General Topics

Table Of Contents

[1. Serialization and Deserialization : 4](#_Toc512972075)

[2. Externalization : 7](#_Toc512972076)

[3. Marker Interface: 13](#_Toc512972077)

[4. Instance of Operator: 13](#_Toc512972078)

[5. Concurrent Modification Exception: 14](#_Toc512972079)

[6. Immutable feature 15](#_Toc512972080)

[6.1 Immutable class creation 15](#_Toc512972081)

[7. Enumeration 17](#_Toc512972082)

[8. Enumeration JMS API of Backend Adapters 17](#_Toc512972083)

[9. Design Patterns 17](#_Toc512972084)

[9.1 Creational Design Pattern 17](#_Toc512972085)

[9.2 Structural Design Pattern 18](#_Toc512972086)

[9.3 Behavioural Design Pattern 18](#_Toc512972087)

[10. Static Blocks and Constructor execution hierarchy 18](#_Toc512972088)

[11. Memory Leak 18](#_Toc512972089)

[1. Review Topics: 20](#_Toc512972090)

*Questions*

# Serialization and Deserialization :

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.

1. If a parent class has implemented Serializable interface then child class doesn’t need to implement it but vice-versa is not true.

2. Only non-static data members are saved via Serialization process.

3. Static data members and transient data members are not saved via Serialization process. So, if you don’t want to save value of a non-static data member then make it transient.

4. Constructor of object is never called when an object is deserialized.

5. Associated objects must be implementing Serializable interface.

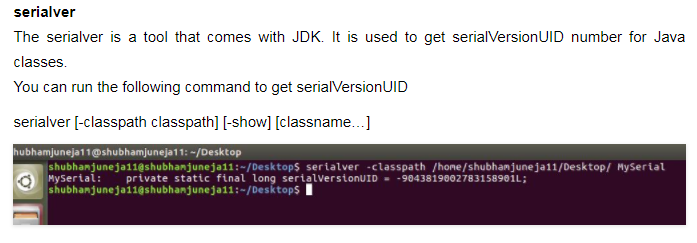
The Serialization runtime associates a version number with each Serializable class called a SerialVersionUID, which is used during Deserialization to verify that sender and reciever of a serialized object have loaded classes for that object which are compatible with respect to serialization. If the reciever has loaded a class for the object that has different UID than that of corresponding sender’s class, the Deserialization will result in an InvalidClassException. A Serializable class can declare its own UID explicitly by declaring a field name.

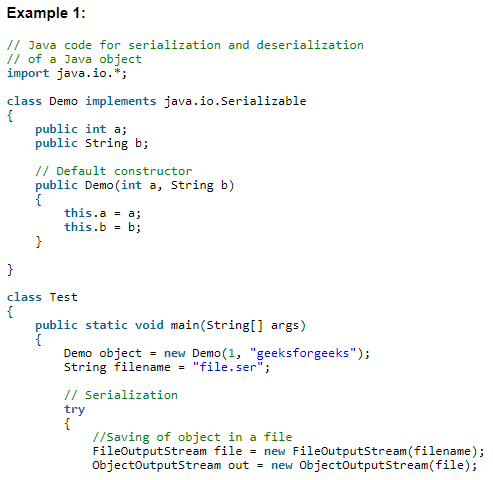
It must be static, final and of type long.

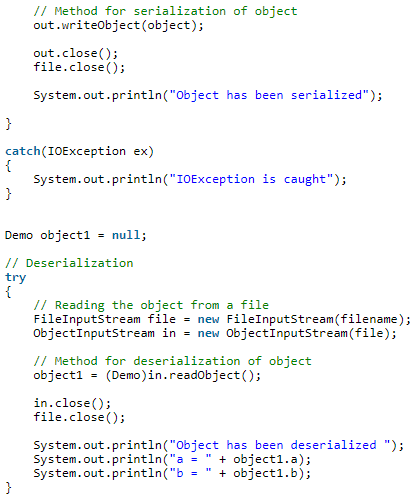
i.e- ANY-ACCESS-MODIFIER static final long serialVersionUID=42L;

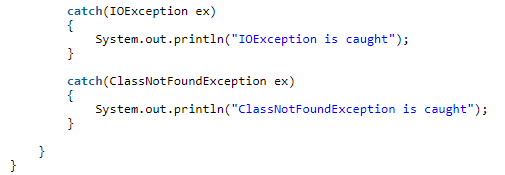
If a serializable class doesn’t explicitly declare a serialVersionUID, then the serialization runtime will calculate a default one for that class based on various aspects of class, as described in Java Object Serialization Specification. However it is strongly recommended that all serializable classes explicitly declare serialVersionUID value, since its computation is highly sensitive to class details that may vary depending on compiler implementations, any change in class or using different id may affect the serialized data.

