Java Notes

1. Write a short note on Java language and its features?

Java is a programming language. It was first developed by **James Gosling** at **Sun Microsystems**, which is now a part of Oracle Corporation. It is a ***general purpose, high level*** programming language which is **completely object oriented**.
Following are the features of Java Programming language:

1. **Simple**
2. **Secure**
3. **Portable**
4. **Object-oriented**
5. **Robust**
6. **Multithreaded**
7. **Architecture-neutral**
8. **Interpreted**
9. **High performance**
10. **Distributed**
11. **Dynamic**

Simple
Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object oriented features of C++, most programmers have little trouble learning Java

Secure
Java is secured because:

- No explicit pointer
- Programs run inside virtual machine sandbox.
- **Classloader**- adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier**- checks the code fragments for illegal code that can violate access right to objects.
- **Security Manager**- determines what resources a class can access such as reading and writing to the local disk.

These security are provided by java language. Some security can also be provided by application developer through SSL,JAAS,cryptography etc.

Portable
We may carry the java bytecode to any platform. the intermediate byte code is platform and architecture neutral and can be excuted on any machine (e.g. windows, linux, mac)

Object-Oriented
Unlike C++, Java is a completely object oriented programming language i.e. everything in java is written inside a class.
In Java, everything is an Object. Java can be easily extended since it is based on the Object model.

Robust
The multiplatformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. Many hard to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java. Knowing that what you have written will behave in a predictable way under diverse conditions is a key feature of Java.

Multithreaded
Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run time system comes with an elegant yet sophisticated solution for multi-process synchronization that enables you to construct smoothly running interactive systems. Java's easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem

Architectural-neutral
Java compiler generates an architecture-neutral object file format which makes the compiled code to be executable on many processors, with the presence of Java runtime system.

Interpreted
Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine. The Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform independent code

High Performance
Java is faster than traditional interpretation since byte code is "close" to native code still somewhat slower than a compiled language (e.g., C++)

Distributed
We can create distributed applications in java. RMI and EJB are used for creating distributed applications. We may access files by calling the methods from any machine on the internet.

Dynamic
Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

2. Define constructor in Java and its types? Constructor overloading?

A constructor is a special member function of a class which is called automatically when an object of that class is created (default constructor).
A constructor has following properties

- It has same name as its class
- It does not have a return type, not even void
- It cannot be static
- It is generally public

Constructors are used to initialize the instance variables of an object.
It is constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, ready to use object immediately
Example code:

```
package com.myjava.constructors;
public class MyDefaultConstructor
{
	public MyDefaultConstructor()
	{
		System.out.println("I am inside default constructor...");
	}
	public static void main(String a[])
	{
		MyDefaultConstructor mdc = new MyDefaultConstructor();
	}
}
```

Types of constructors:

- Default constructor

- Parameterized constructor

➢ Default Constructor:
  - A constructor which does not take any arguments as parameters is known as default constructor.
  - It is primarily used to initialize the object with default initial values.
  - When no default constructor is defined by users explicitly, java provides its own version of default constructor to initialize the object

➢ Parameterized constructor
  - A constructor that consists of one or more values passed as arguments in called parameterized constructor.
  - This constructor is primarily used to initialize object's member variables with specific values that can be passes as arguments.

Constructor Overloading

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists.The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

Example of Constructor Overloading:

```
class Student5{
   int id;
   String name;
   int age;
   Student5(int i,String n){
   id = i;
   name = n;
   }
   Student5(int i,String n,int a){
   id = i;
   name = n;
   age=a;
   }
   void display(){System.out.println(id+" "+name+" "+age);}

   public static void main(String args[]){
   Student5 s1 = new Student5(111,"Karan");
   Student5 s2 = new Student5(222,"Aryan",25);
   s1.display();
   s2.display();
  }
}
```

3. Inheritance in Java? Types of Inheritance? Why is multiple Inheritance not supported in Java? Explain with example.

- Inheritance is one of the key features of object-oriented programming because it allows the creation of hierarchical classifications.
- Inheritance in java is a mechanism in which one object acquires all the properties and behaviors of parent object.

- The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.

Inheritance has advantages like:

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

In the terminology of Java, a class that is inherited is called a **super class**. The new class is called a **subclass**.

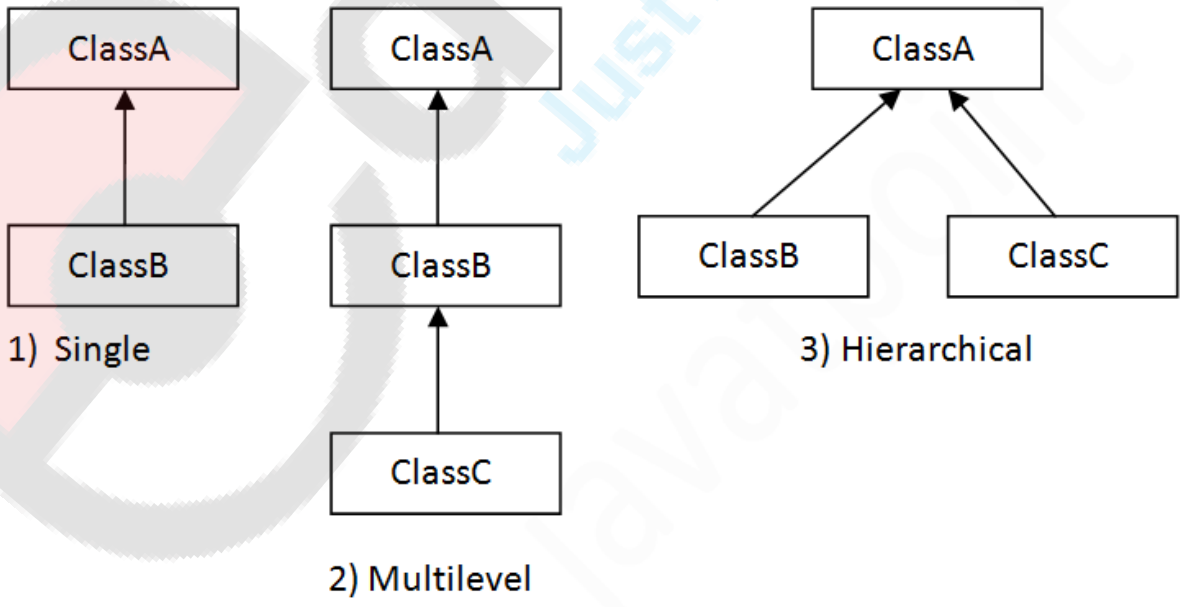
Syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

The **extends** keyword indicates that you are making a new class that derives from an existing class.

Types of inheritance in java

On the basis of class, there can be **3 types** of inheritance in java: **single**, **multilevel** and **hierarchical**. In java programming, multiple and hybrid inheritance is supported through interface only.



Reason why multiple inheritance is not supported in Java:

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

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

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

Program Code:

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

 Public Static void main(String args[]){
   C obj=new C();
   obj.msg();//Now which msg() method would be invoked?
}
}
```

Output:

Error

4. Short note on static member variables, static member functions and static block in Java?

- The **static** keyword in java is used for memory management mainly.
- We can apply java static keyword with variables, methods, blocks and nested class.
- The static keyword belongs to the class than instance of the class.

The static can be:

- **variable (also known as class variable)**
- **method (also known as class method)**
- **block**
- **nested class**

1) Java static variable

- If you declare any variable as static, it is known **static** variable.
- The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees,college name of students etc.

- The static variable gets memory only once in class area at the time of class loading.

Advantage of static variable

It makes your program memory efficient (i.e it saves memory).

Example of static variable

```
//Program of static variable
class Student8{
  int rollno;
  String name;
  static String college ="ITS";

  Student8(int r,String n){
  rollno = r;
  name = n;
  }
 void display (){System.out.println(rollno+" "+name+" "+college);}

 public static void main(String args[]){
 Student8 s1 = new Student8(111,"Karan");
 Student8 s2 = new Student8(222,"Aryan");

 s1.display();
 s2.display();
 }
}
```

Output:

```
111 Karan ITS
222 Aryan ITS
```

2) Java static method

- If you apply static keyword with any method, it is known as static method.
- A static method belongs to the class rather than object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- static method can access static data member and can change the value of it.

Example of static method

```
//Program to get cube of a given number by static method
class Calculate{
 static int cube(int x){
 return x*x*x;
 }

 public static void main(String args[]){
 int result=Calculate.cube(5);
 System.out.println(result);
```

```
 }
}
```

There are two main restrictions for the static method. They are:

- The static method can not use non static data member or call non-static method directly.
- ***this*** and ***super*** cannot be used in static context.

3) Java static block

- Is used to initialize the static data member.
- It is executed before main method at the time of classloading.

Example of static block

```
class A2{
  static{System.out.println("static block is invoked");}
  public static void main(String args[]){
   System.out.println("Hello main");
  }
}
```

Output:

```
static block is invoked
Hello main
```

5. Final keyword in Java and its uses?

The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

- **variable**
- **method**
- **class**

- The final keyword can be applied with the variables, a final variable that have no value it is called **blank final variable** or **uninitialized final variable**.
- It can be initialized in the constructor only.
- The blank final variable can be static also which will be initialized in the static block only.

1) Java final variable

- If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}//end of class
```

Output:
Compile Time Error

2) Java final method

- If you make any method as final, you cannot override it.
- final method is inherited but you cannot override it.

Example of final method

```
class Bike{
  final void run(){System.out.println("running");}
}

class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}

   public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
   }
}
```

Output:
Compile Time Error

3) Java final class

If you make any class as final, you cannot extend it.

Example of final class

```
final class Bike{}
class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda1 honda= new Honda();
  honda.run();
  }
}
```

Output:
Compile Time Error

6. Super keyword in Java and its uses?

- The super keyword in java is a reference variable that is used to refer immediate parent class object.
- Whenever you create the instance of subclass, an instance of parent class is created implicitly i.e. referred by super reference variable.

