## **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

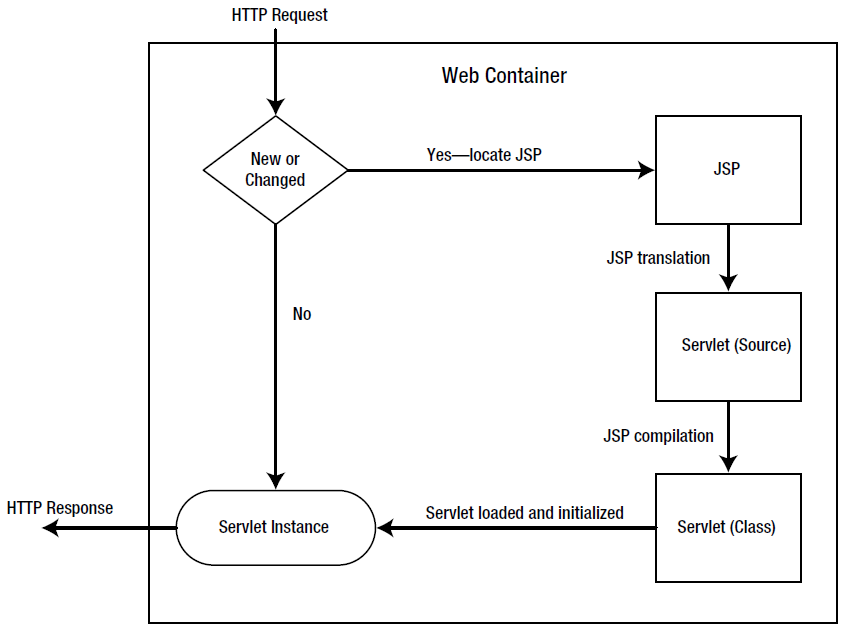
OPEN c\_dbuser FOR

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:***

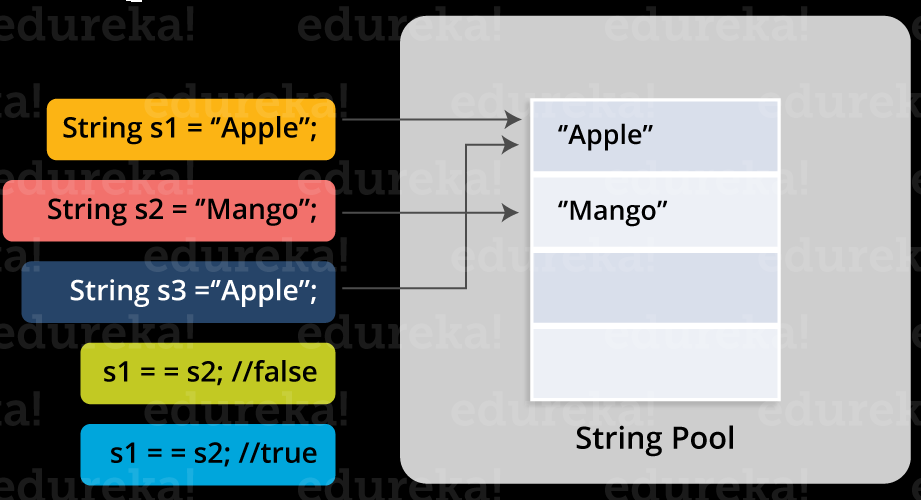
There are two ways to create a String object:

* **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:



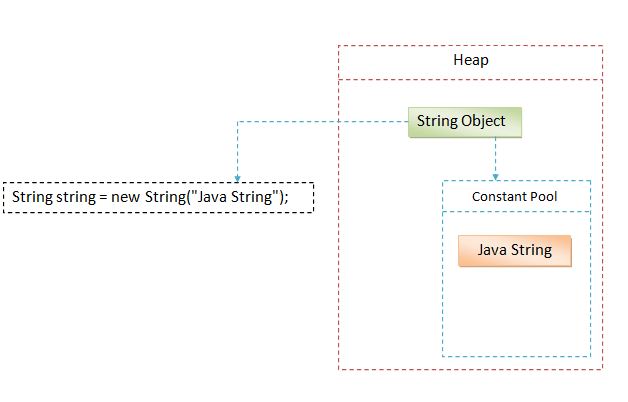
In the above image, two Strings are created using literal i.e “Apple” and “Mango”. Now, when third String is created with the value “Apple”, instead of creating a new object, the already present object reference is returned. That’s the reason Java String pool came into the picture.

