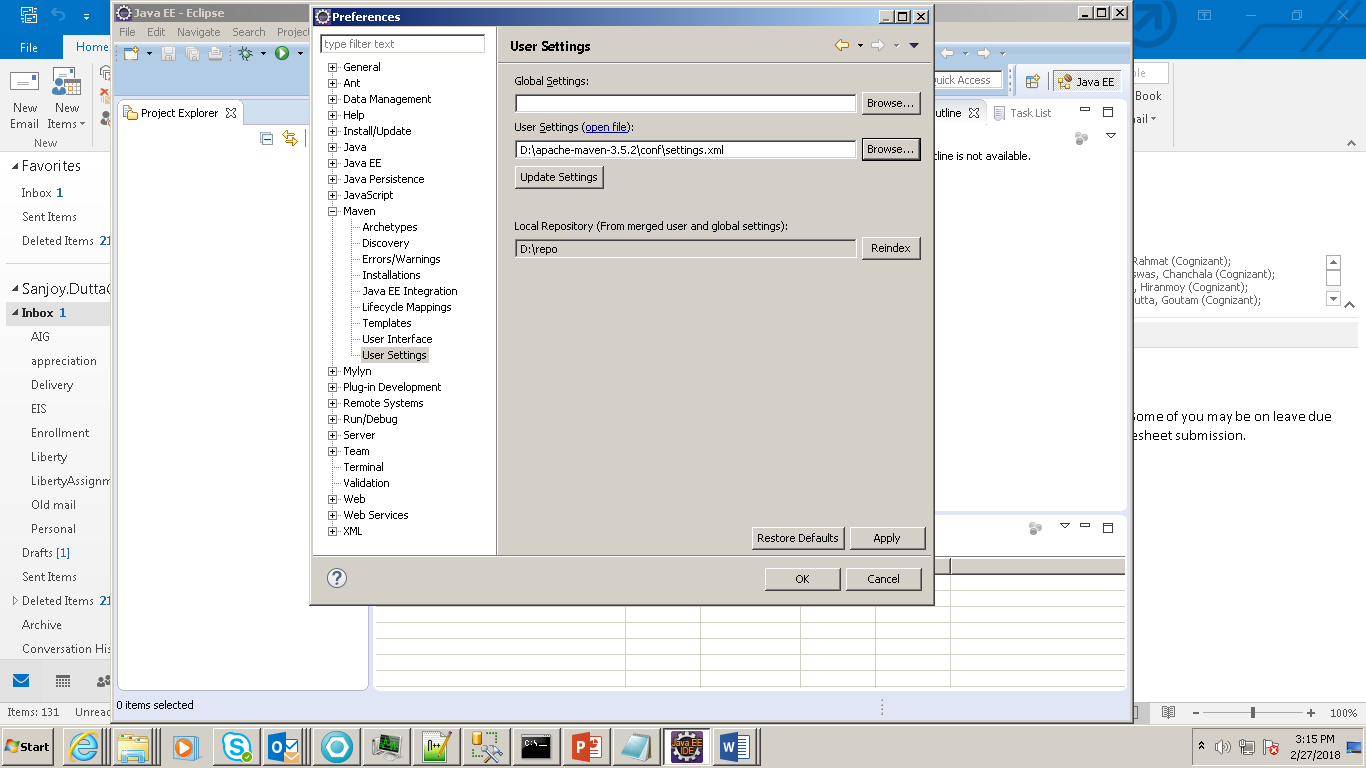
**Mvn –version**

And any problem check the jdk version in maven (**D:\apache-maven-3.5.2\bin\mvn.cmd**)