Usage of java super Keyword

- **super is used to refer immediate parent class instance variable.**
- **super() is used to invoke immediate parent class constructor.**
- **super is used to invoke immediate parent class method.**

1) super is used to refer immediate parent class instance variable.

```
//example of super keyword
class Vehicle{
  int speed=50;
}

class Bike4 extends Vehicle{
  int speed=100;

  void display(){
   System.out.println(super.speed);//will print speed of Vehicle now
  }
  public static void main(String args[]){
   Bike4 b=new Bike4();
   b.display();

}
}
```

Output:50

2) super is used to invoke parent class constructor.

```
class Vehicle{
  Vehicle(){System.out.println("Vehicle is created");}
}

class Bike5 extends Vehicle{
  Bike5(){
   super();//will invoke parent class constructor
   System.out.println("Bike is created");
  }
  public static void main(String args[]){
   Bike5 b=new Bike5();
```

```
}
}
```
Output:
Vehicle is created
Bike is created

3) super can be used to invoke parent class method
The super keyword can also be used to invoke parent class method. It should be used in case subclass contains the same method as parent class as in the example given below:

```
class Person{
void message(){System.out.println("welcome");}
}

class Student16 extends Person{
void message(){System.out.println("welcome to java");}

void display(){
message();//will invoke current class message() method
super.message();//will invoke parent class message() method
}

public static void main(String args[]){
Student16 s=new Student16();
s.display();
}
}
```
Output:
welcome to java
welcome

7. *this* keyword in Java and its uses?

There can be a lot of usage of java this keyword. In java, this is a reference variable that refers to the current object.

Usage of java this keyword

Here is given the 6 usage of java this keyword.

- this keyword can be used to refer current class instance variable.
- this() can be used to invoke current class constructor.
- this keyword can be used to invoke current class method (implicitly)
- this can be passed as an argument in the method call.

- this can be passed as argument in the constructor call.
- this keyword can also be used to return the current class instance.

```
//example of this keyword
class Student11{
    int id;
    String name;

    Student11(int id,String name){
    this.id = id;
    this.name = name;
    }
    void display(){System.out.println(id+" "+name);}
    public static void main(String args[]){
    Student11 s1 = new Student11(111,"Karan");
    Student11 s2 = new Student11(222,"Aryan");
    s1.display();
    s2.display();
}
}
```

8. What is Package in Java? Explain with example? Levels of Access Protection for packages?
    - Packages are containers for classes.
    - Packages are primarily used to keep the class name space compartmentalized and provide a mechanism for partitioning the class name space into more manageable chucks
    - Packages is both a naming convention and visibility control mechanism
    - You can define class members that are exposed only to other members of the same package
    - This Allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world
    - Defining a package:
        - To create a package simply use the package command/keyword

    This is the general form of package statement:
    package *mypack;*
    Java uses file system directories to store packages. For example the **.class** files for any classes you declare to be part of mypack must be stored in the directory mypack. Also it is case sensitive.
    You can create a hierarchy of packages, to do so simply separate each package name from the one above it by using a period(**.**)
    The general form of multilevel package is:
    Package pkg1**.**pkg2**.**pkg3;

    Access Protection in packages

Packages add another dimension to access control.
Packages act as containers for classes and other subordinate packages.
Following is the access control mechanism provided by packages along with classes in Java:

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

- The three access specifiers, private, public, and protected, provide a variety of ways to produce the many levels of access required by these categories.
- Anything declared public can be accessed from anywhere.
- Anything declared private cannot be seen outside of its class.
- When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package.
- If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element protected.
- A non-nested class has only two possible access levels: default and public. When a class is declared as public, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package. When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class.

9. How do we achieve multiple inheritance in java? Explain with java program? (Interfaces in java)?
    - The functionality of **multiple inheritance** is achieved by using Interfaces in Java.
    - Interfaces are designed to support **dynamic method resolution** at **run time**.
    - Using the keyword ***interface***, you can fully abstract a class's interface from its implementation. That is, using interface, you can specify what a class must do, but not how it does it.
    - Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.

- Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.

An interface is defined much like a class. This is the general form of an interface:

access interface *name* {
return-type method-name1(parameter-list);
return-type method-name2(parameter-list);
type final-varname1 = value;
type final-varname2 = value;
// ...
return-type method-nameN(parameter-list);
type final-varnameN = value;
}

When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as public, the interface can be used by any other code.

Variables can be declared inside of interface declarations. They are implicitly final and static, meaning they cannot be changed by the implementing class. They must also be initialized. All methods and variables are implicitly public.

Here is an example of an interface definition:

interface Callback {
void callback(int param);
}

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface.

When you implement an interface method, it must be declared as public.

The general form of a class that includes the implements clause looks like this:

class *classname* [extends superclass] [implements interface [,interface...]] {
// class-body
}

Example Program:

```
interface MyInterface
{
  public void method1();
  public void method2();
}
class XYZ implements MyInterface
{
 public void method1()
 {
    System.out.println("implementation of method1");
```

```
  }
  public void method2()
  {
     System.out.println("implementation of method2");
  }
  public static void main(String arg[])
  {
     MyInterface obj = new XYZ();
     obj. method1();
  }
}
```

10. Difference between Interface and Abstract class?

Interface	Abstract class
Support multiple inheritance	Doesn't support multiple inheritance
Doesn't contain data members	Contains data members
Doesn't contain constructors	Contains constructors
Contains only signatures of member functions	It can contain function declarations as well has definitions
Cannot have access modifiers. By default everything is public	Can contain access modifiers for the subs, functions, properties
Members of interface cannot be static	Only complete members of abstract class can be static

11. Exception Handling in Java? Which are the types of exceptions? Explain try, catch, throw, throws keywords?

Answer:

- An exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error. Java provides exception handling mechanism to tackle such run time error situations.
- A Java exception is an object that describes an exceptional condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on.
- A Java exception is an object that describes an exceptional condition that has occurred in a piece of code.

- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.

Java exception handling is managed via five keywords: ***try, catch, throw, throws,*** and *finally*.

Working:

- Program statements that you want to monitor for exceptions are contained within a try block.
- If an exception occurs within the **try block**, it is thrown.
- Your code can **catch** this exception (**using catch**) and handle it in some rational manner.
- System-generated exceptions are automatically thrown by the Java run-time system.
- To manually throw an exception, use the keyword **throw**.
- Any exception that is thrown out of a method must be specified as such by a **throws clause**.
- Any code that absolutely must be executed after a try block completes is put in a finally block.

This is the general form of an exception-handling block:

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed after try block ends
}
```

Here, *ExceptionType* is the type of exception that has occurred.

There are 2 types of exceptions:

- Checked Exceptions
- Unchecked Exceptions

Checked Exceptions are those which are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword.

Unchecked are the exceptions that are not checked at compiled time. It is up to the programmers to be civilized, and specify or catch the exceptions.

Here are some exceptions defined by java:

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type
IndexOutOfBoundsException	Some type of index is out-of-bounds
NullPointerException	Invalid use of a null reference
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
ClassNotFoundException	Class not found

User Defined Exception Program:

```
// This program creates a custom exception type.
class MyException extends Exception
 {
        private int detail;
        MyException(int a)
        {
        detail = a;
        }
        public String toString() {
                return "MyException[" + detail + "]";
        }
}
class ExceptionDemo {
                static void compute(int a) throws MyException {
                System.out.println("Called compute(" + a + ")");
                if(a > 10)
                throw new MyException(a);
                System.out.println("Normal exit");
        }
        public static void main(String args[]) {
                try {
                        compute(1);
                        compute(20);
                }
                 catch (MyException e) {
                System.out.println("Caught " + e);
                }
        }
}
```

12. What is multithreading environment? OR Multithreading in Java with example?

- Java provides built-in support for multithreaded programming. A multithreaded program **contains two or more parts that can run concurrently**. Each part of such a program is called a **thread**, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.
- In a thread-based multitasking environment, the **thread is the smallest unit of dispatchable code.** This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.
- Multithreading enables you to write very efficient programs that make **maximum use of the CPU**, because idle time can be kept to a minimum. This is especially important for the interactive, networked environment in which Java operates, because idle time is common.

Multithreading in Java can be achieved by two ways:

- **By inheriting the Thread class OR**
- **By implementing the Runnable interface**

Inheriting the Thread Class:

- Here you have to inherit the Thread class properties into your custom class by extending the Thread class.
- After extending this Thread class, you have to override the run() method of the Thread class.
- After the object is created, start() function needs to be called in order to start the execution of the thread.

Program code:

```
class MultithreadingDemo extends Thread{
  public void run(){
    System.out.println("My thread is in running state.");
  }
  public static void main(String args[]){
     MultithreadingDemo obj=new MultithreadingDemo();
    obj.start();
  }
}
```

Implementing the Runnable Interface

- The easiest way to create a thread is to create a class that implements the Runnable interface.
- Runnable abstracts a unit of executable code. You can construct a thread on any object that implements Runnable. To implement Runnable, a class need only implement a single method called run( ), which is declared like this: public void run( )
- Inside run( ), you will define the code that constitutes the new thread. This thread will end when run( ) returns.

- After you create a class that implements Runnable, you will instantiate an object of that class and call the start() function to start the thread's run() function

Program Code:

```
class MultithreadingDemo implements Runnable{
  public void run(){
    System.out.println("My thread is in running state.");
  }
  public static void main(String args[]){
     MultithreadingDemo obj=new MultithreadingDemo();
     Thread tobj =new Thread(obj);
     tobj.start();
  }
}
```

13. What Synchronization? How is thread created? Explain thread lifecycle?

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.
- Key to synchronization is the concept of the monitor (also called a semaphore). A monitor is an object that is used as a mutually exclusive lock, or mutex.
- Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor.
- A thread that owns a monitor can reenter the same monitor if it so desires.

You can synchronize your code in either of two ways. Both involve the use of the synchronized keyword, and both are examined here.

- **Using Synchronized Methods**
- **The synchronized Block Statement**

Using Synchronized Methods

- Synchronization is easy in Java, because all objects have their own implicit monitor associated with them.
- To enter an object's monitor, just call a method that has been modified with the synchronized keyword.
- While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait.
- To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

Example of Synchronized method:

```
public class MyTest {
```

```
 static int i = 0;
 public static void main(String[] args) {
    new Thread(t1).start();
    new Thread(t2).start();
 }

private synchronized static void countMe(){
    i++;
    System.out.println("Current Counter is: " + i);
}

 private static Runnable t1 = new Runnable() {
    public void run() {
       try{
          for(int i=0; i<5; i++){
             countMe("t1");
          }
       } catch (Exception e){}

    }
 };

 private static Runnable t2 = new Runnable() {
    public void run() {
       try{
          for(int i=0; i<5; i++){
             countMe("t2");
          }
       } catch (Exception e){}
    }
 };
}
```

The synchronized Block Statement

- While creating synchronized methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases.
- To understand why, consider the following. Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use synchronized methods.
- Further, this class was not created by you, but by a third party, and you do not have access to the source code.
- Thus, you can't add synchronized to the appropriate methods within the class. How can access to an object of this class be synchronized?

- Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a synchronized block.

This is the general form of the synchronized statement:

```
synchronized(object) {
// statements to be synchronized
}
```

Here, object is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's monitor.

Program Code:

```
class Table{

 void printTable(int n){
   synchronized(this){//synchronized block
     for(int i=1;i<=5;i++){
      System.out.println(n*i);
      try{
       Thread.sleep(400);
      }catch(Exception e){System.out.println(e);}
     }
   }
 }//end of the method
}

class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}
```

```
public class TestSynchronizedBlock1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

Thread Lifecycle:

A thread can be in one of the five states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

- **New**
- **Runnable**
- **Running**
- **Non-Runnable (Blocked)**
- **Terminated**

1) New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running

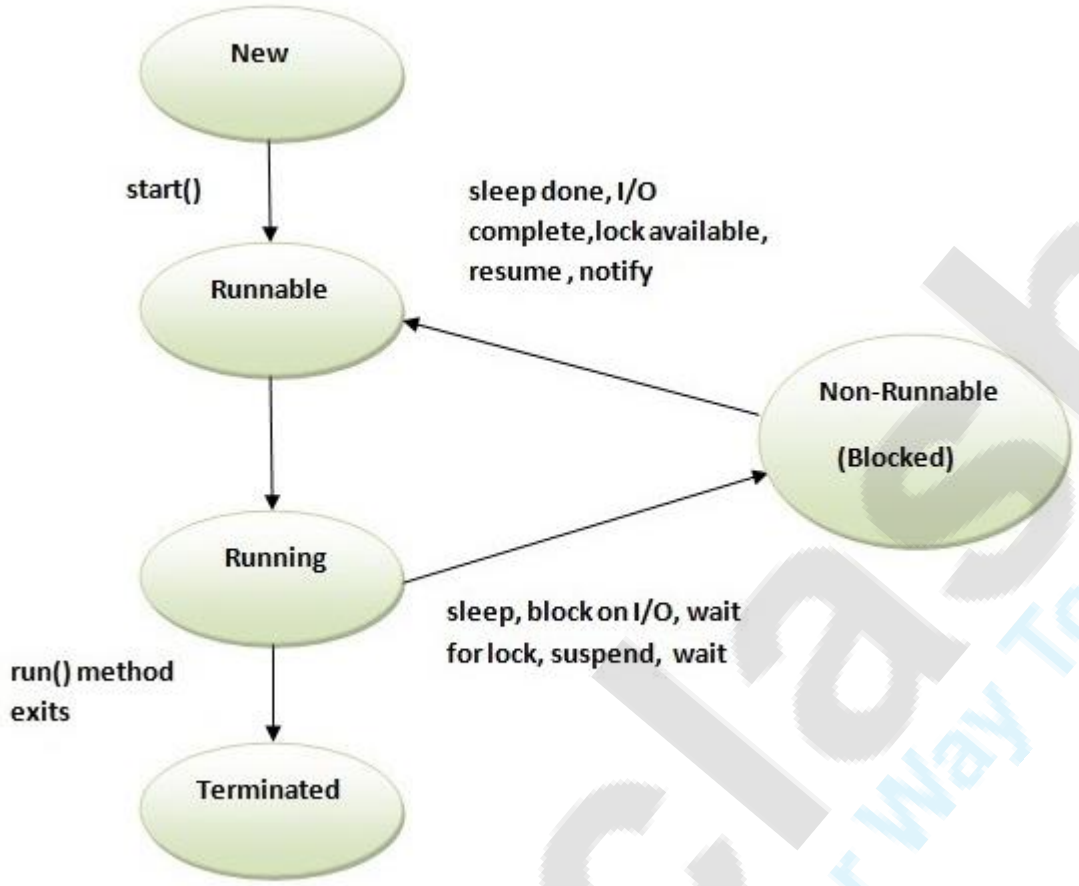
The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

A thread is in terminated or dead state when its run() method exits.



14. I/O in Java? Draw hierarchy of inputstream and outputstream classes?

- Java performs I/O through Streams. In general, a stream means continuous flow of data. A stream can be defined as a sequence of data.
- Java encapsulates Stream under java.io package. The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, Object, localized characters, etc.

Java defines two types of streams. They are:

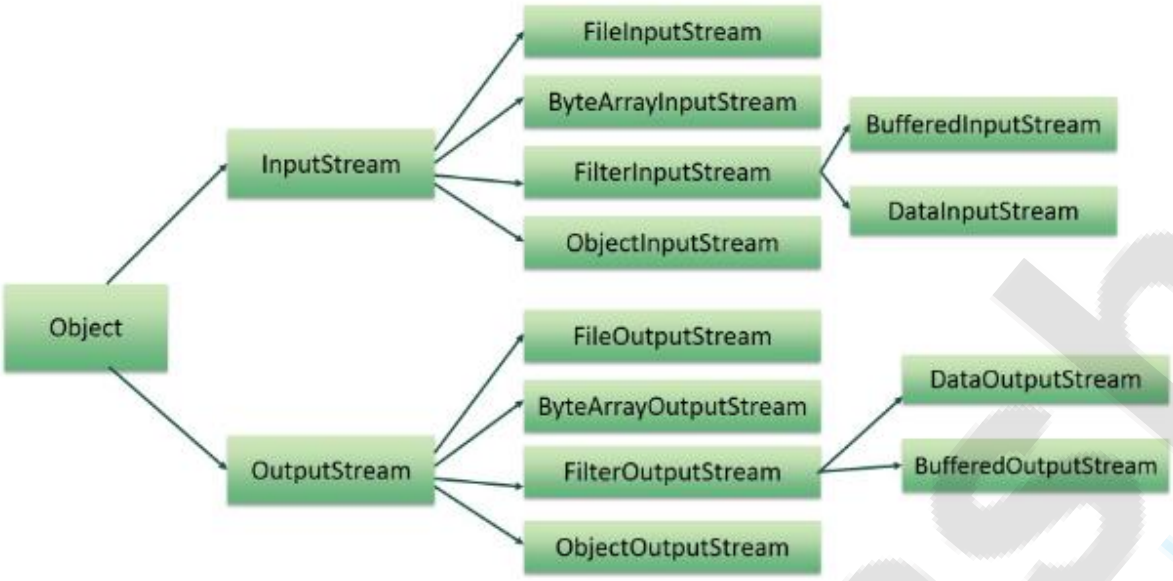
Byte Stream :

- It provides a convenient means for handling input and output of byte.

Character Stream :

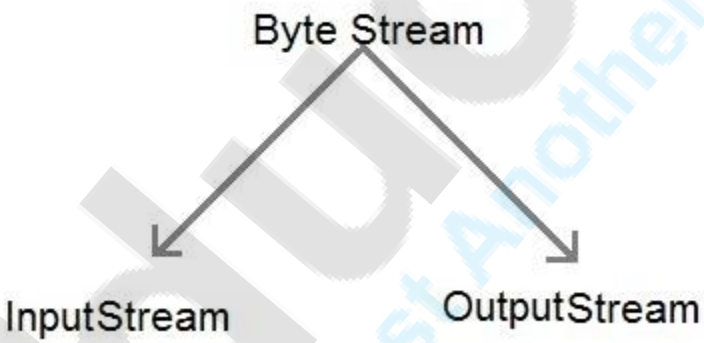
- It provides a convenient means for handling input and output of characters. Character stream uses Unicode and therefore can be internationalized.

Stream hierarchy diagram:



Byte Stream Classes

- Byte stream is defined by using two abstract class at the top of hierarchy, they are InputStream and OutputStream.
- These two abstract classes have several concrete classes that handle various devices such as disk files, network connection etc.



These classes define several key methods. Two most important are

- read() : reads byte of data.
- write() : Writes byte of data.

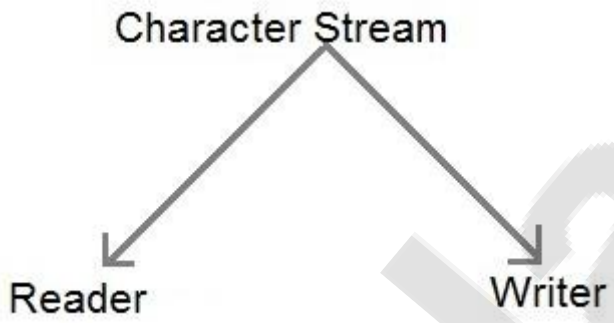
Some important Byte stream classes.

Stream class	Description
BufferedInputStream	Used for Buffered Input Stream.
BufferedOutputStream	Used for Buffered Output Stream.
DataInputStream	Contains method for reading java standard datatype
DataOutputStream	An output stream that contain method for writing java standard data type
FileInputStream	Input stream that reads from a file

FileOutputStream	Output stream that write to a file.
InputStream	Abstract class that describe stream input.
OutputStream	Abstract class that describe stream output.
PrintStream	Output Stream that contain `print()` and `println()` method

Character Stream Classes

- Character stream is also defined by using two abstract class at the top of hierarchy, they are **Reader** and.
- These two abstract classes have several concrete classes that handle **unicode** character.



Some important Character stream classes.

Stream class	Description
BufferedReader	Handles buffered input stream.
BufferedWriter	Handles buffered output stream.
FileReader	Input stream that reads from file.
FileWriter	Output stream that writes to file.
InputStreamReader	Input stream that translate byte to character
OutputStreamReader	Output stream that translate character to byte.
PrintWriter	Output Stream that contain `print()` and `println()` method.
Reader	Abstract class that define character stream input
Writer	Abstract class that define character stream output

Reading Console Input Writer

We use the object of BufferedReader class to take inputs from the keyboard.

Reading Characters

read() method is used with BufferedReader object to read characters. As this function returns integer type value has we need to use typecasting to convert it into char type.
Below is a simple example explaining character input.

```
class CharRead
{
```

```
 public static void main( String args[])
 {
  BufferedReader br = new Bufferedreader(new InputstreamReader(System.in));
  char c = (char)br.read();       //Reading character
 }
}
```

Reading Strings

To read string we have to use readLine() function with BufferedReader class's object.
Program to take String input from Keyboard in Java

```
import java.io.*;
class MyInput
{
 public static void main(String[] args)
 {
  String text;
  InputStreamReader isr = new InputStreamReader(System.in);
  BufferedReader br = new BufferedReader(isr);
  text = br.readLine();         //Reading String
  System.out.println(text);
 }
}
```

15. Explain Event Delegation method?

The modern approach to handling events is based on the **delegation event model**, which defines standard and consistent mechanisms to generate and process events.
Its concept is quite simple:

- A source generates an event and sends it to one or more listeners.
- In this scheme, the listener simply waits until it receives an event.
- Once an event is received, the listener processes the event and then returns.