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)

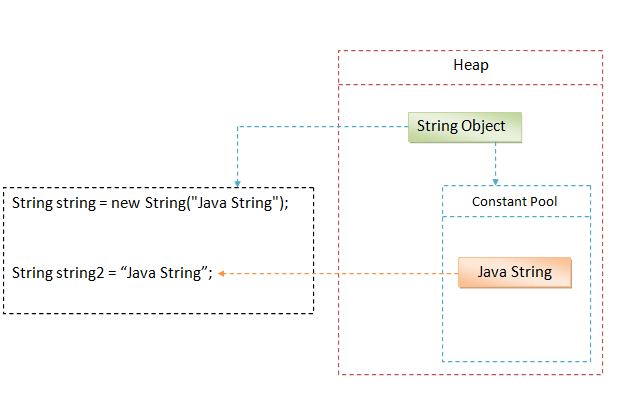
Look at the above diagram how String object is being created in Heap area.  
  
Generally we create lot of string objects in our project or program during run time. So to make this process faster, Java though let’s create a separate area where we can create String objects faster to make our program response 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***



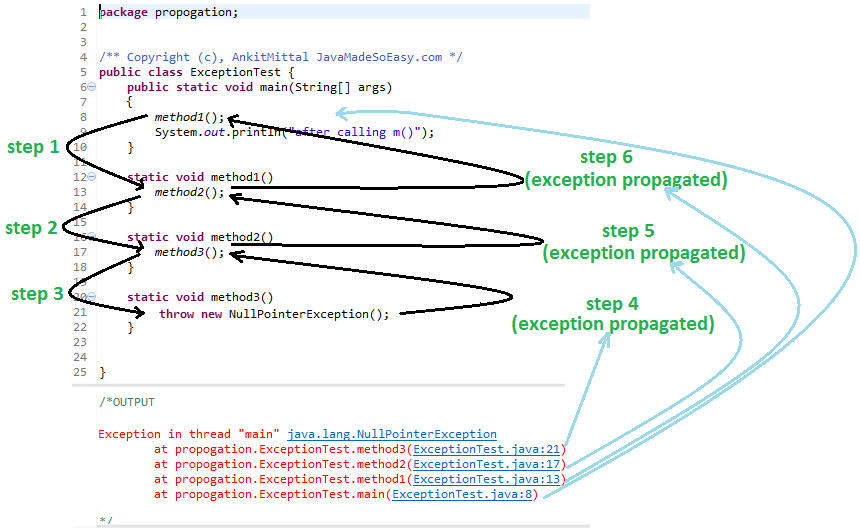
## ***Exception Propagation***

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.

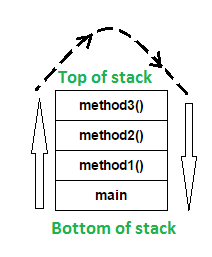


***Now, i’ll be explaining you how unchecked exception*** *was* ***propagated.***

***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***

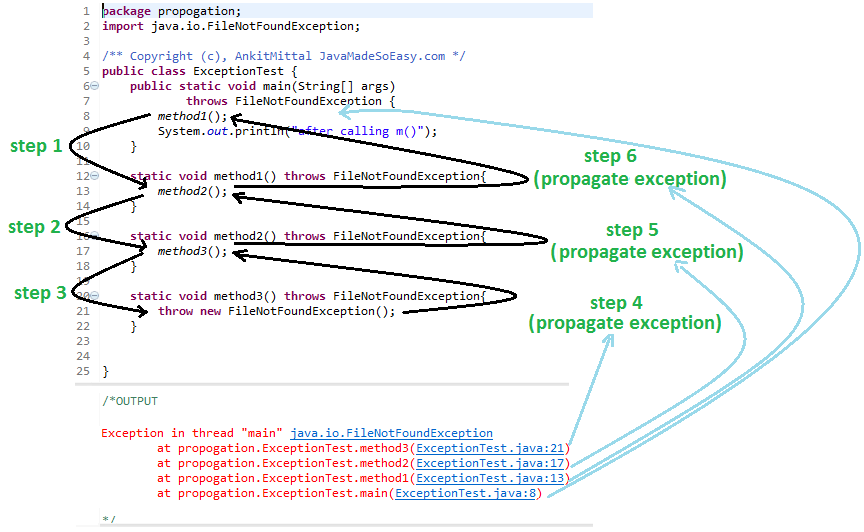


In the above program, stack is formed and an exception is first thrown from the top of the stack [ method3() ] and it remains uncaught there, and starts coming down the stack to previous methods to method2(), then to method1(), than to main() and it remains uncaught throughout.