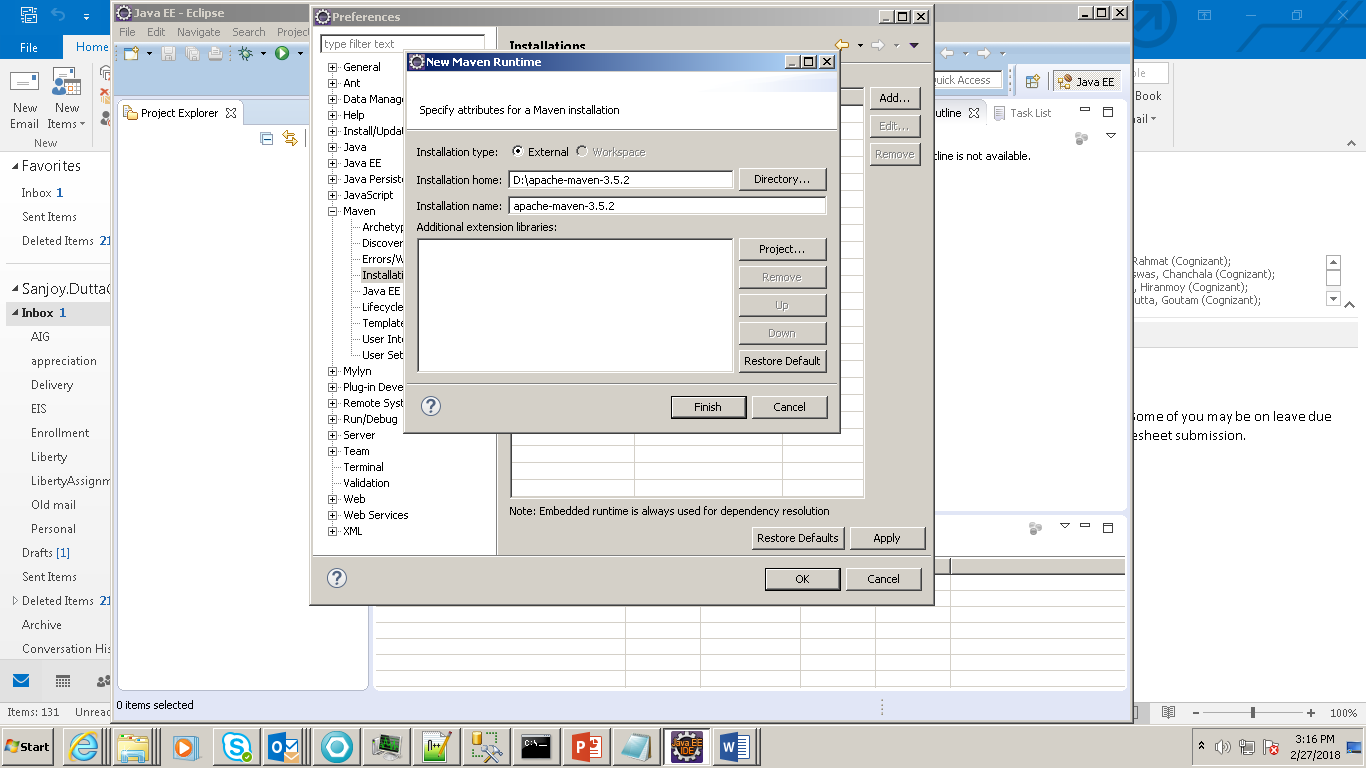




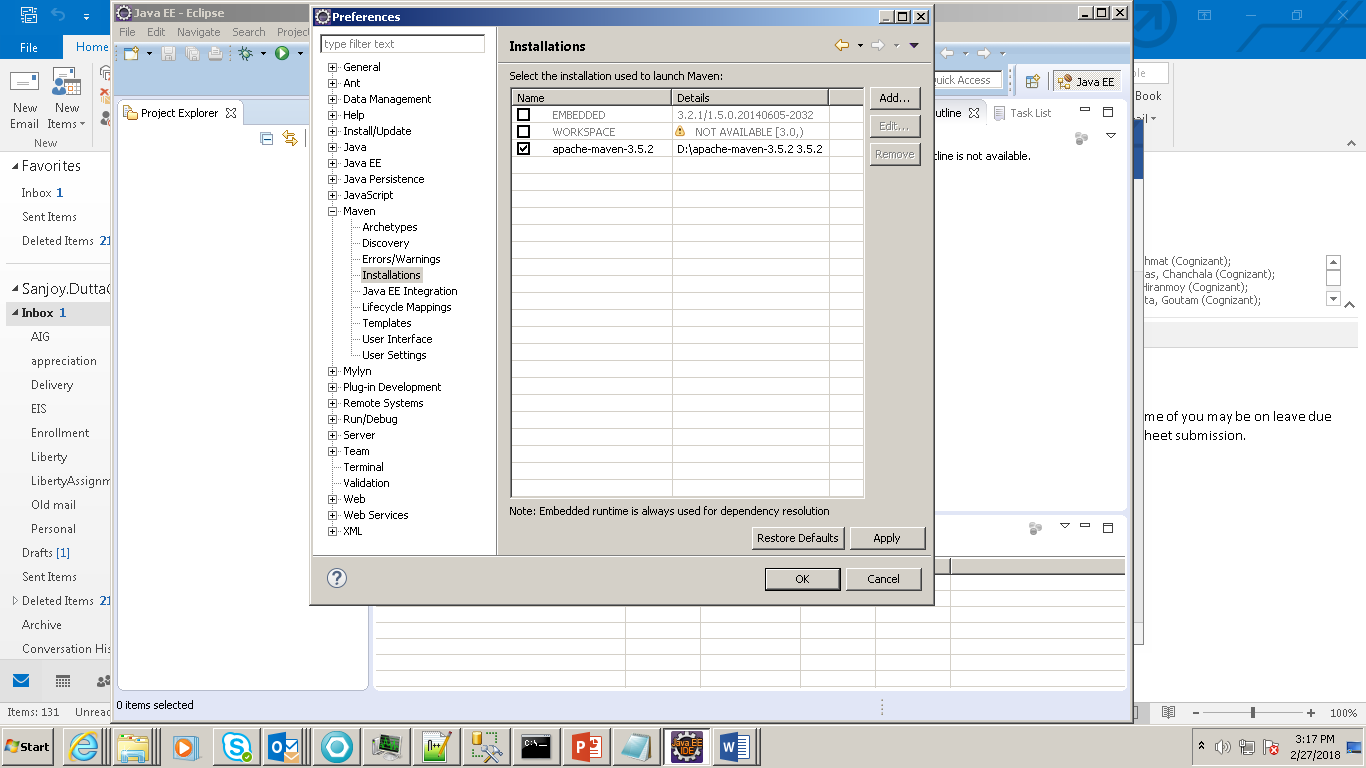
2.