The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events.
A user interface element is able to “delegate” the processing of an event to a separate piece of code.
In the delegation event model, **listeners must register with a source** in order to receive an event notification.
This provides an important benefit: **notifications are sent only to listeners that want to receive them.** This is a more efficient way to handle events than the design used by the old Java 1.0

approach. Previously, an event was propagated up the containment hierarchy until it was handled by a component. This required components to receive events that they did not process, and it wasted valuable time. The delegation event model eliminates this overhead.

Events

In the delegation model, **an event is an object that describes a state change in a source.** It can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are **pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse**. Many other user operations could also be cited as examples. Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a **timer expires, a counter exceeds a value, a software or hardware failure occurs**, or an operation is completed. You are free to define events that are appropriate for your application.

Event Sources

A source is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event. **A source must register listeners in order for the listeners** to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form:

public void addTypeListener(TypeListener el)

Here, **Type** is the name of the event, and el is a reference to the event listener. For example, the method that registers a keyboard event listener is called addKeyListener( ). The method that registers a mouse motion listener is called addMouseMotionListener( ). When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as multicasting the event. In all cases, notifications are sent only to listeners that register to receive them.

Some sources may allow only one listener to register. The general form of such a method is this:

public void addTypeListener(TypeListener el) throws java.util.TooManyListenersException

Here, **Type** is the name of the event, and el is a reference to the event listener. When such an event occurs, the registered listener is notified. This is known as **unicasting** the event. A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

public void removeTypeListener(TypeListener el)

Here, Type is the name of the event, and el is a reference to the event listener. For example, to remove a keyboard listener, you would call removeKeyListener( ).

The methods that add or remove listeners are provided by the source that generates events. For example, the Component class provides methods to add and remove keyboard and mouse event listeners.

Event Listeners

A listener is an object that is notified when an event occurs. It has two major requirements.

First, it must have been registered with one or more sources to receive notifications about specific types of events.
Second, it must implement methods to receive and process these notifications.
The methods that receive and process events are defined in a set of interfaces found in java.awt.event. For example, the MouseMotionListener interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface.

As explained, the delegation event model has two parts: sources and listeners. Listeners are created by implementing one or more of the interfaces defined by the java.awt.event package. Here are a list of commonly used Event Listeners Interfaces

Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

TABLE 22-3 Commonly Used Event Listener Interfaces

16. Explain different types of Layout Managers in detail?

- A layout manager automatically arranges your controls within a window by using some type of algorithm.
- Each Container object has a layout manager associated with it. A layout manager is an instance of any class that implements the LayoutManager interface. The layout manager is set by the setLayout( ) method. If no call to setLayout( ) is made, then the default layout manager is used.

- Whenever a container is resized (or sized for the first time), the layout manager is used to position each of the components within it.

The setLayout( ) method has the following general form:
void setLayout(LayoutManager layoutObj)
Here, layoutObj is a reference to the desired layout manager.

- Each layout manager keeps track of a list of components that are stored by their names.
- The layout manager is notified each time you add a component to a container. Whenever the container needs to be resized, the layout manager is consulted via its minimumLayoutSize( ) and preferredLayoutSize( ) methods.
- Each component that is being managed by a layout manager contains the getPreferredSize( ) and getMinimumSize( ) methods. These return the preferred and minimum size required to display each component. The layout manager will honor these requests if at all possible, while maintaining the integrity of the layout policy.
- You may override these methods for controls that you subclass. Default values are provided otherwise.

Java has several predefined LayoutManager classes, several of which are described next. You can use the layout manager that best fits your application.
Following are the Layout Manager classes in Java:

- **FlowLayout**
- **BorderLayout**
- **GridLayout**
- **CardLayout**
- **GridBagLayout**

FlowLayout

- FlowLayout is the default layout manager.
- This is the layout manager that the preceding examples have used. FlowLayout implements a simple layout style, which is similar to how words flow in a text editor.
- The direction of the layout is governed by the container's component orientation property, which, by default, is left to right, top to bottom. Therefore, by default, components are laid out line-by-line beginning at the upper-left corner.
- In all cases, when a line is filled, layout advances to the next line.
- A small space is left between each component, above and below, as well as left and right.

Here are the constructors for
FlowLayout:
FlowLayout( )
FlowLayout(int how)
FlowLayout(int how, int horz, int vert)

- The first form creates the default layout, which centers components and leaves five pixels of space between each component.
- The second form lets you specify how each line is aligned.

Valid values for how are as follows:

FlowLayout.LEFT
FlowLayout.CENTER
FlowLayout.RIGHT
FlowLayout.LEADING
FlowLayout.TRAILING
These values specify left, center, right, leading edge, and trailing edge alignment, respectively.

- The third constructor allows you to specify the horizontal and vertical space left between components in horz and vert, respectively.

BorderLayout

- The BorderLayout class implements a common layout style for top-level windows.
- It has four narrow, fixed-width components at the edges and one large area in the center.
- The four sides are referred to as north,south, east, and west.
- The middle area is called the center.

Here are the constructors defined
by BorderLayout:
BorderLayout( )
BorderLayout(int horz, int vert)

- The first form creates a default border layout.
- The second allows you to specify the horizontal and vertical space left between components in horz and vert, respectively.

BorderLayout defines the following constants that specify the regions:
BorderLayout.CENTER
BorderLayout.EAST
BorderLayout.NORTH
BorderLayout.SOUTH
BorderLayout.WEST

When adding components, you will use these constants with the following form of add( ), which is defined by Container:
void add(Component compObj, Object region)
Here, compObj is the component to be added, and region specifies where the component will be added.

GridLayout

- GridLayout lays out components in a two-dimensional grid. When you instantiate a GridLayout, you define the number of rows and columns.

The constructors supported
by GridLayout are shown here:
GridLayout( )
GridLayout(int numRows, int numColumns)
GridLayout(int numRows, int numColumns, int horz, int vert)

- The first form creates a single-column grid layout.
- The second form creates a grid layout with the specified number of rows and columns.
- The third form allows you to specify the horizontal and vertical space left between components in horz and vert, respectively.

Either numRows or numColumns can be zero. Specifying numRows as zero allows for unlimited-length columns. Specifying numColumns as zero allows for unlimited-length rows

CardLayout

- The CardLayout class is unique among the other layout managers in that it stores several different layouts.
- Each layout can be thought of as being on a separate index card in a deck that can be shuffled so that any card is on top at a given time.
- This can be useful for user interfaces with optional components that can be dynamically enabled and disabled upon user input.
- You can prepare the other layouts and have them hidden, ready to be activated when needed.

CardLayout provides these two constructors:

CardLayout( )

CardLayout(int horz, int vert)

- The first form creates a default card layout.
- The second form allows you to specify the horizontal and vertical space left between components in horz and vert, respectively.
- Use of a card layout requires a bit more work than the other layouts. The cards are typically held in an object of type Panel. This panel must have CardLayout selected as its layout manager.
- The cards that form the deck are also typically objects of type Panel. Thus, you must create a panel that contains the deck and a panel for each card in the deck.
- Next, you add to the appropriate panel the components that form each card. You then add these panels to the panel for which CardLayout is the layout manager.
- Finally, you add this panel to the window. Once these steps are complete, you must provide some way for the user to select between cards.
- One common approach is to include one push button for each card in the deck. When card panels are added to a panel, they are usually given a name.
- Thus, most of the time, you will use this form of add( ) when adding cards to a panel:

void add(Component panelObj, Object name)

- Here, name is a string that specifies the name of the card whose panel is specified by panelObj.

After you have created a deck, your program activates a card by calling one of the following methods defined by CardLayout:

void first(Container deck)

void last(Container deck)

void next(Container deck)

void previous(Container deck)

void show(Container deck, String cardName)

- Here, deck is a reference to the container (usually a panel) that holds the cards, and cardName is the name of a card.
- Calling first( ) causes the first card in the deck to be shown.
- To show the last card, call last( ). To show the next card, call next( ).
- To show the previous card, call previous( ).
- Both next( ) and previous( ) automatically cycle back to the top or bottom of the deck, respectively.
- The show( ) method displays the card whose name is passed in cardName.

GridBagLayout

- In GridBagLayout you can specify the relative placement of components by specifying their positions within cells inside a grid.
- The key to the grid bag is that each component can be a different size, and each row in the grid can have a different number of columns.
- This is why the layout is called a grid bag. It's a collection of small grids joined together. The location and size of each component in a grid bag are determined by a set of constraints linked to it.
- The constraints are contained in an object of type GridBagConstraints. Constraints include the height and width of a cell, and the placement of a component, its alignment, and its anchor point within the cell.
- The general procedure for using a grid bag is to first create a new GridBagLayout object and to make it the current layout manager. Then, set the constraints that apply to each component that will be added to the grid bag.
- Finally, add the components to the layout manager. Although GridBagLayout is a bit more complicated than the other layout managers, it is still quite easy to use once you understand how it works. GridBagLayout defines only one constructor, which is shown here:

GridBagLayout( )

GridBagLayout defines several methods, of which many are protected and not for general use. There is one method, however, that you must use: setConstraints( ). It is shown here:

void setConstraints(Component comp, GridBagConstraints cons)

- Here, comp is the component for which the constraints specified by cons apply.
- This method sets the constraints that apply to each component in the grid bag.
- The key to successfully using GridBagLayout is the proper setting of the constraints, which are stored in a GridBagConstraints object. GridBagConstraints defines several fields that you can set to govern the size, placement, and spacing of a component.

17. What is database drivers? What are the types of database drivers?

- A JDBC driver is a software component enabling a Java application to interact with a database.
- JDBC drivers implement the defined interfaces in the JDBC API, for interacting with your database server.
- For example, using JDBC drivers enable you to open database connections and to interact with it by sending SQL or database commands then receiving results with Java.
- The Java.sql package that ships with JDK, contains various classes with their behaviors defined and their actual implementations are done in third-party drivers. Third party vendors implements the java.sql.Driver interface in their database driver.

JDBC Drivers Types

JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates.