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.



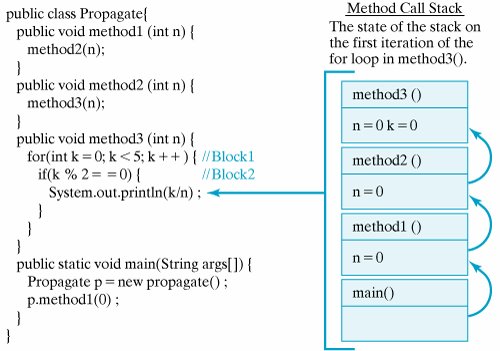
***Now, i’ll be explaining you how checked exception*** *was* ***propagated****.*

***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:**

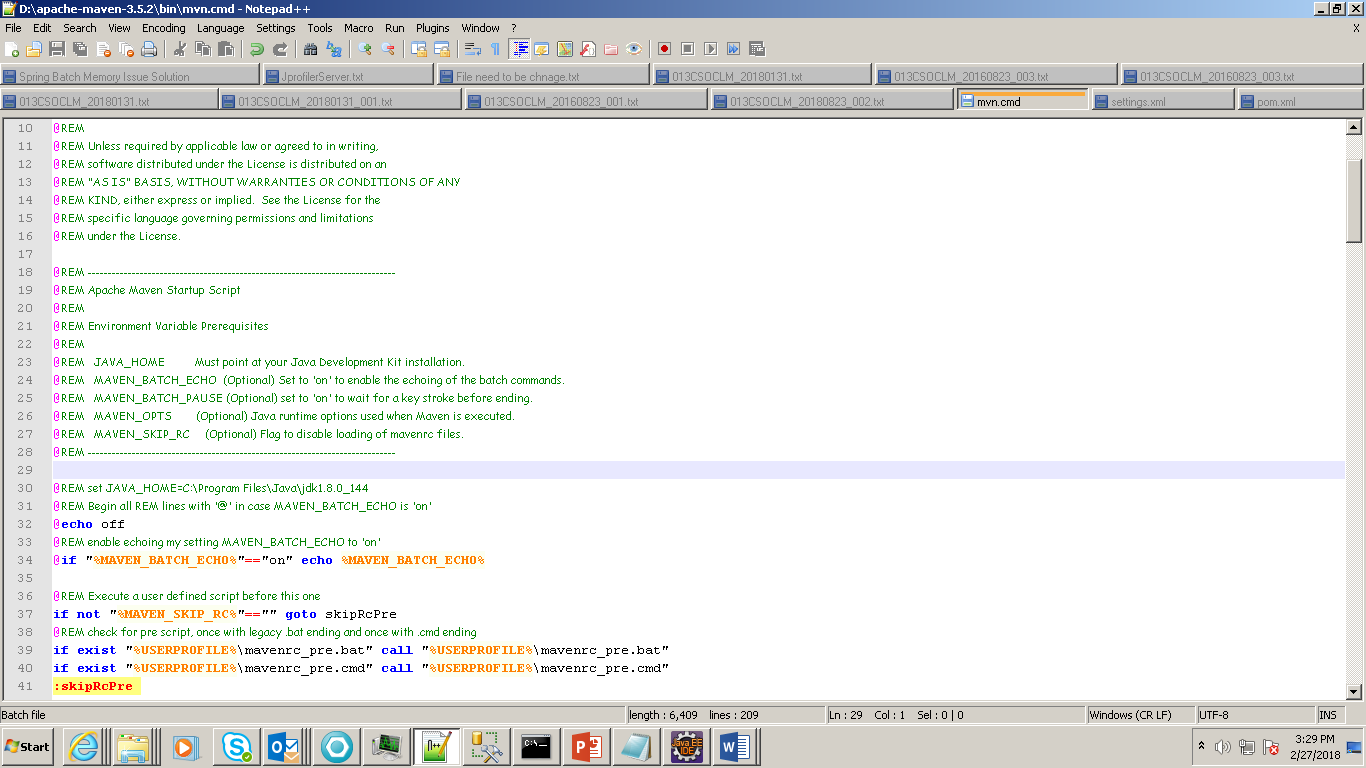
Set **JAVA\_HOME** and **M2\_HOME** and **MAVEN\_HOME**