It is also recommended to use private modifier for UID since it is not useful as inherited member.









*Note :* The Reason to Serialize the Named query class in EJB implemented project

This usually happens if you mix HQL and native SQL queries. In HQL, Hibernate maps the types you pass in to whatever the DB understands. When you run native SQL, then you must do the mapping yourself. If you don't, then the default mapping is to serialize the parameter and send it to the database (in the hope that it does understand it

1. <http://www.jusfortechies.com/java/core-java/externalization.php>

# Externalization :

Before going into what externalization is, you need to have some knowledge on what serialization is because externalization is nothing but serialization but an alternative for it and Externalizable interface extends Serializable interface. Check Serialization article for information on serialization. Just as an overview, Serialization is the process of converting an object's state (including its references) to a sequence of bytes, as well as the process of rebuilding those bytes into a live object at some future time. Serialization can be achieved by an object by implementing Serializable interface or Externalizable interface.

Well, when serialization by implementing Serializable interface is serving your purpose, why should you go for externalization?

Good question! Serializing by implementing Serializable interface has some issues. Lets see one by one what they are.

Serialization is a recursive algorithm. What I mean to say here is, apart from the fields that are required, starting from a single object, until all the objects that can be reached from that object by following instance variables, are also serialized. This includes the super class of the object until it reaches the "Object" class and the same way the super class of the instance variables until it reaches the "Object" class of those variables. Basically all the objects that it can read. This leads to lot of overheads. Say for example, you need only car type and licence number but using serialization, you cannot stop there. All the information that includes description of car, its parts, blah blah will be serialized. Obviously this slows down the performance.

Both serializing and deserializing require the serialization mechanism to discover information about the instance it is serializing. Using the default serialization mechanism, will use reflection to discover all the field values. Also the information about class description is added to the stream which includes the descption of all the serializable superclasses, the description of the class and the instance data associated with the specific instance of the class. Lots of data and metadata and again performance issue.

You know that serialization needs serialVersionUID, a unique Id to identify the information persisted. If you dont explicitly set a serialiVersionUID, serialization will compute the serialiVersionUID by going through all the fields and methods. So based on the size of the class, again serialization mechanism takes respective amount of time to calculate the value. A third performance issue.

Above three points confirm serialization has performance issues. Apart from performance issues,

When an object that implements Serializable interface, is serialized or de-serialized, no constructor of the object is called and hence any initialization which is done in the constructor cannot be done. Although there is an alternative of writing all initialization logic in a separate method and call it in constructor and readObject methods so that when an object is created or deserialized, the initialization process can happen but it definitely is a messy approach.

The solution for all the above issues is Externalization. Cool. Here enters the actual topic.

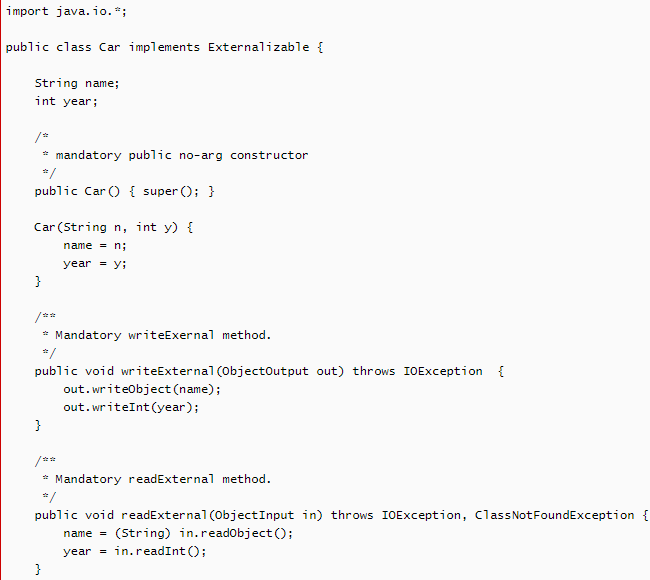
So what is externalization?