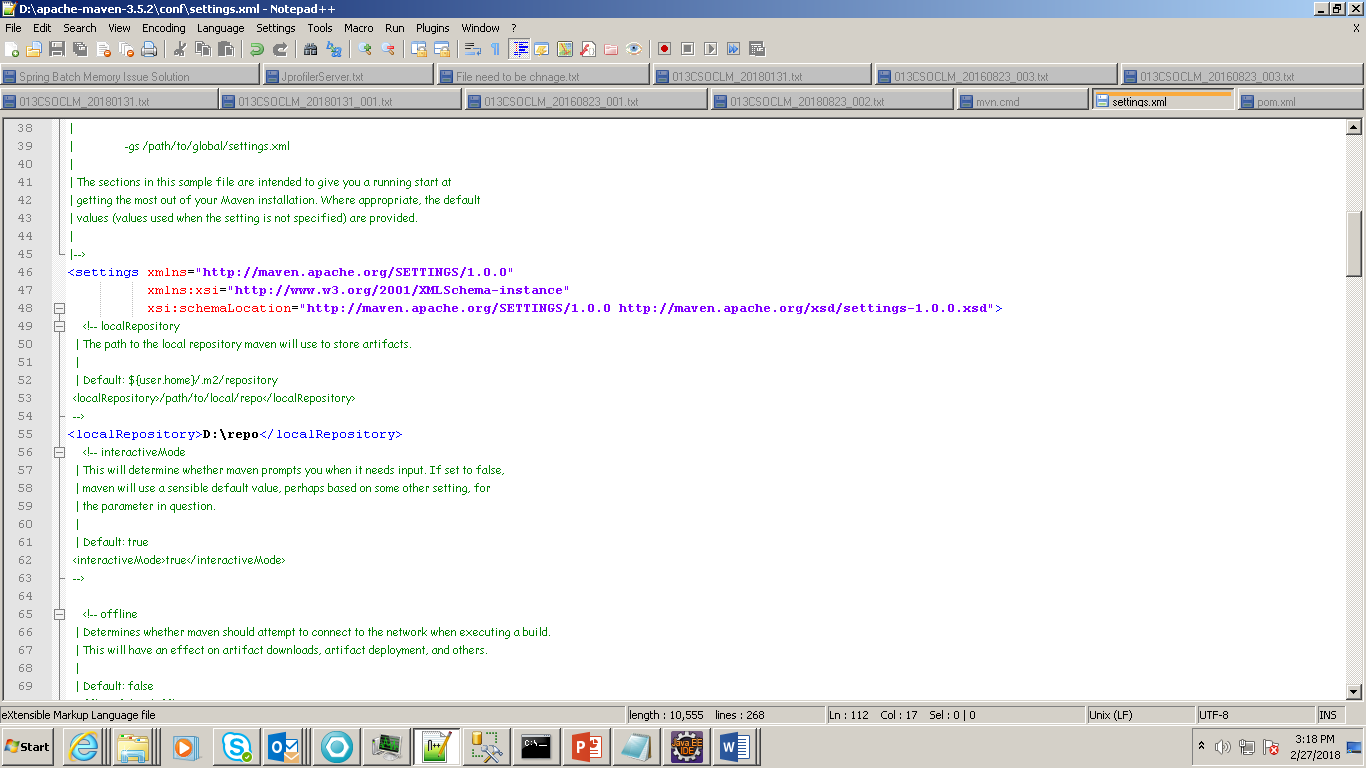
**Click on Installation and Add**

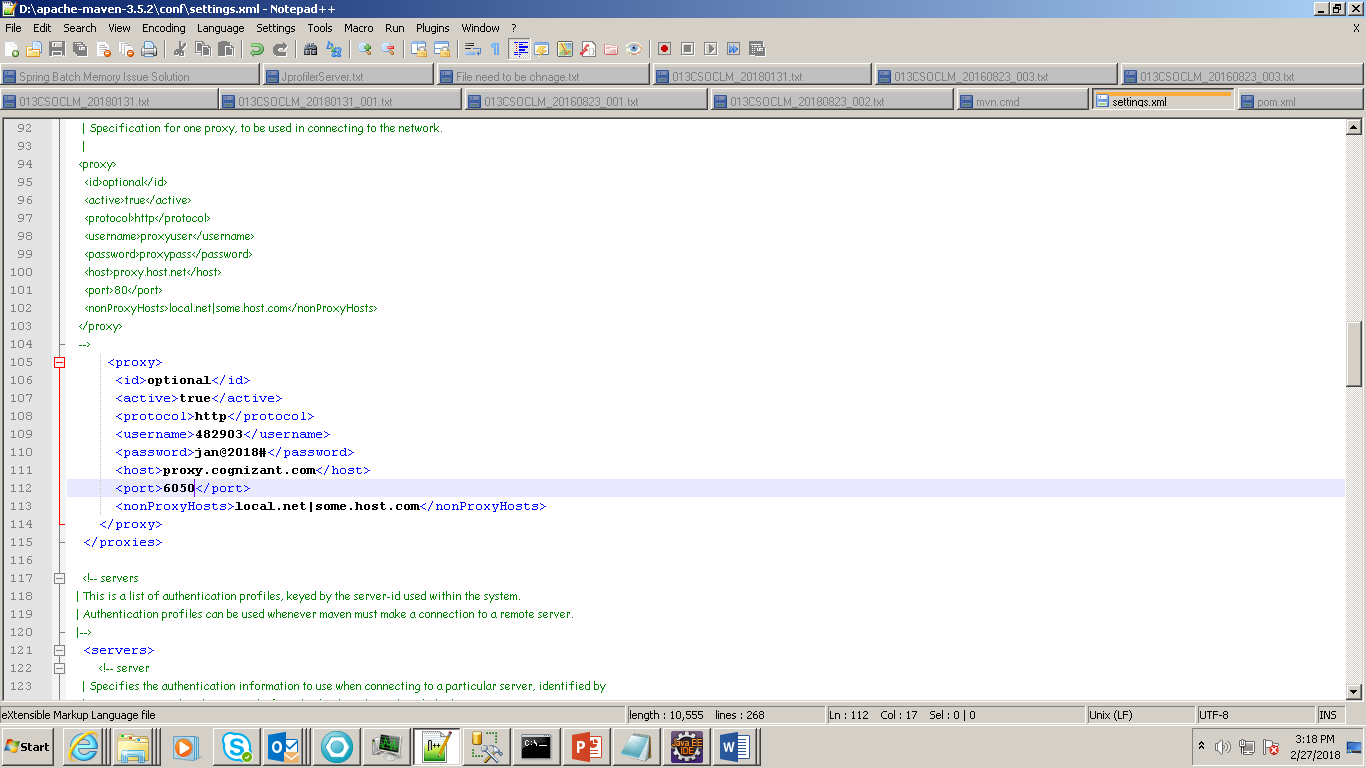


**Click on checkbox and apply**



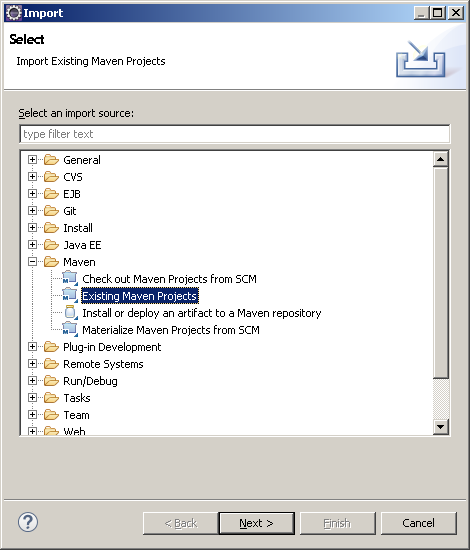
**Change in setting.xml of Maven:**





**For home there is no proxy**.

**Now Import the Maven project:**



**D:\spring-boot-web-jsp>mvn spring-boot:run**

**mvn clean install**

**Unix Shell Program:**

**Positional parameters**. A **positional parameter** is a **variable** within a shell program; its value is set from an argument specified on the command line that invokes the program. **Positional parameters** are numbered and are referred to with a preceding ``$'': $1, $2, $3, and so on.

[user3@localhost ~]$ **cd bca6**

[user3@localhost bca6]$ **ls -lrt**

total 32

-rw-rw-r--. 1 user3 user3 9 Mar 16 09:33 test2.txt

-rw-rw-r--. 1 user3 user3 47 Mar 16 09:37 test.txt

-rwxrw-r--. 1 user3 user3 114 Mar 16 10:06 fact.sh

-rwxrw-r--. 1 user3 user3 181 Mar 16 10:32 fibon.sh

-rwxrw-r--. 1 user3 user3 275 Mar 16 11:32 krishnamurti.sh

-rwxrw-r--. 1 user3 user3 304 Mar 16 11:47 palinnum.sh

-rwxrw-r--. 1 user3 user3 241 Mar 16 11:49 armstrong.sh

-rwxrw-r--. 1 user3 user3 251 Mar 16 12:32 arm.sh

[user3@localhost bca6]$ **cat fact.sh**

#!/bin/bash

fact=1

for ((c=1; c<=$1; c++))

do

fact=$(($fact \* $c))

done

echo "The Factorialof $1 is : $fact"