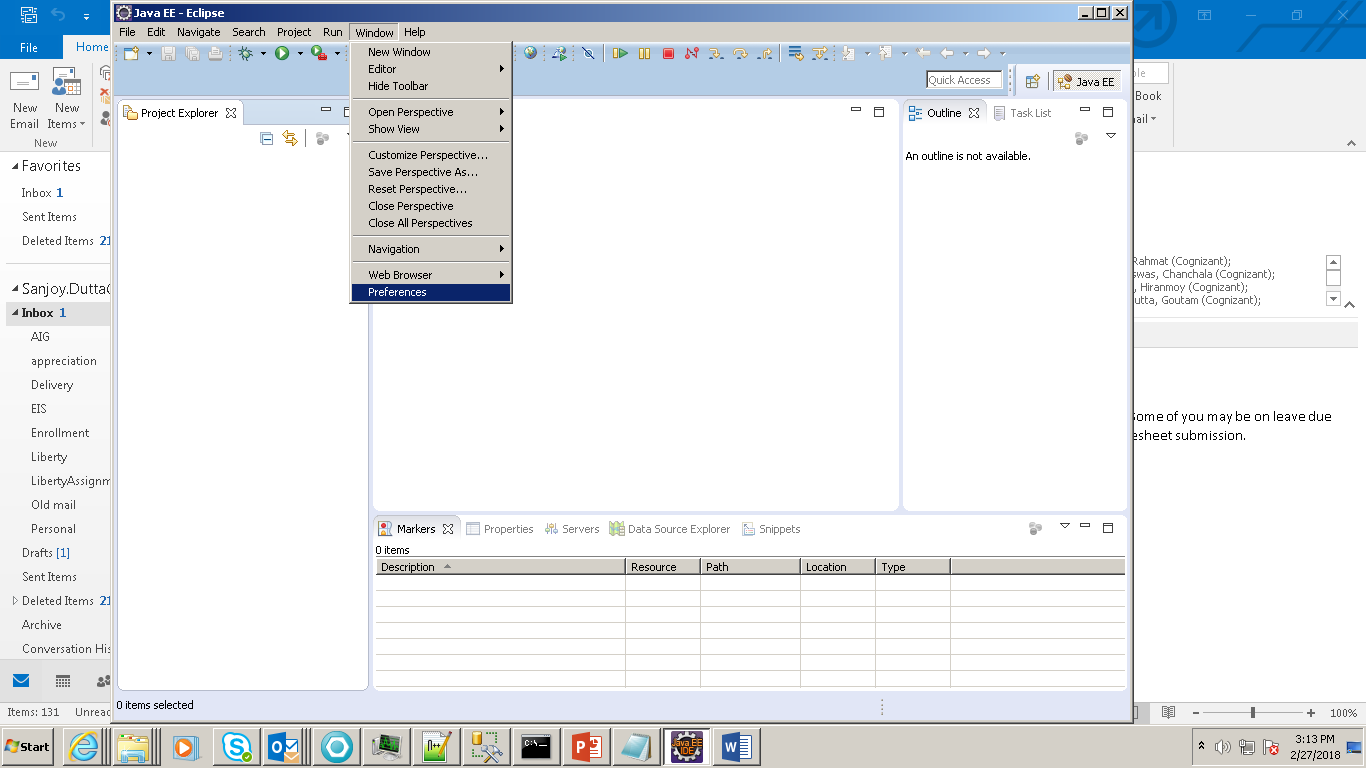
And run in command line to see the every thing works fine or not:

**Java –version**

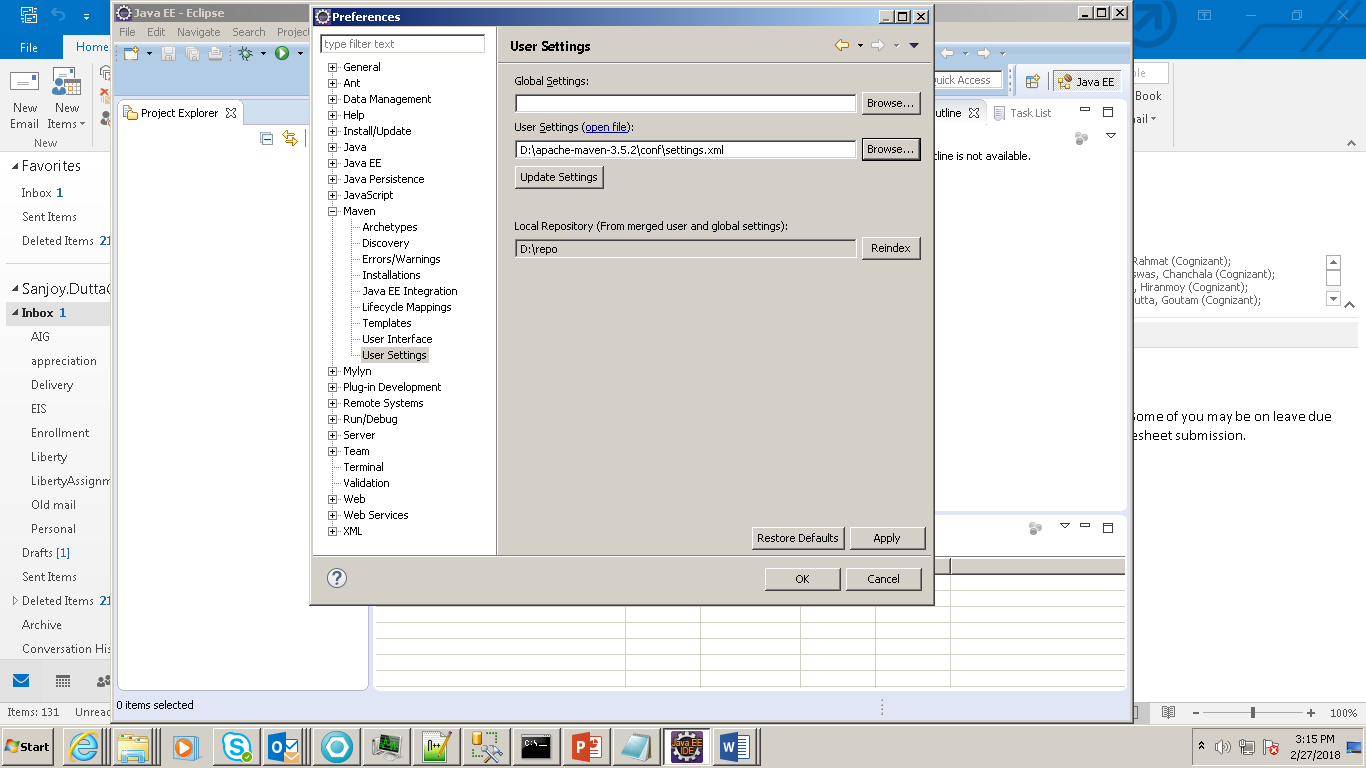
**Mvn –version**

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