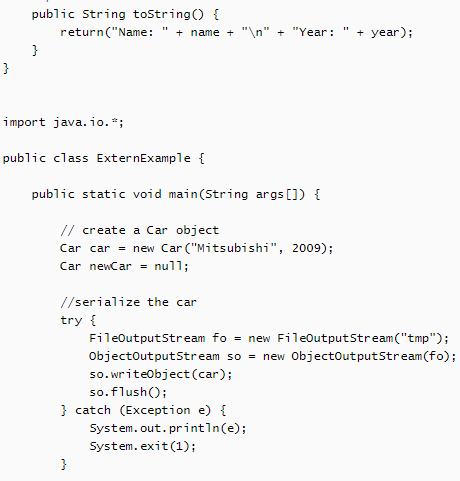
Externalization is nothing but serialization but by implementing Externalizable interface to persist and restore the object. To externalize your object, you need to implement Externalizable interface that extends Serializable interface. Here only the identity of the class is written in the serialization stream and it is the responsibility of the class to save and restore the contents of its instances which means you will have complete control of what to serialize and what not to serialize. But with serialization the identity of all the classes, its superclasses, instance variables and then the contents for these items is written to the serialization stream. But to externalize an object, you need a default public constructor.

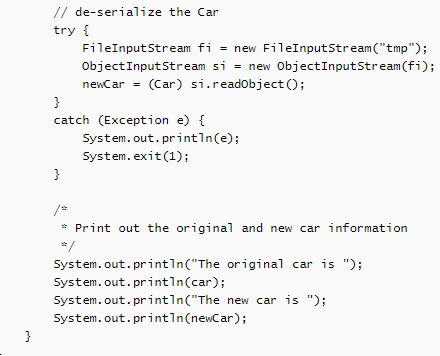
Unlike Serializable interface, Externalizable interface is not a marker interface and it provides two methods - writeExternal and readExternal. These methods are implemented by the class to give the class a complete control over the format and contents of the stream for an object and its supertypes. These methods must explicitly coordinate with the supertype to save its state. These methods supersede customized implementations of writeObject and readObject methods.

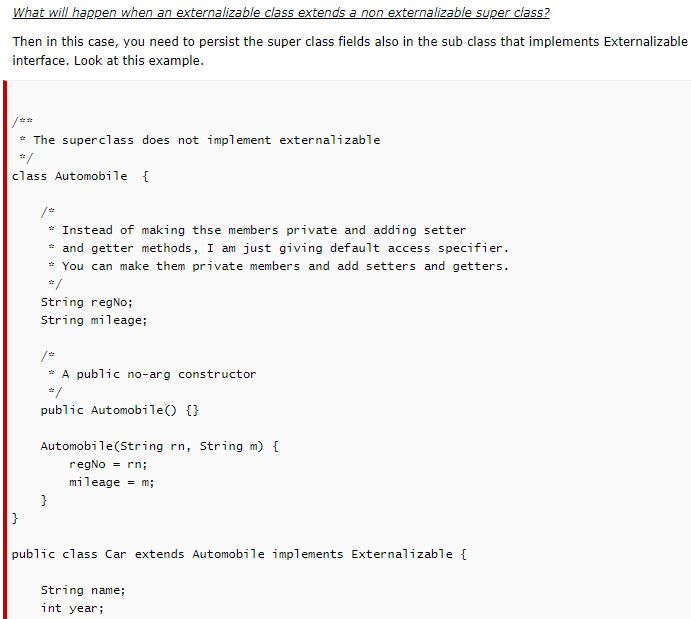
How serialization happens? JVM first checks for the Externalizable interface and if object supports Externalizable interface, then serializes the object using writeExternal method. If the object does not support Externalizable but implement Serializable, then the object is saved using ObjectOutputStream. Now when an Externalizable object is reconstructed, an instance is created first using the public no-arg constructor, then the readExternal method is called. Again if the object does not support Externalizable, then Serializable objects are restored by reading them from an ObjectInputStream.