There are **4 types** of JDBC driver types based on their mode of working:

Type 1: JDBC-ODBC Bridge Driver

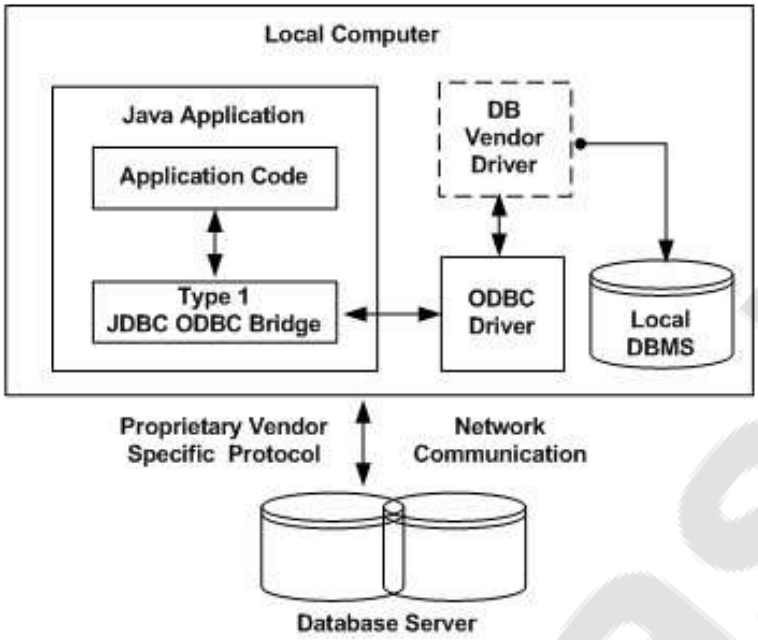
Type 2: JDBC-Native API

Type 3: JDBC-Net pure Java OR Network-Protocol driver (Middleware driver)

Type 4: 100% Pure Java OR Database-Protocol driver (Pure Java driver)

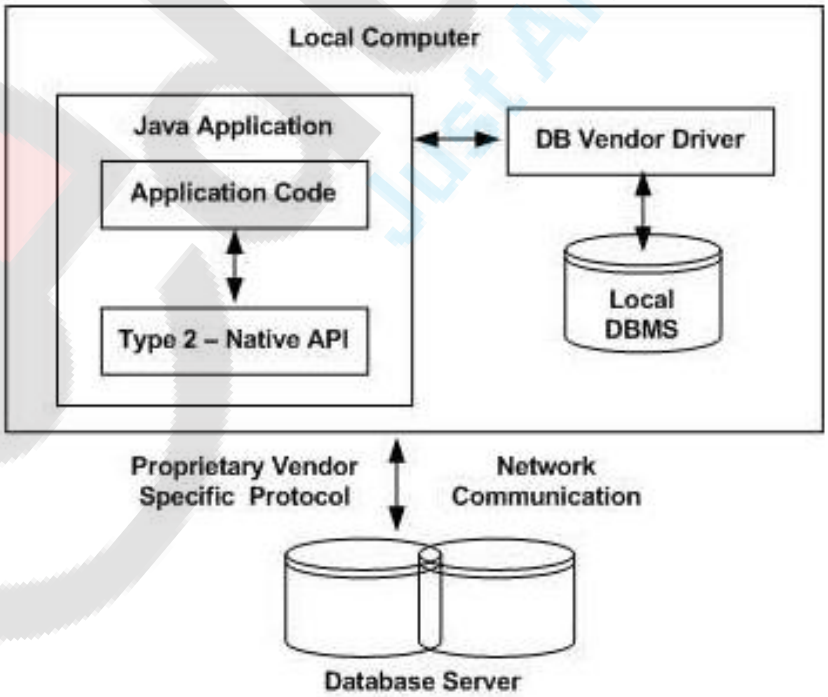
Type 1: JDBC-ODBC Bridge Driver

- A JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC, requires configuring on your system a Data Source Name (DSN) that represents the target database.
- The driver is platform-dependent as it makes use of ODBC which in turn depends on native libraries of the underlying operating system the JVM is running upon.
- Also, use of this driver leads to other installation dependencies; for example, ODBC must be installed on the computer having the driver and the database must support an ODBC driver.
- Sun provided a JDBC-ODBC Bridge driver: sun.jdbc.odbc.JdbcOdbcDriver. This driver is native code and not Java, and is closed source. Oracle's JDBC-ODBC Bridge was removed in Java 8



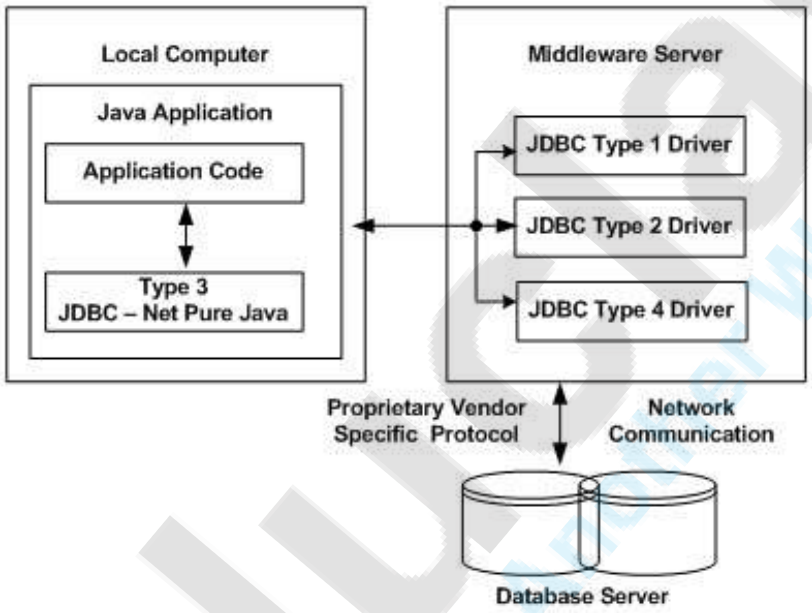
Type 2: JDBC-Native API

- In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls, which are unique to the database. These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge. The vendor-specific driver must be installed on each client machine.
- If we change the Database, we have to change the native API, as it is specific to a database and they are mostly obsolete now, but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead.
- The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.



Type 3: JDBC-Net pure Java OR Network-Protocol driver (Middleware driver)

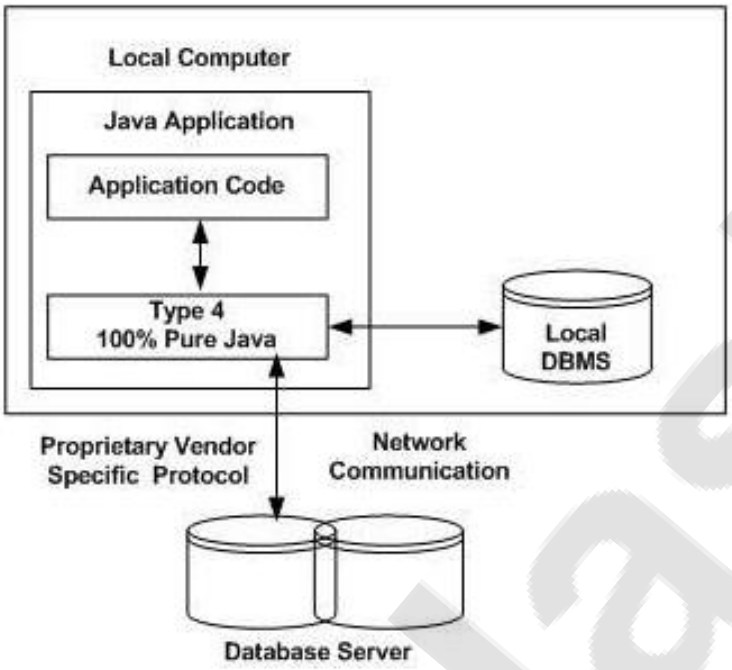
- In a Type 3 driver, a three-tier approach is used to access databases. The JDBC clients use standard network sockets to communicate with a middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server.
- This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.
- You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver type.



Type 4: 100% Pure Java OR Database-Protocol driver (Pure Java driver)

- In a Type 4 driver, a pure Java-based driver communicates directly with the vendor's database through socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself.
- This kind of driver is extremely flexible, you don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.

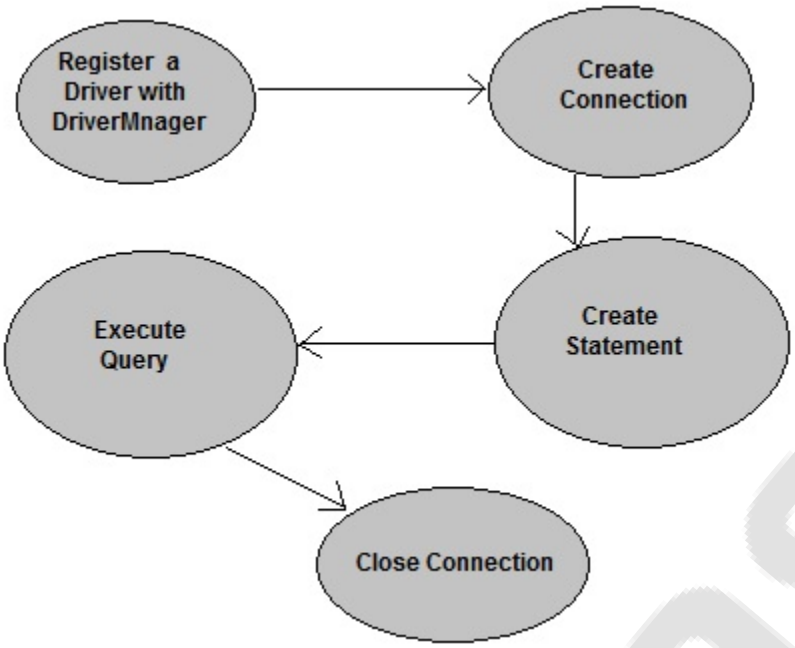
- MySQL's Connector/J driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.



18. State and explain the steps involved in creating a Database connection in Java?

There are 5 steps to connect any java application with the database in java using JDBC. They are as follows:

1. **Register the driver class**
2. **Creating connection**
3. **Creating statement**
4. **Executing queries**
5. **Closing connection**



1) Register the driver class

The forName() method of Class class is used to register the driver class. This method is used to dynamically load the driver class.

Syntax of forName() method

public static void forName(String className)throws ClassNotFoundException

Example to register the OracleDriver class

Class.forName("oracle.jdbc.driver.OracleDriver");

2) Create the connection object

The getConnection() method of DriverManager class is used to establish connection with the database.

Syntax of getConnection() method

public static Connection getConnection(String url)throws SQLException

public static Connection getConnection(String url,String name,String password) throws SQLException

Example to establish connection with the Oracle database

Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","password");

3) Create the Statement object

The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.

Syntax of createStatement() method

public Statement createStatement()throws SQLException

Example to create the statement object

Statement stmt=con.createStatement();

4) Execute the query

The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.