[user3@localhost bca6]$ cat fibon.sh

#!/bin/bash

t1=0

t2=1

next1=0

for ((c=0; c<=$1; c++))

do

if [ $c -eq 0 ]

then

echo "$c"

else

next1=$(( $t1 + $t2 ))

t1=${t2}

t2=${next1}

echo "$next1"

fi

done

[user3@localhost bca6]$

[user3@localhost bca6]$ cat krishnamurti.sh

#!/bin/bash

n=$1

a=0

r=0

c=0

s=0

a=$n

while [ $n -gt 0 ]

do

r=$(( $n % 10 ))

s=1

for(( i=$r; i>=1; i-- ))

do

s=$(( $s \* $i ))

done

c=$(( $c + $s ))

n=$(( n / 10 ))

done

if [ $c -eq $a ]

then

echo "num is krishnamurty"

else

echo "numis not krishnamurty"

fi

[user3@localhost bca6]$ ls -lrt

total 32

-rw-rw-r--. 1 user3 user3 9 Mar 16 09:33 test2.txt

-rw-rw-r--. 1 user3 user3 47 Mar 16 09:37 test.txt

-rwxrw-r--. 1 user3 user3 114 Mar 16 10:06 fact.sh

-rwxrw-r--. 1 user3 user3 181 Mar 16 10:32 fibon.sh

-rwxrw-r--. 1 user3 user3 275 Mar 16 11:32 krishnamurti.sh

-rwxrw-r--. 1 user3 user3 304 Mar 16 11:47 palinnum.sh

-rwxrw-r--. 1 user3 user3 241 Mar 16 11:49 armstrong.sh

-rwxrw-r--. 1 user3 user3 251 Mar 16 12:32 arm.sh

[user3@localhost bca6]$ cat palinnum.sh

#!/bin/bash

reversenum=0

remainder=0

n=$1

while [ $n -ne 0 ]

do

remainder=$(( $n % 10 ))

reversenum=$(( $reversenum \* 10 + $remainder));

n=$(( $n / 10 ))

done

if [ $1 -eq $reversenum ]

then

echo "the num is palindrome"

else

echo "the num is not palindrome"

fi

echo "reversed number $reversenum"

[user3@localhost bca6]$

if [ $1 -eq $reversenum ]

then

echo "the num is palindrome"

else

echo "the num is not palindrome"

fi

echo "reversed number $reversenum"

[user3@localhost bca6]$ **ls -lrt**

total 32

-rw-rw-r--. 1 user3 user3 9 Mar 16 09:33 test2.txt

-rw-rw-r--. 1 user3 user3 47 Mar 16 09:37 test.txt

-rwxrw-r--. 1 user3 user3 114 Mar 16 10:06 fact.sh

-rwxrw-r--. 1 user3 user3 181 Mar 16 10:32 fibon.sh

-rwxrw-r--. 1 user3 user3 275 Mar 16 11:32 krishnamurti.sh

-rwxrw-r--. 1 user3 user3 304 Mar 16 11:47 palinnum.sh

-rwxrw-r--. 1 user3 user3 241 Mar 16 11:49 armstrong.sh

-rwxrw-r--. 1 user3 user3 251 Mar 16 12:32 arm.sh

**Write a shell program to take a no from the user and also check whether it is Armstrong or not**

[user3@localhost bca6]$ **cat armstrong.sh**

#!/bin/bash

n=$1

a=$1

s=0

r=0

while [ $n -gt 0 ]

do

r=$(( $n % 10 ))

d=$(( $r \* $r \* $r ))

s=$(( $s + $d ))

n=$(( $n / 10 ))

done

if [ $s -eq $a ]

then

echo "the number $a is armstrong"

else

echo "the num $a is not armstrong"

fi

[user3@localhost bca6]$

**Write a shell program to print** Hello World **in the console**

[user3@localhost ~]$ cat program1.sh

clear

echo "Hello World"

[user3@localhost ~]$

**Write a shell program to take two no’s from the user and find the sum**

[user3@localhost ~]$ **cat program2.sh**

clear

echo "Enter First number: "

read num

echo "Enter second number"

read num1

sum=$((num+num1))

echo "the sum is" $sum

[user3@localhost ~]$

**Write a shell program to take a no from the user and find the reverse and also check whether it is Palindrome or not**

[user3@localhost ~]$ **cat rev.sh**

clear

echo "Enter Number:- "

read n

n1=$n

sum=0

while [ $n -gt 0 ]

do

x=$(expr $n % 10)

n=$(expr $n / 10)

sum=$(expr $sum \\* 10 + $x)

done

echo " Reverse the number of $n1 is $sum"

if [$n1-eq$sum]

then

echo "the no is palindrome"

else

echo "the no is not palindrome"

fi

echo "end"

[user3@localhost ~]$

**Write a shell program to take a no from the user and find the sum of digits of that no**

[user3@localhost ~]$ **cat sumofdigit.sh**

echo "enter a number:- "

read num

n=$num

sum=0

r=0

while [ $n -gt 0 ]

do

r=`expr $n % 10`

sum=`expr $sum + $r`

n=`expr $n / 10`

done

echo "the sum of digits of $num is : $sum"

[user3@localhost ~]$

**To run the shell program:**

**[user3@localhost ~]./ sumofdigit.sh**

**Permission issue of the file:**

**[user3@localhost ~]chmod a+x sumofdigit.sh**

1.Overview of Java And Creation of Java

Structured programming cannot manage complexity of complex programs. Obeject Oriented Programming makes it possible to organise large and complex programs using classes, objects, inheritance, polymorphism and encapsulation.C++ for years was most widely used OOP language. Due to advent of internet, Java became widely used and gained tremendous popularity.

James Ghosling,Naughton at Sun Microsystems in 1991.Initial name was Oak but it was renamed to Java in 1995

2.Features of Java:

1.Object Oriented

2.Platform Independent

3.Robust

4.Secured

5.Multi threading

6.Portable

3.Setting Classpath and Path for Java

Setting the path variable is like telling command prompt that where it should search for javac and java commands

ClassPath:

>javac -d (Specify the path where to save generated .class files) FileName.java

I:\Sources> javac -d I:\Classes ProgramOne.java

This will save generated .class file in I:\Classes.

If you run java command from this location, you will get an error saying could not find or load main class. Because .class file is saved in another folder. To run .class file, you need to go to that location, again using CD command. This will be the time consuming.

The easy and best way to run .class files saved in another folder is use -classpath option of java command. Syntax of java command

with -classpath option is,

>java -classpath (path of generated .class files) ClassName

In our example it looks like,

I:\Sources> java -classpath I:\Classes ProgramOne

You can also avoid use of -classpath option each time you run java command by setting the classpath variable by using set classpath command.

I:\Sources> set classpath=I:\Classes

Command line argument in Java:Command line argument is a parameter that is passed to the application at the time of execution.

4.JVM---interpretor for ByteCode

5.JDK,JRE,JVM

JRE==JVM+ other components to run applets and applications written in Java Language

JRE does not contain tools such as compilers or debuggers

JDK==JRE + tools such as compiler and debugger

6.Program Explanation:

Static is again a keyword used to make a function static.To execute a static function

you do not have to create an object of the class.The main() method here is called by JVM without creating any object for class."System" is a final class from java.lang package. "out" is static member of System class of type PrintStream. "println" is a method of PrintStream class.