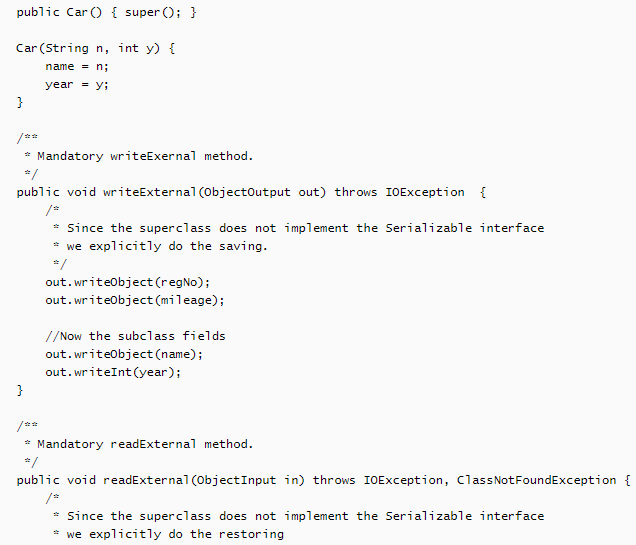
Lets see a simple example.

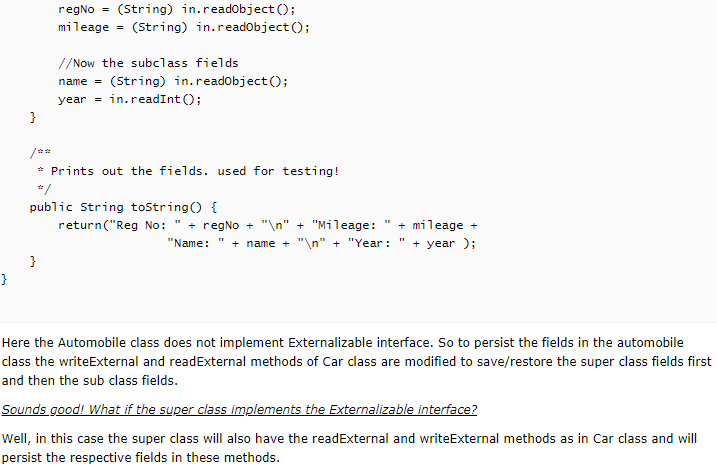












Ref Link for Externalization : <http://www.jusfortechies.com/java/core-java/externalization.php>

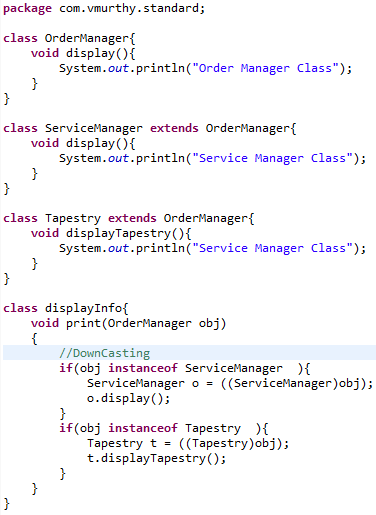
# Marker Interface:

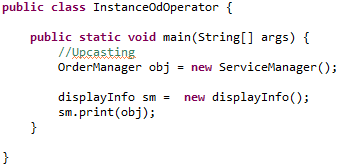
As the name suggests it can used to define a class to be considered for specific category. For example if we have hundred classes we can define say 10 of such classes as serializable by implementing marker interface.

# Instance of Operator:

The operator is used to validate the type of any given variable. The real time use of the operator would be to invoke specific methods based on the type of the object.

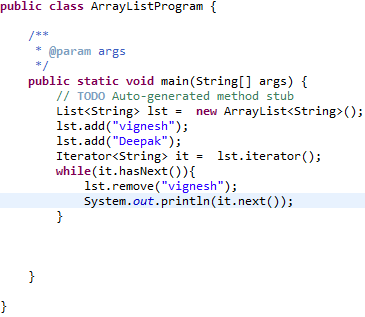
In Real time if we have two objects implementing the same interface, and if an implementation is to be devised to invoke specific methods of the class based on the type of the object, we would make use of instance of operator to accomplish the same.



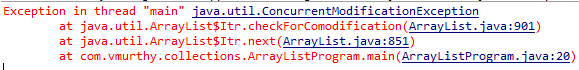


# Concurrent Modification Exception:

Upon initiating traversal on any given collection through iterator and during the traversal if any new element is added or removed the action results in “ConcurrentModificationException”.