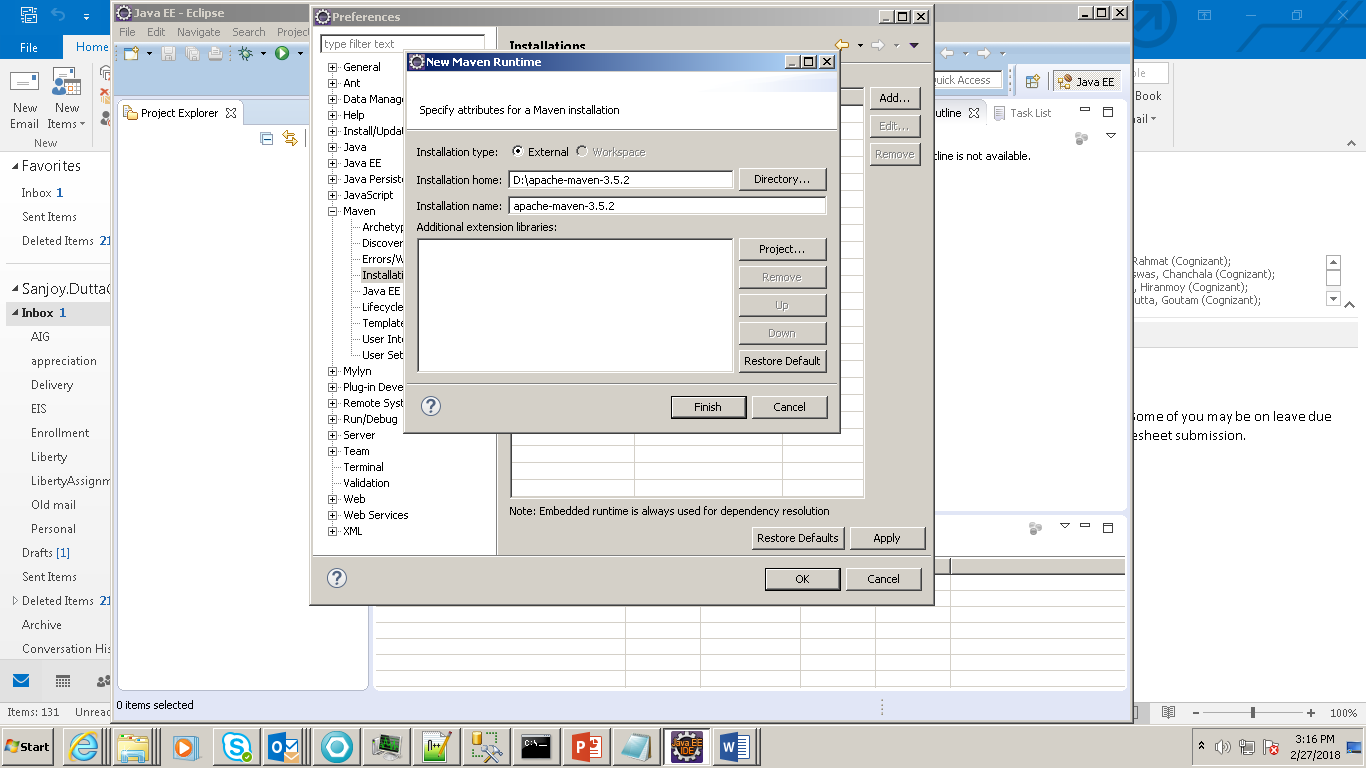




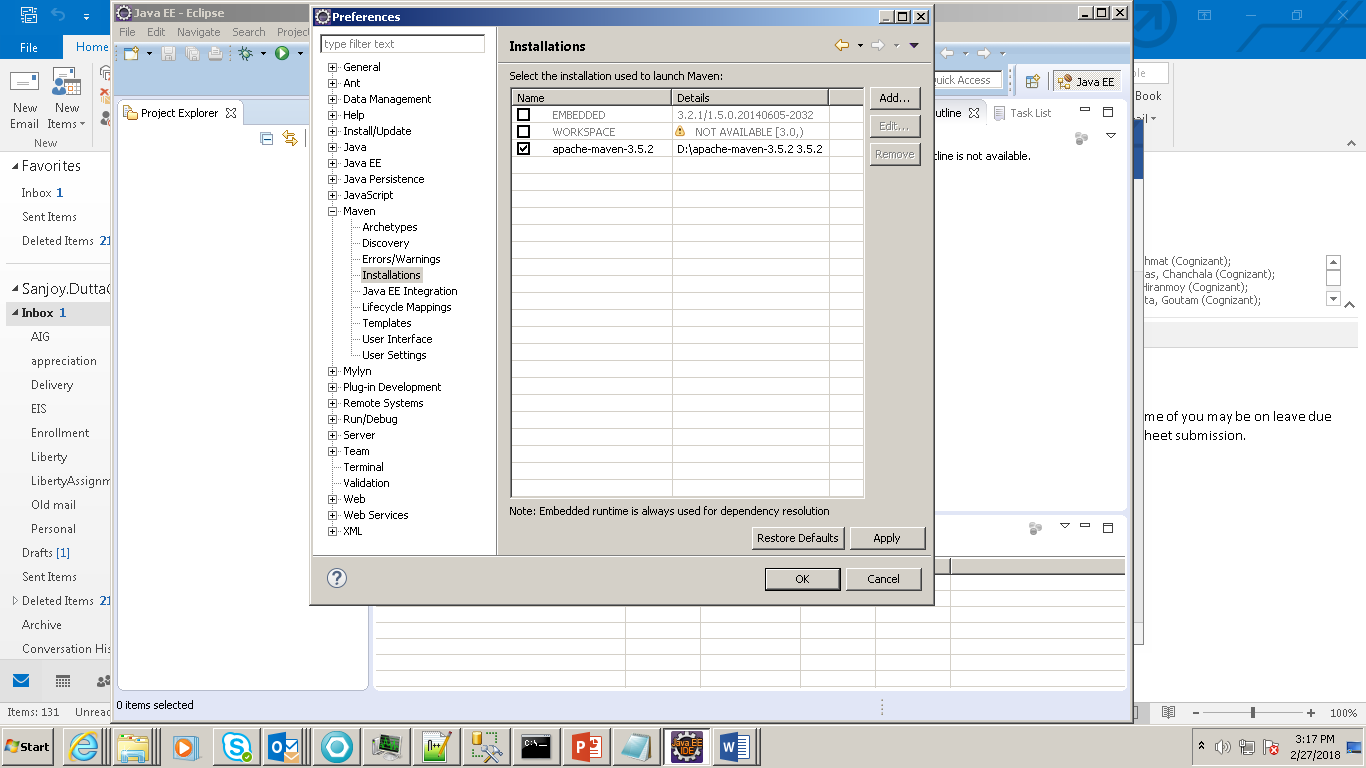
2.