You can explore the source code of both System class and String class. Go to JDK installation directory and extract the 'src' zip file.

Then go to src –> java –> lang. In lang folder you will find both System and String Java files.

7.Data types in Java

8.Variables In Java:

1.Instance variables:Instance variables are variables that are declare inside a class but outside any method,constructor or block.

2.Static variables:Static are class variables declared with static keyword

3.Local variables

9.Java Operators:

Bitwise operators --

& Bitwise AND

| Bitwise OR

^ Bitwise exclusive OR

<< left shift

>> right shift

a b a & b a | b

0 0 0 0

0 1 0 1

1 0 0 1

1 1 1 1

instanceOf operator:

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

The operator checks whether the object is of particular type at runtime

public class Test

{

public static void main(String[] args)

{

Test t= new Test();

System.out.println(t instanceof Test);

}

}

10.Object and Classes:

Class is a blue print or prototype that defines the data member and method common to all objects of a particular type.

Object is an instance of a class.

Features of OOPS

11.Constructors

First statement in a constructor must be either super() or this(). If you put any other statements you will get compile time error.If you don't

include these statements, by default compiler will keep super() calling statement.super() – It is a calling statement to default constructor of super class. this()- it is a calling statement to constructor of the same class.

There are three types of Constructor

•Default Constructor

•Parameterized constructor

•Copy constructor

Constructor Overloading -One class can have more than one constructors. It is called Constructor Overloading

Constructor chaining --Constructor chaining is a phenomena of calling one constructor from another constructor of same class

this keyword:this keyword is used to refer to current object.

12.Garbage Collection--Garbage Collection is a process that only frees the memory of Objects that are no more in use.

Can the Garbage Collection be forced explicitly ?

No, the Garbage Collection can not be forced explicitly. We may request JVM for garbage collection by calling System.gc() method. But This

does not guarantee that JVM will perform the garbage collection.

Advantages of Garbage Collection:

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

a.Programmer doesn't need to worry about dereferencing an object.

b.It is done automatically by JVM.

c.Increases memory efficiency and decreases the chances for memory leak.

finalize() method:

Sometime an object will need to perform some specific task before it is destroyed such as closing an open connection or releasing any resources held.

To handle such situation finalize() method is used

Memory Leak in Java:

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

A memory leak in Java is a situation where some objects are not used by the application any more, but GC fails to recognize them as unused.

As a result, these objects remain in memory indefinitely, reducing the amount of memory available to the application.

13.Modifiers in Java

1.Access control modifier:

Default : Default has scope only inside the same package

Public : Public scope is visible everywhere

Protected : Protected has scope within the package and all sub classes

Private : Private has scope only within the classes

2.Non Access Modifier:

a.Final

b.Static

c.Transient

d.Synchronized

e.Volatile

Transient:

If the variable is declared as transient, then it will not be persisted. That is the main purpose of the transient keyword

Volatile:

By making a variable volatile using the volatile keyword in Java, application programmer ensures that its value should always be read from main memory and thread should not use cached value of that variable from their own stack. volatile is used to indicate that a variable's value will be modified by different threads

14.Inheritance (IS-A)

super keyword--super keyword is used to refer to immediate parent class of a class

Q. Can you use both this() and super() in a Constructor?

NO, because both super() and this() must be first statement inside a constructor. Hence we cannot use them together.

15.Aggregation (HAS-A)

Q.Can we Override static method ? Explain with reasons ?

No, we cannot override static method. Because static method is bound to class whereas method overriding is associated with object i.e at runtime.

16.Polymorphism

17.Package:

a.Built-in Package:java.lang, java.util etc.

b.User Defined Package

Static import:

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

Static import is used to import static member of a class.

import static package.class-name.static-member-name;

Example using static import:

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

import static java.lang.System.\*;

public class Test

{

public static void main(String[] args)

{

out.println("Welcome");

}

}

18.Abstract class and Interface

Interface is a contract for what a class can do.

19.Nested Class:

Example:

class Outer

{

public void display()

{

Inner in=new Inner();

in.show();

}

class Inner

{

public void show()

{

System.out.println("Inside inner");

}

}

}

class Test

{

public static void main(String[] args)

{

Outer ot=new Outer();

ot.display();

}

}

Annonymous class:

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

A class without any name is called Annonymous class.

20.String Handling,StringBuffer and StringBuilder:

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

Each time you create a String literal, the JVM checks the string pool first. If the string literal already exists in the pool, a reference to the pool instance is returned. If string does not exist in the pool, a new string object is created, and is placed in the pool. String objects are stored in a special memory area known as string constant pool inside the heap memory.

Example showing difference between String and StringBuffer:

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

class Test {

public static void main(String args[])

{

String str = "Test";

str.concat("Java");

System.out.println(str); // Output: Test

StringBuffer strB = new StringBuffer("Test");

strB.append("Java");

System.out.println(strB); // Output: TestJava

}

}

Example of StringBuilder:

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

class Test {

public static void main(String args[])

{

StringBuilder str = new StringBuilder("Test");

str.append( "Java" );

System.out.println(str);

str.replace( 6, 13, "today");

System.out.println(str);

str.reverse();

System.out.println(str);

str.replace( 6, 13, "today");

}

}

21.Exception Handling:

Exception is an event that occurs during the execution of a program which interupts the flow.

Exception Handling Mechanism

In java, exception handling is done using five keywords,

1.try

2.catch

3.throw

4.throws

5.finally

Exception handling is done by transferring the execution of a program to an appropriate exception handler when exception occurs.

Example for Unreachable Catch block:

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

While using multiple catch statements, it is important to remember that exception sub classes inside catch must come before any of their super classes

otherwise it will lead to compile time error.

class Excep

{

public static void main(String[] args)

{

try

{

int arr[]={1,2};

arr[2]=3/0;

}

catch(Exception e) //This block handles all Exception

{

System.out.println("Generic exception");

}

Throw and Throws Keyword:

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

By using throw we can throw an exception explicitly.

By using throws we can list the type of exceptions that a method might throw.

More about Overriden Methods and Exceptions:

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

If Super class method throws an exception, then Subclass overriden method can throw the same exception or no exception, but must not throw parent

exception of the exception thrown by Super class method.

It means, if Super class method throws object of NullPointerException class, then Subclass method can either throw same exception, or can throw no exception, but it can never throw object of Exception class (parent of NullPointerException class).

There are few things to remember when overriding a method with exception handling. If super class method does not declare any exception, then sub

class overriden method cannot declare checked exception but it can declare unchecked exceptions.