*Error:*



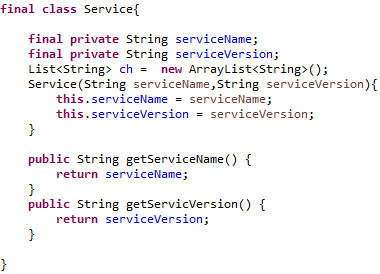
# Immutable feature

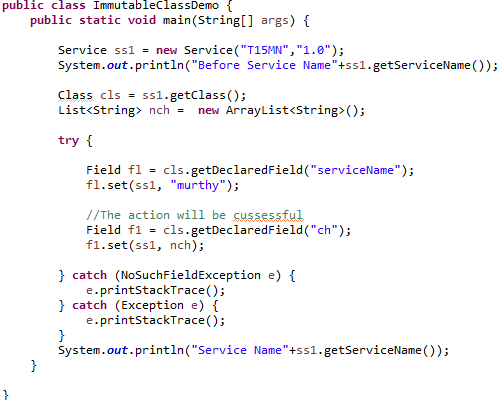
Java String contained in package “*java.lang.String*” is immutable meaning it cannot be changed once it is created and any further change will create a new instance without modifying the existing one.

## 6.1 Immutable class creation

The following rules help us in creating an immutable class.

1. Declare all the member variables of the class ass private and final by doing so the fields will not participate in inheritance and the final key would enforce restriction on the initialization of the fields (it should be done at the time of declaration or during instantiation through constructor).
2. Only the getter methods need to be implemented for the member variables and no setter methods.
3. In order to avoid the subclasses from overriding the member functions define the class as final.





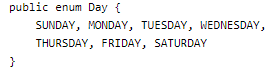
The assumption is that in spite of such constraints through reflection we will be able to access the fields but in Java 1.8 while doing so an exception is being generated stating that the field is final and there the action is forbidden.

# Enumeration

It is data type which enables a variable to be associated with a set of predefined constants. The variable must be equal to one of the defined constants. Common examples include planets in the solar system, directions in a compass.

The only pre requisite is that the programmer should be aware of all the possible values at compile time.

Example: Syntax for creating Enum in java



The enum declaration defines a class called enum type. The enum class body can have methods and other fields. The compiler automatically adds some methods when it creates an enum. For example static method ***values*** gets automatically populated within enum class.

Any enum creation by default extends the enum class as a result of which the enum cannot inherit any other classes.

# Enumeration JMS API of Backend Adapters

# Design Patterns

## 9.1 Creational Design Pattern

This design pattern is used as a guide line for class creation and object instantiation. The creational design pattern can be further divided into class creational pattern and object creational pattern.

Following are the different types of Creational design Pattern.

#### Factory Design Pattern

It is one among the commonly used pattern in java. It provides the best way to create an object. In Factory design Pattern we create an object without exposing the underlying logic to the client and we refer to the created object using a common interface.

*Project Implementation*: There exists a common interface for the CustomerProfileService stream. The factory method of the class “CreateProfileServiceObj” will create the appropriate object based on the incoming transactionfor example NewCustomerProfileIAF.

#### Abstract Factory Design Pattern

1. Abstract Factory Method.
2. Builder.
3. Singleton.
4. Object Pool Prototype.

## 9.2 Structural Design Pattern

This design Pattern is all about organizing different classes and objects to form larger structures and to provide new functionality.

Structural design patterns are Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Private Class Data and Proxy.

## 9.3 Behavioural Design Pattern

This design pattern helps in defining relationship between entities.

# Static Blocks and Constructor execution hierarchy

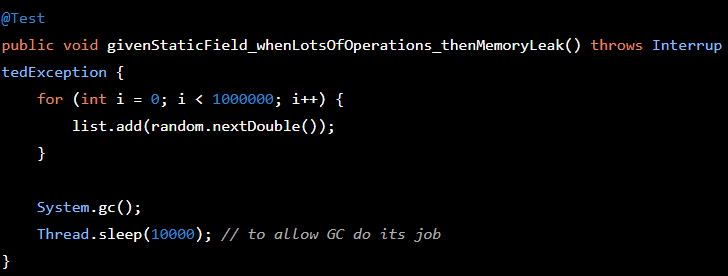
In a hierarchy where a sub class extends a master class the execution of the static and constructor blocks will have a specific pattern which is illustrated by the below mentioned example.

# Memory Leak

Definition: A memory leak is a scenario where an object with a huge size has been created. When the object is no longer used the garbage collector is unable to remove it from the working memory.

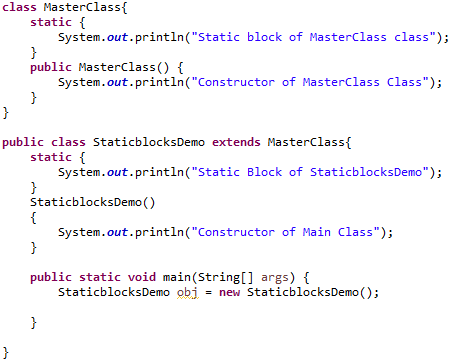
A simple example to trigger a simple memory leak is as defied below.