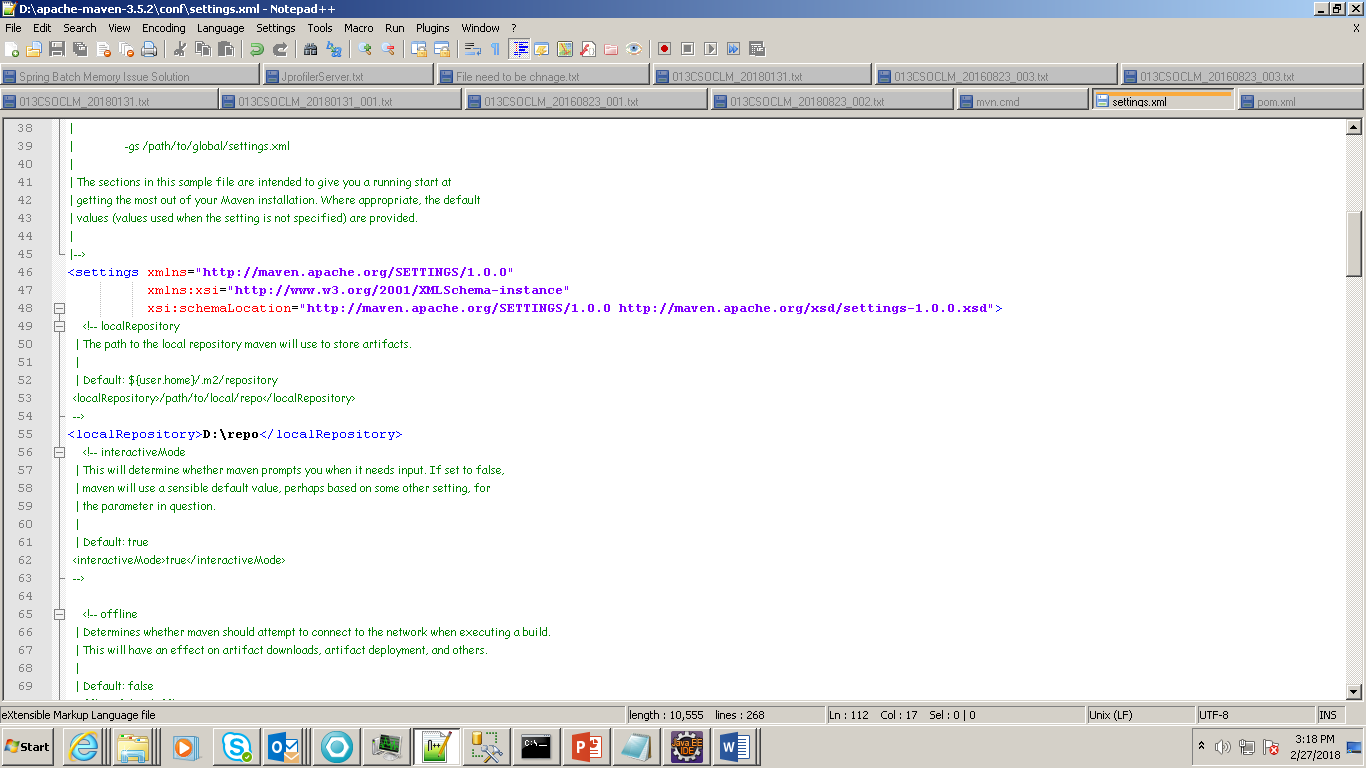
**Click on Installation and Add**

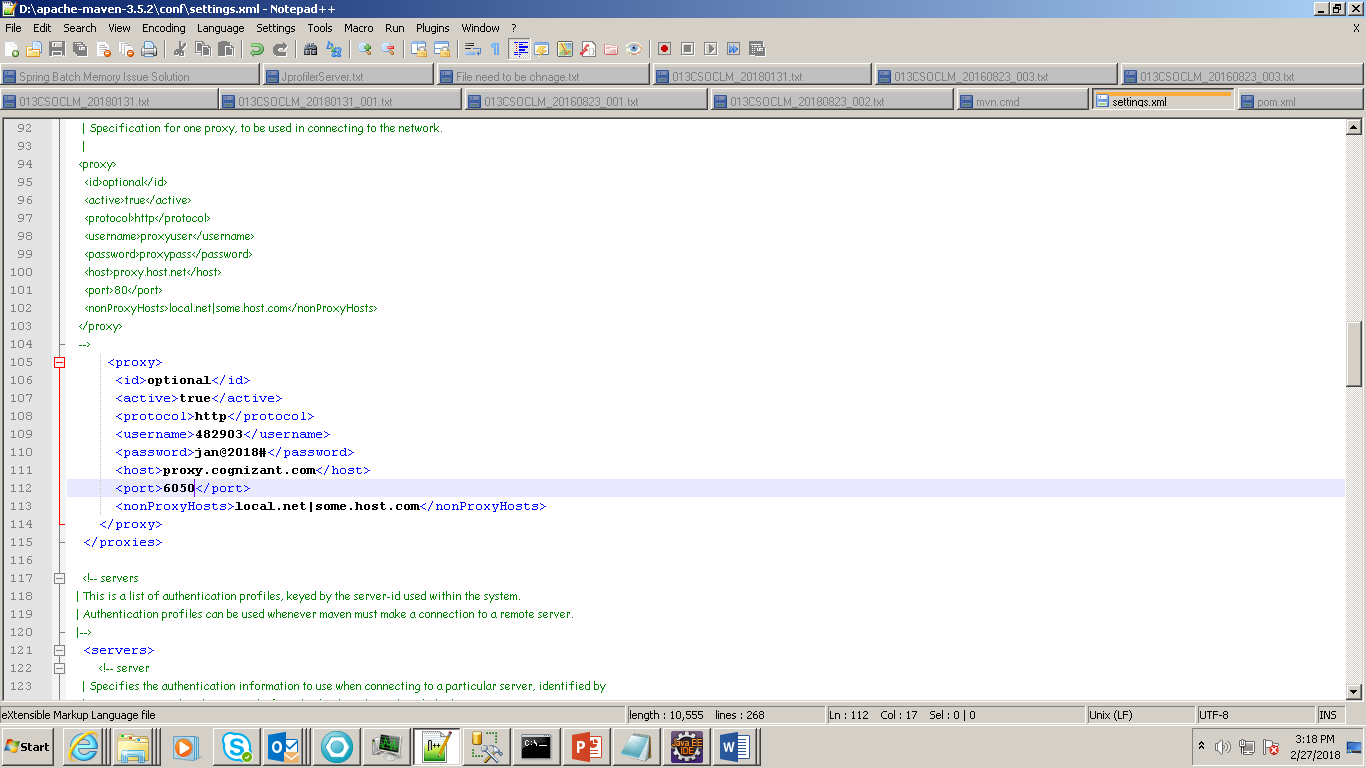


**Click on checkbox and apply**



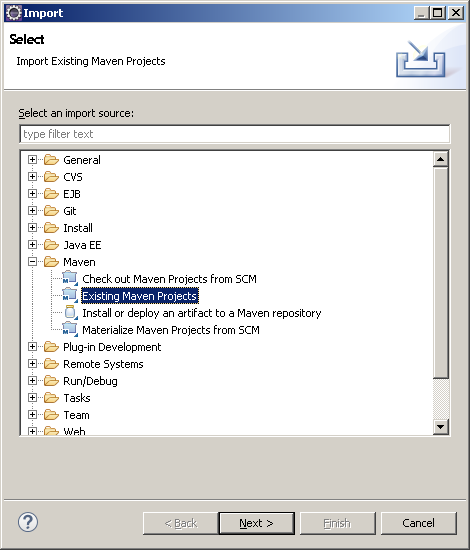
**Change in setting.xml of Maven:**





**For home there is no proxy**.

**Now Import the Maven project:**



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

**mvn clean install**