Example of Subclass overriden Method declaring Checked Exception

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

import java.io.\*;

class Super

{

void show() { System.out.println("parent class"); }

}

public class Sub extends Super

{

void show() throws IOException //Compile time error

{ System.out.println("parent class"); }

public static void main( String[] args )

{

Super s=new Sub();

s.show();

}

}

As the method show() doesn't throws any exception while in Super class, hence its overriden version can also not throw any checked exception.

Example of Subclass overriden Method declaring Unchecked Exception:

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

import java.io.\*;

class Super

{

void show(){ System.out.println("parent class"); }

}

public class Sub extends Super

{

void show() throws ArrayIndexOutOfBoundsException //Correct

{ System.out.println("child class"); }

public static void main(String[] args)

{

Super s=new Sub();

s.show();

}

}

Output : child class

Because ArrayIndexOutOfBoundsException is an unchecked exception hence, overrided show() method can throw it.

Example of Subclass overriden method with same Exception:

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

import java.io.\*;

class Super

{

void show() throws Exception

{ System.out.println("parent class"); }

}

public class Sub extends Super {

void show() throws Exception //Correct

{ System.out.println("child class"); }

public static void main(String[] args)

{

try {

Super s=new Sub();

s.show();

}

catch(Exception e){}

}

}

Example of Subclass overriden method with no Exception:

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

import java.io.\*;

class Super

{

void show() throws Exception

{ System.out.println("parent class"); }

}

public class Sub extends Super {

void show() //Correct

{ System.out.println("child class"); }

public static void main(String[] args)

{

try {

Super s=new Sub();

s.show();

}

catch(Exception e){}

}

}

Example of Subclass overriden method with parent Exception:

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

import java.io.\*;

class Super

{

void show() throws ArithmeticException

{ System.out.println("parent class"); }

}

public class Sub extends Super {

void show() throws Exception //Cmpile time Error

{ System.out.println("child class"); }

public static void main(String[] args)

{

try {

Super s=new Sub();

s.show();

}

catch(Exception e){}

}

}catch(ArrayIndexOutOfBoundsException e) //This block is unreachable

{

System.out.println("array index out of bound exception");

}

}

}

22. Multithreading

Synchronization:

Synchronization is a process that ensures only one thread can access one resource at a time.

The synchronization keyword in java creates a block of code referred to as critical section.

Every Java object with a critical section of code gets a lock associated with the object. To enter critical section a thread need to obtain the corresponding object's lock.

General Syntax :

synchronized (object)

{

//statement to be synchronized

}

Why we use Syncronization ?

If we do not use syncronization, and let two or more threads access a shared resource at the same time, it will lead to distorted results.

Consider an example, Suppose we have two different threads T1 and T2, T1 starts execution and save certain values in a file temporary.txt which will be used to calculate some result when T1 returns. Meanwhile, T2 starts and before T1 returns, T2 change the values saved by T1 in the file temporary.txt (temporary.txt is the shared resource). Now obviously T1 will return wrong result.

To prevent such problems, synchronization was introduced. With synchronization in above case, once T1 starts using temporary.txt file, this file will be locked(LOCK mode), and no other thread will be able to access or modify it until T1 returns.

Using Synchronized Methods

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

Using Synchronized methods is a way to accomplish synchronization. But lets first see what happens when we do not use synchronization in our program.

Example with no Synchronization:

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

class First

{

public void display(String msg)

{

System.out.print ("["+msg);

try

{

Thread.sleep(1000);

}catch(InterruptedException e)

{

e.printStackTrace();

}

System.out.println ("]");

}

}

class Second extends Thread

{

String msg;

First fobj;

Second (First fp,String str)

{

fobj = fp;

msg = str;

start();

}

public void run()

{

fobj.display(msg);

}

}

public class Syncro

{

public static void main (String[] args)

{

First fnew = new First();

Second ss = new second(fnew, "welcome");

Second ss1= new second (fnew,"new");

Second ss2 = new second(fnew, "programmer");

}

}

Output :

[welcome [ new [ programmer]

]

]

In the above program, object fnew of class First is shared by all the three running threads(ss, ss1 and ss2) to call the shared method(void display).

Hence the result is unsynchronized and such situation is called Race condition.

Synchronized Keyword

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

To synchronize above program, we must serialize access to the shared display() method, making it available to only one thread at a time. This is done by

using keyword synchronized with display() method.

synchronized void display (String msg)

Using Synchronized block

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

If you have to synchronize access to object of a class that has no synchronized methods, and you cannot modify the code. You can use synchronized block to use it.

class First

{

public void display(String msg)

{

System.out.print ("["+msg);

try

{

Thread.sleep(1000);

}

catch(InterruptedException e)

{

e.printStackTrace();

}

System.out.println ("]");

}

}

class Second extends Thread

{

String msg;

First fobj;

Second (First fp,String str)

{

fobj = fp;

msg = str;

start();

}

public void run()

{

synchronized(fobj) //Synchronized block

{

fobj.display(msg);

}

}

}

public class Syncro

{

public static void main (String[] args)

{

First fnew = new First();

Second ss = new second(fnew, "welcome");

Second ss1= new second (fnew,"new");

Second ss2 = new second(fnew, "programmer");

}

}

Output :

[welcome]

[new]

[programmer]

Because of synchronized block this program gives the expected output.

23.Autoboxing and Unboxing:

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

•Autoboxing and Unboxing features was added in Java5.

•Autoboxing is a process by which primitive type is automatically encapsulated(boxed) into its equivalent type wrapper

•Auto-Unboxing is a process by which the value of object is automatically extracted from a type wrapper.

Example of Autoboxing and Unboxing:

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

class Test

{

public static void main(String[] args)

{

Integer iob = 100; //Autoboxing of int

int i = iob; //unboxing of Integer

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

Character cob = 'a'; /Autoboxing of char

char ch = cob; //Auto-unboxing of Character

System.out.println(cob+" "+ch);

}

}

24.Serialization and Deserialization in Java

Serialization is the process for writing the state of an object to a byte stream.

25.Marker interface:

Marker Interface is an interface which does not have any method. Marker interface is used to inform compiler that the class implementing it has some special behaviour or meaning. Some example of Marker interface are,

•java.io.Serializable

•java.lang.Cloneable

•java.rmi.Remote

•java.util.RandomAccess

26.Generics:

Generics was added in Java 5 to provide compile-time type checking and removing risk of ClassCastException that was common while working with

collection classes. The whole collection framework was re-written to use generics for type-safety. Let’s see how generics help us using collection

classes safely.

List list = new ArrayList();

list.add("abc");

list.add(new Integer(5)); //OK

for(Object obj : list){

//type casting leading to ClassCastException at runtime

String str=(String) obj;

}

Above code compiles fine but throws ClassCastException at runtime because we are trying to cast Object in the list to String whereas one of the element

is of type Integer. After Java 5, we use collection classes like below.