In the above mentioned example even after the call to garbage collector the space acquired by the static list object will not be reclaimed as it tagged to the lifecycle of the JVM itself. If the same variable is declared as a non-static variable once the variable is out of scope the garbage collector reclaims the space.

The Un-closed connection or read write operations may also result in Memory leak.



Output:



# Brute Force Method

The prime example for Brute force method is String matching algorithms where we have to identify if the given string available in a line of text. Using brute force method we will have to compare every character of the pattern with the character of the text if we find a match then match the further characters in the text if not we once again match the first character of the pattern with the character in the next position.

This would incur more time compared to other searching algorithms. Efficient algorithm might first perform indexing of the text and then start the search but in scenario where the search is a onetime event the indexing might not be required and in such requirement brute Force method will serve its purpose.

# Top Down and Bottom up Approach

The top down approach takes the high level definition where a problem is divided into sub problem and we continue to do so until we reach a level where the solution can be devised. This is often associated to functional decomposition style of programming.

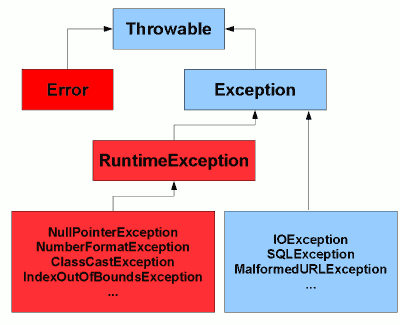
In Bottom up approach we combine the solution of the smaller module to present a solution to a higher level requirement.

In reality almost all programming is done with the combination of both the approach where we divide the requirement in to smaller tasks and combine the results of the smaller to task to address the solution to the original requirement.

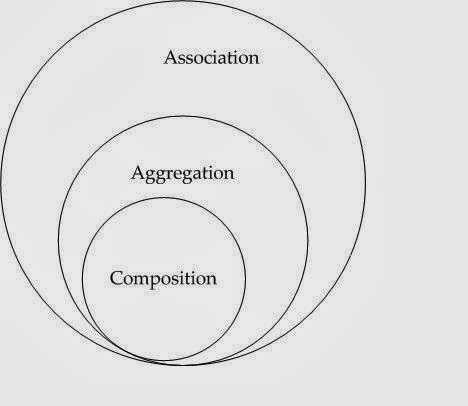
# Merge Sort

The sorting mechanism is based on “*divide and conquer*” approach. We start dividing the array into sub array recursively until we are left with only one element and then we start comparing the divided elements and merge them in sorted manner to for the original array.

# Exception Hierarchy



# Association, Aggregation, Composition and Inheritance.



Inheritance represents “IS-A” relationship while the Composition represents “HAS-A” relation ship.

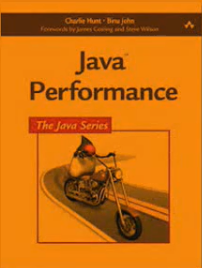
# Future Topics

|  |  |  |
| --- | --- | --- |
| ***SI NO*** | ***Topic*** | ***Status*** |
| 1 | Writing Custom Hash Map |  |
| 2 | Custom String class which creates objects in the same way as that of String constant pool(immutable) |  |
| 3 | Enhance hashmap to store values in ascending or descending order. |  |
| 4 | Enhance hashmap to store keys in ascending or descending order. |  |
| 5. | Memory modem of latest java version. |  |
| 6. | Custom Hash Map, Custom Iterator |  |
| 7. | OutOfMemory Exception : What, How to analyse and how to resolve |  |
| 8. | Concept of Transaction and session Management |  |
| 9. | How to implement Data source. |  |
|  |  |  |
|  |  |  |

# Future Subject

|  |  |  |
| --- | --- | --- |
| ***SI NO*** | ***Topic*** | ***Status*** |
| 1 | Basic Restful Webservice, Advanced Restful webservice |  |
| 2 | Maven |  |
| 3 | Junit |  |
| 4 | Spring Security |  |
| 5. | Java Script |  |
| 6. | Database Concepts |  |
| 7. | Class Designing Tool |  |
| 8. |  |  |
| 9. |  |  |
|  |  |  |
|  |  |  |

Java performance :



JVM architecture : chapter 3,4,7