Syntax of executeQuery() method

public ResultSet executeQuery(String sql)throws SQLException

Example to execute query

```
ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next()){
System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```

5) Close the connection object

By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

Syntax of close() method

public void close()throws SQLException

Example to close connection

```
con.close();
```

19. What is ResultSet in Java? Explain its types?

- A ResultSet object **maintains a cursor pointing to its current row of data**.
- Initially the cursor is positioned before the first row. The next method moves the cursor to the next row, and because it returns false when there are no more rows in the ResultSet object, it can be used in a while loop to iterate through the result set.

- A default ResultSet object is not updatable and has a cursor that moves forward only. Thus, you can iterate through it only once and only from the first row to the last row. It is possible to produce ResultSet objects that are scrollable and/or updatable

The methods of the ResultSet interface can be broken down into three categories −

- **Navigational methods:** Used to move the cursor around.
- **Get methods:** Used to view the data in the columns of the current row being pointed by the cursor.
- **Update methods:** Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.

Following are the types of ResultSets depending upon their mode of operations:

ResultSet.TYPE_FORWARD_ONLY

The cursor can only move forward in the result set.

ResultSet.TYPE_SCROLL_INSENSITIVE

The cursor can scroll forward and backward, and the result set is not sensitive to changes made by others to the database that occur after the result set was created.

ResultSet.TYPE_SCROLL_SENSITIVE

The cursor can scroll forward and backward, and the result set is sensitive to changes made by others to the database that occur after the result set was created.

20. Difference between AWT and Swing Components?

AWT	SWING
Components are platform dependent	Platform independent
Components are heavy weight	Components are light weight
Doesn't support pluggable look and feel	Supports pluggable look and feel
Provides less components than swing	Provides more powerful components such as tables, lists, scroll panes
AWT doesn't follow MVC design pattern	SWING follows MVC design pattern

21. Write a short note on Servlets? Explain the servlet lifecycle with suitable diagram?

Servlets are programs that execute on the server side of a web connection. Just as applets dynamically extend the functionality of a web browser, servlets dynamically extend the functionality of a web server.

Prior to servlets, CGI (Common Gateway Interface) was used to communicate with webservers.

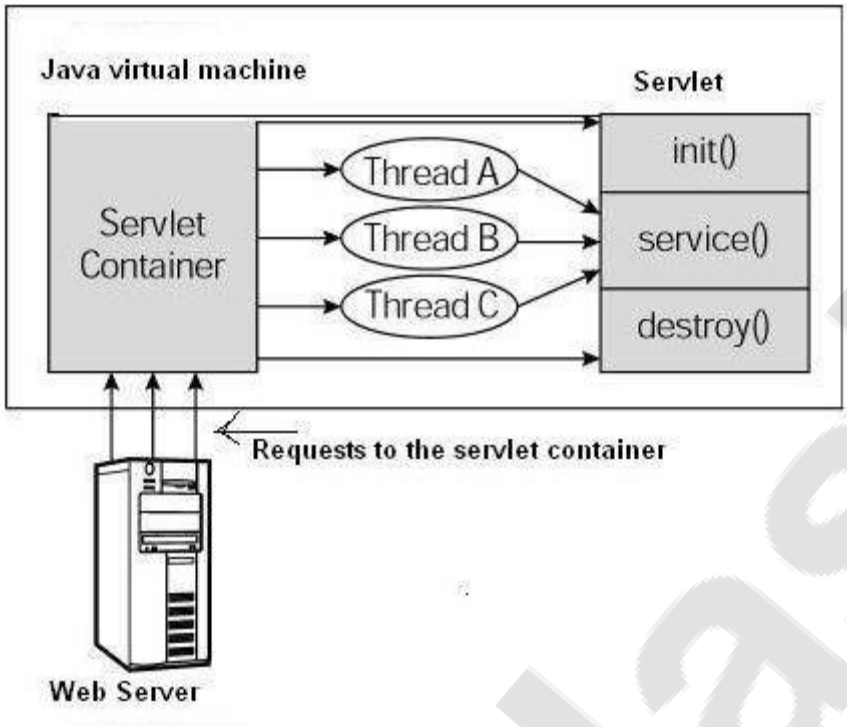
However, CGI suffered serious performance problems. It was expensive in terms of processor and memory resources to create a separate process for each client request. It was also expensive to open and close database connections for each client request.

Servlets offer several advantages in comparison with CGI.

- First, performance is significantly better. Servlets execute within the address space of a web server. It is not necessary to create a separate process to handle each client request.
- Second, servlets are platform-independent because they are written in Java.
- Third, the Java security manager on the server enforces a set of restrictions to protect the resources on a server machine.
- Finally, the full functionality of the Java class libraries is available to a servlet. It can communicate with applets, databases, or other software via the sockets and RMI mechanisms that you have seen already.

Two packages contain the classes and interfaces that are required to build servlets. These are **javax.servlet** and **javax.servlet.http.** They constitute the Servlet API.

Servlet Lifecycle



Three methods are central to the life cycle of a servlet. These are init( ), service( ), and destroy( ). They are implemented by every servlet and are invoked at specific times by the server.

The following are the paths followed by a servlet

- The servlet is initialized by calling the **init ()** method.
- The servlet calls **service()** method to process a client's request.
- The servlet is terminated by calling the **destroy()** method.
- Finally, servlet is garbage collected by the garbage collector of the JVM.

The init() method :

The init method is designed to be called only once. It is called when the servlet is first created, and not called again for each user request. So, it is used for one-time initializations, just as with the init method of applets.

The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started.

When a user invokes a servlet, a single instance of each servlet gets created, **with each user request resulting in a new thread that is handed off to doGet or doPost as appropriate.** The init() method simply creates or loads some data that will be used throughout the life of the servlet.

The init method definition looks like this:

```
public void init() throws ServletException {
  // Initialization code...
}
```

The service() method :

The service() method is the main method to perform the actual task. The servlet container (i.e. web server) calls the service() method to handle requests coming from the client(browsers) and to write the formatted response back to the client.

Each time the server receives a request for a servlet, the server spawns a new thread and calls service. The service() method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls doGet, doPost, doPut, doDelete, etc. methods as appropriate.

Here is the signature of this method:

```
public void service(ServletRequest request, ServletResponse response) throws
ServletException, IOException
{
}
```

The service () method is called by the container and service method invokes doGe, doPost, doPut, doDelete, etc. methods as appropriate. So you have nothing to do with service() method but you override either doGet() or doPost() depending on what type of request you receive from the client.

The doGet() and doPost() are most frequently used methods with in each service request. Here is the signature of these two methods.

The doGet() Method

A GET request results from a normal request for a URL or from an HTML form that has no METHOD specified and it should be handled by doGet() method.

```
public void doGet(HttpServletRequest request,  HttpServletResponse response)  throws
ServletException, IOException {
   // Servlet code}
```

The doPost() Method

A POST request results from an HTML form that specifically lists POST as the METHOD and it should be handled by doPost() method.

```
public void doPost(HttpServletRequest request,
                HttpServletResponse response)
   throws ServletException, IOException {
   // Servlet code
}
```

The destroy() method :

The destroy() method is called only once at the end of the life cycle of a servlet. This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities.

After the destroy() method is called, the servlet object is marked for garbage collection. The destroy method definition looks like this:

```
 public void destroy() {
   // Finalization code...
```

}

22. What is cookie class? What are methods of Cookie class? Explain with example?

- Cookies are text files stored on the client computer and they are kept for various information tracking purpose.
- A cookie is a small piece of information that is persisted between the multiple client requests. A cookie has a name, a single value, and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number.
- This session tracking technique is implemented on the client side, i.e. if the browser does not support cookies on the user machine this technique does not work.
- Java Servlets transparently supports HTTP cookies.

Setting Cookies with Servlet:

Setting cookies with servlet involves three steps:

(1) Creating a Cookie object: You call the Cookie constructor with a cookie name and a cookie value, both of which are strings.

Cookie cookie = new Cookie("key","value");

Keep in mind, neither the name nor the value should contain white space or any of the following characters:[ ] ( ) = , " / ? @ : ;

(2) Setting the maximum age: You use setMaxAge to specify how long (in seconds) the cookie should be valid. Following would set up a cookie for 24 hours.

cookie.setMaxAge(60*60*24);

(3) Sending the Cookie into the HTTP response headers: You use response.addCookie to add cookies in the HTTP response header as follows:

response.addCookie(cookie);

Servlet Cookies Methods:

Following is the list of useful methods which you can use while manipulating cookies in servlet:

- ***public void setDomain(String pattern)***
  - This method sets the domain to which cookie applies, for example tutorialspoint.com.
- ***public String getDomain()***
  - This method gets the domain to which cookie applies, for example tutorialspoint.com.
- ***public void setMaxAge(int expiry)***
  - This method sets how much time (in seconds) should elapse before the cookie expires. If you don't set this, the cookie will last only for the current session.
- ***public int getMaxAge()***
  - This method returns the maximum age of the cookie, specified in seconds, By default, -1 indicating the cookie will persist until browser shutdown.
- ***public String getName()***

    - This method returns the name of the cookie. The name cannot be changed after creation.
- ***public void setValue(String newValue)***
    - This method sets the value associated with the cookie.
- ***public String getValue()***
    - This method gets the value associated with the cookie.
- ***public void setPath(String uri)***
    - This method sets the path to which this cookie applies. If you don't specify a path, the cookie is returned for all URLs in the same directory as the current page as well as all subdirectories.
- ***public String getPath()***
    - This method gets the path to which this cookie applies.
- ***public void setSecure(boolean flag)***
    - This method sets the boolean value indicating whether the cookie should only be sent over encrypted (i.e. SSL) connections.
- ***public void setComment(String purpose)***
    - This method specifies a comment that describes a cookie's purpose. The comment is useful if the browser presents the cookie to the user.
- ***public String getComment()***
    - This method returns the comment describing the purpose of this cookie, or null if the cookie has no comment.

23. Session Tracking in Servlets and Java?

- Session simply means a particular interval of time.
- Session Tracking is a way to **maintain state (data) of a user**. It is also known as **session management** in servlet.
- **Http protocol** is a **stateless** so we need to maintain state using session tracking techniques. Each time user requests to the server, server treats the request as the new request. So we need to maintain the state of an user to recognize to particular user.
- Session tracking process is primarily used to recognize a particular user and his behavior, events, activities, demographics etc.

Session Tracking Techniques

There are four techniques used in Session tracking:

- **Cookies**
- **Hidden Form Field**
- **URL Rewriting**
- **HttpSession**

- **Cookies:**
    - You can use HTTP cookies to store information. Cookies will be stored at browser side.
- **URL rewriting:**

- With this method, the information is carried through url as request parameters. In general added parameter will be sessionid, userid.

- **HttpSession:**
  - Using HttpSession, we can store information at server side.
  - Http Session provides methods to handle session related information.
- **Hidden form fields:**
  - By using hidden form fields we can insert information in the webpages and these information will be sent to the server. These fields are not visible directly to the user, but can be viewed using view source option from the browsers.
  - The hidden form fields are as given below: <input type='hidden' name='siteName' value='mcastud'/>

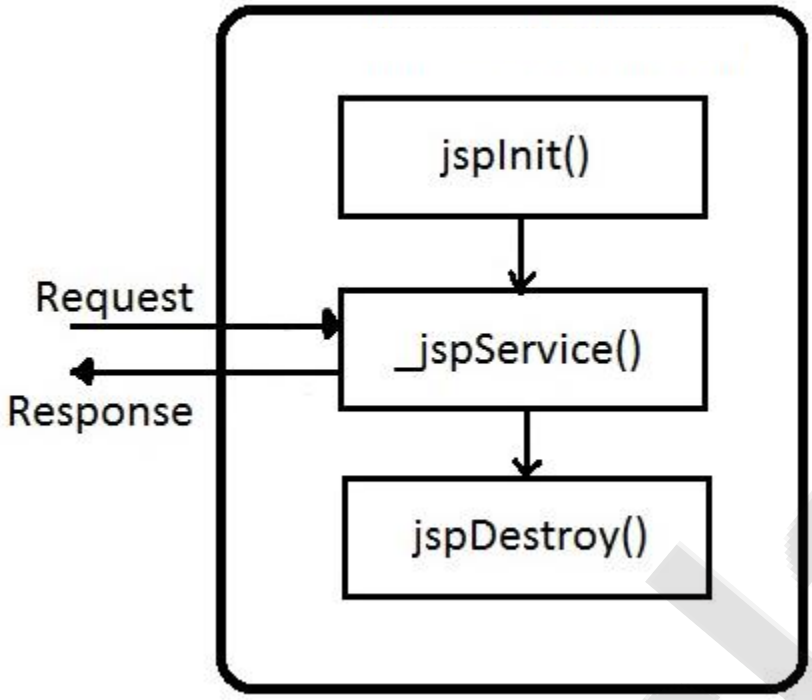
24. Difference between Session and Cookie?

Session	Cookie
Data is stored on the server side	Data stored on client side
Can store any type of data	Can store only textual data
Age of data is not fixed	WE can set the age limit of cookies
Session data is destroyed after session time out or logout	Data remains on client machine
Less data travels through the network	A cookie needs to travel each time client sends a request
More secure mechanism for session tracking	Less secure compared to session

25. Write a short note on JSP and its life cycle phases? Also compare JSP with servlets?

- JavaServer Pages (JSP) is a technology that helps software developers create dynamically generated web pages based on HTML, XML, or other document types.
- It is a technology which helps developers insert java code in HTML pages by making use of special JSP tags, most of which start with <% and end with %>.
- It can be thought of as an extension to servlet because it provides more functionality than servlet such as expression language, jstl etc.
- A JSP page consists of HTML tags and JSP tags. The jsp pages are easier to maintain than servlet because we can separate designing and development. It provides some additional features such as Expression Language, Custom Tag etc.
- JSP tags can be used for a variety of purposes, such as retrieving information from a database or registering user preferences, accessing JavaBeans components, passing control between pages and sharing information between requests, pages etc.
- To deploy and run JavaServer Pages, a compatible web server with a servlet container, such as Apache Tomcat or Jetty, is required.

JSP Life cycle phases:



- JSP pages are saved with .jsp extension which lets the server know that this is a JSP page and needs to go through JSP life cycle stages.
- When client makes a request to Server, it first goes to container. Then container checks whether the servlet class is older than jsp page(To ensure that the JSP file got modified). If this is the case then container does the translation again (converts JSP to Servlet) otherwise it skips the translation phase (i.e. if JSP webpage is not modified then it doesn't do the translation to improve the performance as this phase takes time and to repeat this step every time is not time feasible)

The steps in the life cycle of jsp page are:

1.Translation
As stated above whenever container receives request from client, it does translation only when servlet class is older than JSP page otherwsie it skips this phase (reason I explained above).

2.Compilation
Servlet page is compiled by the compiler and gets converted into the class file.

3.Loading
Loads the corresponding servlet class

4.Instantiation
Instantiates the servlet class

5.Initialization

When a container loads a JSP it invokes the jspInit() method before servicing any requests. If you need to perform JSP-specific initialization, override the jspInit()

Typically initialization is performed only once and as with the servlet init method, you generally initialize database connections, open files, and create lookup tables in the jspInit method.

6.RequestProcessing

This phase of the JSP life cycle represents all interactions with requests until the JSP is destroyed.

A new thread is then gets created, which invokes the_jspService() method.

Whenever a browser requests a JSP and the page has been loaded and initialized, the JSP engine invokes the _jspService() method in the JSP.

The _jspService() method takes an HttpServletRequest and an HttpServletResponse as its parameters as follows:

```
void _jspService(HttpServletRequest request,
                 HttpServletResponse response)
{
  // Service handling code...
}
```

The _jspService() method of a JSP is invoked once per a request and is responsible for generating the response for that request and this method is also responsible for generating responses to all seven of the HTTP methods ie. GET, POST, DELETE etc.

7.Destruction

Invokes the jspDestroy() method to destroy the instance of the servlet class.

Advantage of JSP over Servlet

There are many advantages of JSP over servlet. They are as follows:

1) Extension to Servlet

JSP technology is the extension to servlet technology. We can use all the features of servlet in JSP. In addition to, we can use implicit objects, predefined tags, expression language and Custom tags in JSP, that makes JSP development easy.

2) Easy to maintain

JSP can be easily managed because we can easily separate our business logic with presentation logic. In servlet technology, we mix our business logic with the presentation logic.

3) Fast Development: No need to recompile and redeploy

If JSP page is modified, we don't need to recompile and redeploy the project. The servlet code needs to be updated and recompiled if we have to change the look and feel of the application.

4) Less code than Servlet

In JSP, we can use a lot of tags such as action tags, jstl, custom tags etc. that reduces the code. Moreover, we can use EL, implicit objects etc.

26. Difference between JSP and Servlet?

JSP	Servlet
JSP is a webpage scripting language that can generate dynamic content	Servlet are java programs that are already compiled which also create dynamic web content
JSP runs slower compared to servlet as it takes compilation time to convert into Java servlet	Servlet run faster compared to JSP
In MVC JSP act as a View	In MVC servlet acts as a Controller
Preferred when there is not much processing of data required	Used when more processing and manipulation involved
We can use custom tags which can directly call Java beans	No such custom tag facility in servlet

27. Difference between Applet and Application?

Applet	Application
Small program	Large program
Used to run a program on client Browser	Can be executed on stand alone computer system
Can be executed by any Java supported browser	Needs JDK, JRE, JVM installed on client machine
Applet applications are executed in restricted environment	Can access all the resources of computer
Applet has 5 methods which will be automatically invoked on occurance of specific events Init() Start() Stop() Destroy() Paint()	Has a single start point which is main method

28. Write a short note on EJB, EJB architecture and types of Enterprise Java beans?

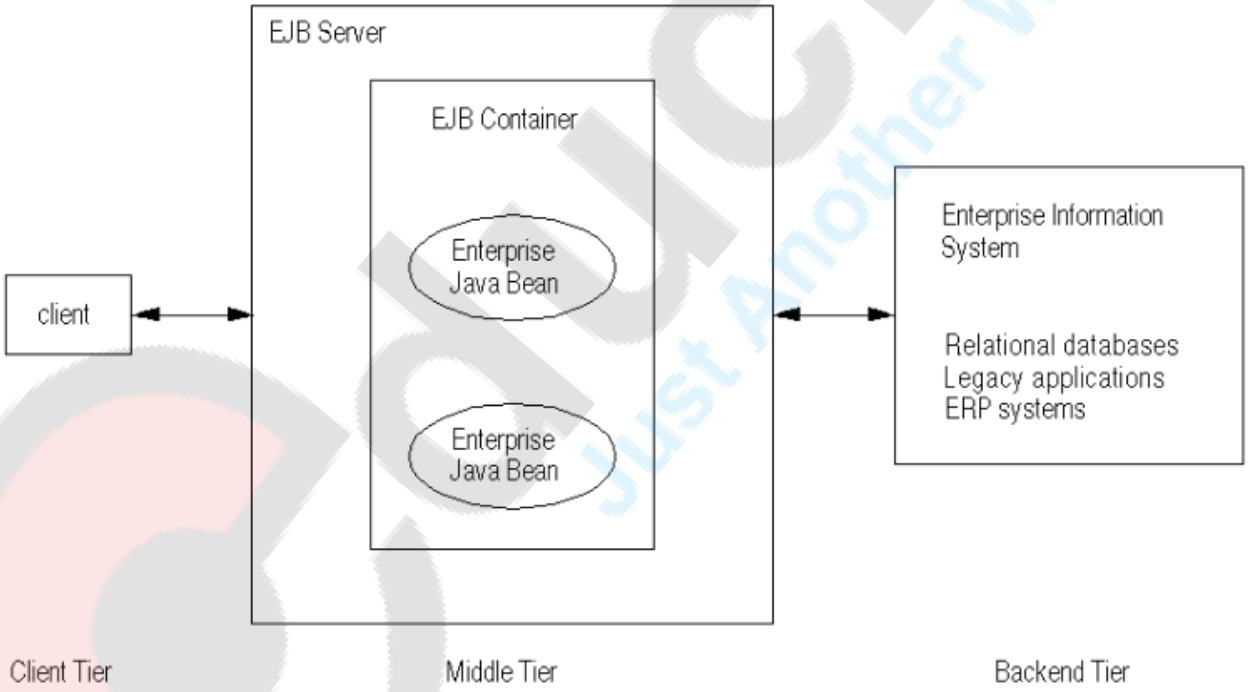
- EJB is an acronym for enterprise java bean. It is a specification provided by Sun Microsystems to develop secured, robust and scalable distributed applications.
- To run EJB application, you need an application server (EJB Container) such as Jboss, Glassfish, Weblogic, Websphere etc.EJB application is deployed on the server, so it is called server side component also.

EJB architecture is primarily used when:

- **Application needs Remote Access. In other words, it is distributed.**
- **Application needs to be scalable. EJB applications supports load balancing, clustering and fail-over.**
- **Application needs encapsulated business logic. EJB application is separated from presentation and persistent layer.**

EJB architecture consists of:

- An Enterprise Bean Server
- Enterprise Bean Container that run on these servers
- Enterprise Beans that run in these containers
- Enterprise Bean Clients
- Other Systems such as Java Naming and Directory Interface [JNDI] and Java transaction Service[JTS]



Enterprise Bean Server

An Enterprise JavaBean server is a Component Transaction Server. It supports the EJB server side component model for developing and deploying distributed enterprise-level applications in multi-tiered environment

An Enterprise JavaBean server provides:

- The framework for creating, deploying and managing middle-tier business logic.

- An environment that allows the execution of applications developed using Enterprise java beans components

And EJB server takes care of:

- Managing and coordinating the allocation of resources to the applications
- security
- Threads
- Connection pooling
- Access to distributed Transaction management services

Enterprise Bean Container

An EJB container manages the enterprise beans contained within it.It is basically a software implementing the EJB specifications. For each enterprise bean the container is responsible for:

- Registering the object
- Providing a remote interface for the object
- Creating and destroying object instances
- Checking security for the object
- Managing the active state of the object
- Coordinating distributed transactions

Enterprise Beans

Business components developed using EJB architecture are called as Enterprise JavaBeans components or simply Enterprise Beans

Enterprise beans are reusable modules of code that combine related tasks into a well-defined interface. These enterprise beans EJB components contain the methods that execute the business logic and access data sources.

Enterprise Bean Clients

Enterprise Bean client is a stand-alone application that provides the User Interface logic on a client machine.

The EJB client makes calls to remote EJB component on a server. The EJB client needs to be informed about:

- How to find the EJB server
- How to interact with the EJB components

Types of EJB

There are 3 types of enterprise bean in java.

- **Session Bean**
  - Session bean contains business logic that can be invoked by local, remote or webservice client.
- **Message Driven Bean**
  - Like Session Bean, it contains the business logic but it is invoked by passing message.

- **Entity Bean**
    - It encapsulates the state that can be persisted in the database. It is deprecated. Now, it is replaced with JPA (Java Persistent API).

Session Bean

- Session bean encapsulates business logic only, it can be invoked by local, remote and webservice client.
- It can be used for calculations, database access etc.
- The life cycle of session bean is maintained by the application server (EJB Container).

Types of Session Bean

There are 3 types of session bean.

1) Stateless Session Bean: It doesn't maintain state of a client between multiple method calls.

2) Stateful Session Bean: It maintains state of a client across multiple requests.

3) Singleton Session Bean: One instance per application, it is shared between clients and supports concurrent access.

Message Driven Bean

- A message driven bean (MDB) is a bean that contains business logic. But, it is invoked by passing the message.
- MDB asynchronously receives the message and processes it.Message driven bean is a stateless bean and is used to do task asynchronously.
- A message-driven bean's instances retain no data or conversational state for a specific client.
- A single message-driven bean can process messages from multiple clients.

Message-driven beans have the following characteristics.

- They execute upon receipt of a single client message.
- They are invoked asynchronously.
- They are relatively short-lived.
- They do not represent directly shared data in the database, but they can access and update this data.
- They can be transaction-aware.
- They are stateless.

Entity Bean

- Entity bean represents the persistent data stored in the database. It is a server-side component.
- An entity bean is a remote object that manages persistent data, performs complex business logic, potentially uses several dependent Java objects, and can be uniquely identified by a primary key.
- Entity beans are normally coarse-grained persistent objects, in that they utilize persistent data stored within several fine-grained persistent Java objects.

29. What is object serialization ? What is the use of serialVersionUID?

- Serialization is the process of writing the state of an object to a byte stream. This is useful when you want to save the state of your program to a persistent storage area, such as a file.
- At a later time, you may restore these objects by using the process of deserialization.
- Serialization is also needed to implement Remote Method Invocation (RMI). RMI allows a Java object on one machine to invoke a method of a Java object on a different machine. An object may be supplied as an argument to that remote method.
- The sending machine serializes the object and transmits it. The receiving machine deserializes it.

Serializable

- Only an object that implements the **Serializable interface** can be saved and restored by the serialization facilities. The Serializable interface defines no members. It is simply used to indicate that a class may be serialized.
- If a class is serializable, all of its subclasses are also serializable. Variables that are declared as transient are not saved by the serialization facilities.
- Also, static variables are not saved.
- The String class and all the wrapper classes implements java.io.Serializable interface by default.

Advantage of Java Serialization

- It is mainly used to travel object's state on the network (known as marshaling).
- Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.

The **ObjectOutputStream** class contains many write methods for writing various data types, but one method in particular stands out:
public final void writeObject(Object x) throws IOException
The above method **serializes an Object** and sends it to the output stream.
Important Methods

Method	**Description**
public final void writeObject(Object obj) throws IOException {}	Writes the specified object to the ObjectOutputStream.
public void flush() throws IOException {}	Flushes the current output stream
public void close() throws IOException {}	Closes the current output stream.

Similarly, the **ObjectInputStream** class contains the following method for **deserializing** an object:
public final Object readObject() throws IOException, ClassNotFoundException

This method retrieves the next Object out of the stream and **deserializes** it. The return value is Object, so you will need to cast it to its appropriate data type.

Important Methods

Method	Description
public final Object readObject() throws IOException, ClassNotFoundException{}	Reads an object from the input stream.
public void close() throws IOException {}	Closes ObjectInputStream.

serialVersionUID

- The serialization runtime associates with each serializable class a version number, called a serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization.
- If the receiver has loaded a class for the object that has a different serialVersionUID than that of the corresponding sender's class, then deserialization will result in an InvalidClassException.
- A serializable class can declare its own serialVersionUID explicitly by declaring a field named "serialVersionUID" that must be static, final, and of type long.

30. What is Struts? Explain Struts architecture ? Explain Struts 2 in detail?

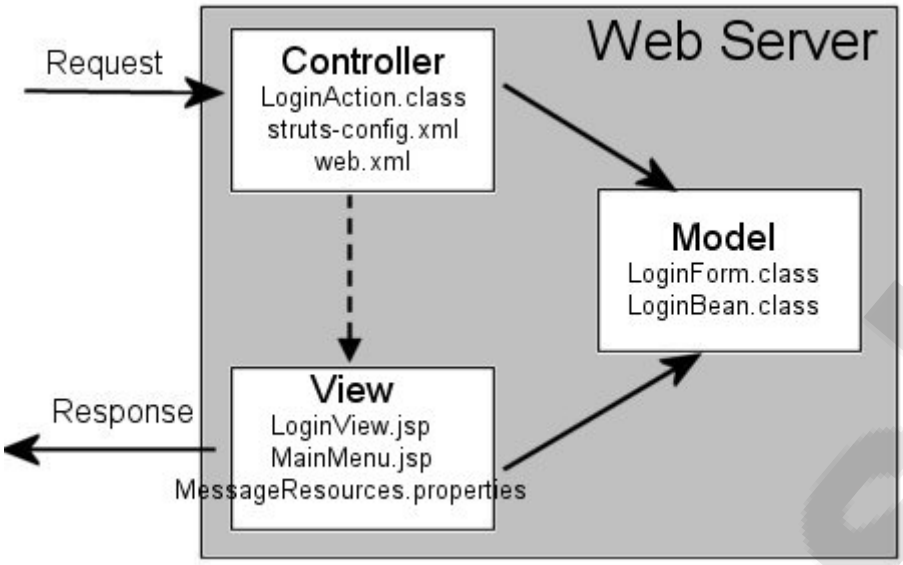
- Struts is a Java based framework that allows clean separation between the application logic that interacts with database from the HTML pages that perform the response.
- A framework is basically a collection of services that provide developers with common set of functionality, which can be reused and leveraged across multiple applications.
- It cuts time out of the development process and makes developers more productive by giving them prebuilt components to assemble Web- Applications from.
- Struts is not a technology, its a framework that can be used along with Java based technologies
- Struts makes the development of enterprise Web application development easier by providing a flexible and extensible application architecture, custom tags and a technique to configure common workflow within an application.
- The Struts framework is a strong implementation of the widely recognized Model View Controller design pattern. The key focus of MVC pattern is separation which is what is desired.

By using MVC design pattern, processing is broken into 3 distinct sections:

Model

View

Controller



Each component has its own unique responsibility and functionality which is independent of the others.

The advantages are:

- The business or the model specific logic is not written within the view component
- The presentation logic is not written within the model and business layers
- This allows resusability and providing flexiblity of changing one section without affecting the other.

Application flow in MVC

- In the MVC design pattern, the application flow is mediated by a Controller.
- The Controller delegates HTTP requests to appropriate handlerA Handler is nothing more than set of logic that is used to process the request. In Struts framework the handler is called Actions.
- The handlers are tied to a Model and each handler acts as an adapter or bridge, between the Request and the Model.A Handler or action may use one or more JavaBeans or EJBs to perform the actual business logic.
- The action gets any information out of the request necessary to perform the desired business logic and then passes it to the JavaBean EJBs

Technically:

- Using Web based data entry form information is submitted to the server.
- The controller receives such requests and to serve them calls the appropriate handler i.e. Action.
- Action processes the request by interacting with the application specific model code.
- Model returns a result to inform the Controller with output page to be sent as a response.

- Information is passed between Model and View in the form of special JavaBeans

- A central configuration file called struts.xml binds all these [Model, View and Controller] together.

Struts 2

Struts 2 is a brand new framework. It is completely new release of the older Struts 1 framework. Struts 2 is very simple compared to Struts 1.

It is the second generation Web application framework based on **OpenSymphony WebWork Framework** that implements the MVC design pattern in an efficient and simple way.

Struts 2:

- Uses JavaBeans instead of Action forms
- Is more powerful and can use Ajax and JSF
- Can easily integrate with other frameworks such as Webwork and Spring.
- Struts provides a cleaner implementation of MVC and introduces several new architectural features that makes the framework cleaner and more flexible.

Following are some features and advantages of using the struts 2 framework:

- **Action** based framework
- **Interceptors** for layering cross-cutting concerns away from action logic
- **Annotations** based configuration to reduce or eliminate XML configurations
- **Object Graph Navigation Language** support
- **Multiple View** options
- **Plugin Support** to modify framework features.
- Classes are based on Interfaces
- **Quick Start Feature** i.e. many changes can be made on the fly with restarting the Web Container
- Manual testing is saved as **built-in debugging tools** are provided for reporting problems
- **Theme based tag libraries** and **AJAX support**

31. Difference between Struts 1.x and Struts 2.x?

Struts 1.x	Struts 2.x
Front controller is ActionServlet	Front controller is FilterDispatcher
We have RequestProcessor class	We have Interceptors instead of RequestProcessor
Multiple tag libraries like html, logic, bean etc	Single library which includes all tags
Properties file must be configured in struts.xml	We need to configure our own resource bundle in struts.properties file
Only JSP as a view technology	Support of multiple View technologies like, velocity, Jasper Reports, JSP etc