List<String> list1 = new ArrayList<String>(); // java 7 ? List<String> list1 = new ArrayList<>();

list1.add("abc");

//list1.add(new Integer(5)); //compiler error

for(String str : list1){

//no type casting needed, avoids ClassCastException

}

27.Collection:

Collection

a.Set

b.List

c.Deque

Accessing a Collection:

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

To access, modify or remove any element from any collection we need to first find the element, for which we have to cycle throught the elements of the collection. There are three possible ways to cycle through the elements of any collection.

1.Using Iterator interface

2.Using ListIterator interface

3.Using for-each loop

Accessing elements using Iterator:

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

Iterator Interface is used to traverse a list in forward direction, enabling you to remove or modify the elements of the collection. Each collection

classes provide iterator() method to return an iterator.

import java.util.\*;

class Test\_Iterator

{

public static void main(String[] args)

{

ArrayList< String> ar = new ArrayList< String>();

ar.add("ab");

ar.add("bc");

ar.add("cd");

ar.add("de");

Iterator it = ar.iterator(); //Declaring Iterator

while(it.hasNext())

{

System.out.print(it.next()+" ");

}

}

}

Output :

ab bc cd de

Accessing element using ListIterator:

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

ListIterator Interface is used to traverse a list in both forward and backward direction. It is available to only those collections that implement the List Interface.

import java.util.\*;

class Test\_Iterator

{

public static void main(String[] args)

{

ArrayList< String> ar = new ArrayList< String>();

ar.add("ab");

ar.add("bc");

ar.add("cd");

ar.add("de");

ListIterator litr = ar.listIterator();

while(litr.hasNext()) //In forward direction

{

System.out.print(litr.next()+" ");

}

while(litr.hasPrevious()) //In backward direction

{

System.out.print(litr.previous()+" ");

}

}

}

Output :

ab bc cd de

de cd bc ab

Using for-each loop:

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

for-each version of for loop can also be used for traversing each element of a collection. But this can only be used if we don't want to modify the contents of a collection and we don't want any reverse access. for-each loop can cycle through any collection of object that implements Iterable interface.

import java.util.\*;

class ForEachDemo

{

public static void main(String[] args)

{

LinkedList< String> ls = new LinkedList< String>();

ls.add("a");

ls.add("b");

ls.add("c");

ls.add("d");

for(String str : ls)

{

System.out.print(str+" ");

}

}

}

Output :

a b c d

Why a non-static variable cannot be referenced from a static context ?

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

When you try to access a non-static variable from a static context like main method, java compiler throws a message like "a non-static variable cannot be referenced from a static context". This is because non-static variables are related with instance of class(object) and they get created when instance of a class is created by using new operator. So if you try to access a non-static variable

without any instance compiler will complain because those variables are not yet created and they don't have any existence until an instance is created and associated with it.

Example of accessing non-static variable from a static context

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

class Test

{

int x;

public static void main(String[] args)

{

x=10;

}

}

ClassLoader:

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

Java ClassLoader loads a java class file into java virtual machine.

1.Bootstrap Class Loader

2.Extensions Class Loader

3.System Class Loader

Polymorphism:

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

Polymorphism means one name multiple forms.It is a concept by which we can perform same action in different ways.

Method Overloading:

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

Method overloading is a programming concept when programmer declares two methods of the same name but with different method signature,

e.g. change in the argument list or change in the type of argument

One of the most popular examples of method overloading is System.out.println() method which is overloaded to accept all kinds of data types in Java.You have println() method which takes String, int, float,double or even char in output.All of those methods are collectively referred as an overloaded method in Java

Can we declare a class as Abstract without having any abstract method?

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

Ans: Yes we can create an abstract class by using abstract keyword before class name even if it doesn’t have any abstract method

abstract class Test

{

void m1()

{

System.out.println("ssss");

}

}

class Test2 extends Test

{

}

class A3

{

public static void main(String dd[])

{

Test2 ttt=new Test2();

ttt.m1();

}

}

Does Importing a package imports its sub-packages as well in Java?

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

Ans: In java, when a package is imported, its sub-packages aren’t imported and developer needs to import them separately if required.

When a lot of changes are required in data, which one should be a preference to be used? String or StringBuffer?

-------------------------------------------------------------------------Ans: Since StringBuffers are dynamic in nature and we can change the values of StringBuffer objects unlike String which is immutable,it’s always a good choice to use StringBuffer when data is being changed too much. If we use String in such a case, for every data change a new String object will be created which will be an extra overhead.

What’s meant by anonymous class?

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

Ans: An anonymous class is a class defined without any name in a single line of code using new keyword

Does java support multiple interitance? Why?

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

Ans) Java doesnt support multiple inheritance but it provide a way through which it can enact it. Consider the scenario is C++

Class A {

public void add() {

// some text

}

}

Class B {

public void add() {

// some text

}

}

Class C extends A,B {

public static void main(String arg[]){

C objC = new C();

objC.add(); // problem, compiler gets confused and cant

decide to call Class A or B method.

}

This problem is called Diamond problem.

This problem in java is taken care with the use of interfaces

In Java similar problem would look like:

interface A {

void add();

}

interface B {

void add();

}

class C implements A,B {

void add() {

// doesnt matter which interface it belong to

}

}

Java supports pass by value or pass by reference?

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

Ans) Java supports only pass by value. The arguments passed as a parameter to a method is mainly primitive data types or objects.

For the data type the actual value is passed.Java passes the references by value just like any other parameter. The pointer to the

object is passed as value.

Why is it preferred to declare: List<String> list = new ArrayList<String>(); instead of ArrayList<String> = new ArrayList<String>();

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

Ans) It is preferred because:

1.If later on code needs to be changed from ArrayList to Vector then only at the declaration place we can do that.

2.The most important one – If a function is declared such that it takes list. E.g void showDetails(List list);

When the parameter is declared as List to the function it can be called by passing any subclass of List like ArrayList,

Vector, LinkedList making the function more flexible.

What is difference between Enumeration and Iterator in Java?

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

Though both Iterator and Enumeration allows you to traverse over elements of Collections in Java,Iterator also allows you to remove elements from

collection during traversal but Enumeration doesn't allow that, it doesn't got the remove() method.

Iterator is more secure and safe as compared to Enumeration because it does not allow other thread to modify the collection object while some thread is iterating over it and throws ConcurrentModificationException.

SIB – Static Initialization Block, Static Variables And Static Methods

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

Static variables, Static Initialization Block and Static Methods – these all are static components or static members of a class.

These static members are stored inside the Class Memory. To access static members, you need not to create objects. Directly you can access

them with class name.

Static Initialization Block is used to initialize only static variables. It is a block without a name. It contains set of statements enclosed within { }.

The syntax of SIB looks like this,

static

{

//Set Of Statements

}

Consider the following program.

class StaticComponents

{

static int staticVariable;

static

{

System.out.println("StaticComponents SIB");

staticVariable = 10;

}

static void staticMethod()

{

System.out.println("From StaticMethod");

System.out.println(staticVariable);

}

}

public class MainClass

{

static

{

System.out.println("MainClass SIB");

}

public static void main(String[] args)

{

//Static Members directly accessed with Class Name

StaticComponents.staticVariable = 20;

StaticComponents.staticMethod();

}

}

Let us discuss execution of above program step by step.

Step 1:

When you trigger >java MainClass, java command divides allocated memory into two parts – Stack and Heap. First, java command enters stack memory for execution. First, it checks whether

MainClass is loaded into heap memory or not. If it is not loaded, loading operation of MainClass starts. Randomly some memory space is allocated to MainClass. It is called Class memory. All static members are loaded into this class memory. There is only one satic member in MainClass – main() method. It is loaded into class memory of MainClass.

Step 2:

After loading all static members, SIB – Static initialization Blocks are executed. Remember, SIBs are not stored in the heap memory. They just come to stack, execute their tasks and leaves the memory. So, after loading main() method, SIB of MainClass enters stack for execution. There is only one statement (Line 22) in SIB. it is executed. It prints "MainClass SIB" on console. After executing this statement, SIB leaves the stack memory.

Step 3:

Now, java command calls main() method for execution. main() method enters the stack. First statement (Line 28) is executed first. First, It checks whether class StaticComponents is loaded into memory. If it is not loaded, loading operation of StaticComponents takes place. Randomly, some memory is allocated to Class StaticComponents, then all static members of StaticComponents – 'staticVariable' and 'staticMethod()' are loaded into that class memory.'staticVariable' is a global variable. So, first it is initialized with default value i.e 0.

Step 4 :

After loading all static members of StaticComponents, SIB blocks are executed. So, SIB of class StaticComponents enters the stack for execution.

First Statement (Line 7) is executed. It prints "StaticComponents SIB" on the console. In the second statement, value 10 is assigned to'staticVariable'. There are no other statements left for execution, so it leaves stack memory.

Step 5 :

Now control comes back to main() method. The remaining part of first statement i.e value 20 is assigned to 'staticVariable' of class

StaticComponents,is executed. In the second statement (Line 29), it calls staticMethod() of class StaticComponents for execution.

Step 6:

staticMethod() of StaticComponents enters stack for execution. First statement (Line 13) is executed first. It prints "From staticMethod" on the console.In the second statement (Line 14), it prints the value of staticVariable i.e 20 on the console. There are no statements left. so, it leaves the stack.

Step 7:

Again, control comes back to main() method. There are no other statements left in main() method. so, it also leaves stack. java command also leaves

the stack.

Can We Overload main() method?

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

Yes, We can overload main() method. A Java class can have any number of main() methods. But to run the java class, class should have main()

method with signature as "public static void main(String[] args)". If you do any modification to this signature, compilation will be successful.

But, you can't run the java program. You will get run time error as main method not found.

public class MainMethod

{

public static void main(String[] args)

{

System.out.println("Execution starts from this method");

}

void main(int args)

{

System.out.println("Another main method");

}

double main(int i, double d)

{

System.out.println("Another main method");

return d;

}

}

Can we declare main() method as private or protected or with no access modifier?

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

No, main() method must be public. You can't define main() method as private or protected or with no access modifier.

This is because to make the main() method accessible to JVM. If you define main() method other than public, compilation will be successful but

you will get run time error as no main method found.

public class MainMethod

{

private static void main(String[] args)

{

//Run time error

}

}

Can We Declare main() Method As Non-Static?

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

No, main() method must be declared as static so that JVM can call main() method without instantiating it's class.

If you remove 'static' from main() method signature, compilation will be successful but program fails at run time.

public class MainMethod

{

public void main(String[] args)

{

System.out.println(1); //Run time error

}

}

Why main() method must be static?

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

Suppose, If main() is allowed to be non-static, then while calling the main method JVM has to instantiate it's class.

While instantiating it has to call constructor of that class. There will be an ambiguity if constructor of that class takes an argument.

For example, In the below program what argument JVM has to pass while instantiating class "MainMethod"

public class MainMethod

{

public MainMethod(int i)

{

//Constructor taking one argument

}

public void main(String[] args)

{

//main method as non-static

}

}

That’s why main() method must be static.

Can we change return type of main() method?

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

No, the return type of main() method must be void only. Any other type is not acceptable.

public class MainMethod

{

public static int main(String[] args)

{

return 1; //run time error : No main method found

}

}

Can main() method take an argument other than string array?

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

No, argument of main() method must be string array. But, from the introduction of var args you can pass var args of string type as an argument to main() method. Again, var args are nothing but the arrays.

public class MainMethod

{

public static void main(String... args)

{

//Var args as an argument

}

}

Can we run java class without main() method?

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

No, you can’t run java class without main method. But, there are some scenarios like if super class has main() method, then sub class can be run without defining main() method in it. For example, if you run class B in the below program, you will get 1 as output.

class A

{

public static void main(String[] args)

{

System.out.println(1);

}

}

public class B extends A

{

}

Note : Before Java 7, you can run java class by using static initializers. But, from Java 7 it is not possible.

Can you create an object without using new operator in Java?

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

Yes, We can create an object without using new operator. There are some other ways to create objects other than using new operator. But, 95% of object creation in java is done through new operator only.

a) Using newInstance() Method

Class c = Class.forName("packageName.MyClass");

MyClass object = (MyClass) c.newInstance();

b) Using clone() method.

MyClass object1 = new MyClass();

MyClass object2 = object1.clone();

c) Using object deserialization

ObjectInputStream inStream = new ObjectInputStream(anInputStream );

MyClass object = (MyClass) inStream.readObject();

d) Creating string and array objects :

String s = "string object";

int[] a = {1, 2, 3, 4};

What is constructor chaining?

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

Constructor Chaining is a technique of calling another constructor from one constructor. this() is used to call same class constructor where as super() is used to call super class constructor.

class SuperClass

{

public SuperClass(int i)

{

System.out.println("Super Class Constructor");

}

}

class SubClass extends SuperClass

{

public SubClass()

{

this(10); //Calling same class constructor

}

public SubClass(int i)

{

super(i); //Calling super class constructor

}

}

Can we call sub class constructor from super class constructor?

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

No. There is no way in java to call sub class constructor from a super class constructor.

Java Notes

Written By Sanjoy Dutta