Access Modifiers

Access modifiers (or access specifiers) are keywords in object-oriented languages that set the accessibility of classes, methods, and other members.

There are four types of access modifiers available in java:

Default (No keyword required): When we do not mention any access modifier, it is called default access modifier. The default modifier is accessible only within package.

Private: Private Data members and methods are only accessible within the class. Class and Interface cannot be declared as private. If a class has private constructor then you cannot create the object of that class from outside of the class.

Protected: Protected data member and method are only accessible by the classes of the same package and the subclasses present in any package.

Public: The members, methods and classes that are declared public can be accessed from anywhere. Public are also called universal access modifiers.

Packages in Java

Packages in Java are groups of similar types of classes, interface and sub packages.

Syntax:-

```
package;
```

Advantage of Java Package

- **Reusability:** The classes contained in the packages of another program can be easily reused.
- **Better Organization:** Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- **Protection:** Java package provides access protection.
- **Name Conflicts:** Java package removes naming collision.

Types of Packages

Built-in Package: Existing Java package for example java.lang, java.util etc.

User-defined-package: Java package created by user to categorize their project's classes and interface.

Steps to create a Java package:

1. Create a package name.
2. Pick up a base directory.
3. Make a subdirectory from the base directory that matches your package name.
4. Place your source files into the package subdirectory.
5. Use the package statement in each source file.
6. Compile your source files from the base directory.
7. Run your program from the base directory.

Example of Java packages

```
//save as FirstProgram.java

package learnjava;

public class FirstProgram{

   public static void main(String args[]) {

      System.out.println("Welcome to package");

   }

}
```

Output: Welcome to package

How to access package from another package?

import keyword

import keyword is used to import built-in and user-defined packages into your java source file so that your class can refer to a class that is in another package by directly using its name.

There are 3 different ways to refer to any class that is present in a different package:

1. **Using fully qualified name**

   If you use fully qualified name to import any class into your program, then only that particular class of the package will be accessible in your program, there is no need to use the import statement.

   Example :

   ```
   //save by A.java
   ```

```
package pack;
public class A {
   public void msg() {
      System.out.println("Hello");
   }
}
//save by B.java
package mypack;
class B {
   public static void main(String args[]) {
      pack.A obj = new pack.A();  //using fully qualified name
      obj.msg();
   }
}
```

Output: Hello

2. **Using packagename.classname**

To import only the class/classes you want to use.

Example :

```
//save by A.java
package pack;
public class A {
   public void msg() {
      System.out.println("Hello");
   }
```

```
}
//save by B.java
package mypack;
import pack.A;
class B {
   public static void main(String args[]) {
      A obj = new A();
      obj.msg();
   }
}
```

Output: Hello

3. **Using packagename.***

To import all the classes from a particular package but the classes and interface inside the subpackages will not be available for use.

Example :

```
//save by First.java
package learnjava;
public class First{
   public void msg() {
      System.out.println("Hello");
   }
}
//save by Second.java
package Java;
```

```
import learnjava.*;

class Second {

   public static void main(String args[]) {

      First obj = new First();

      obj.msg();

   }

}
```

Output: Hello

Subpackage

A package created inside another package is known as a subpackage. When we import a package, subpackages are not imported by default. They have to be imported explicitly. In the following statement :

import java.util.*;

util is a subpackage created inside java package.

Example of Subpackage

```
package com.javatpoint.core;

class Simple{

  public static void main(String args[]){

   System.out.println("Hello subpackage");

  }

}
```

Output: Hello subpackage

Abstract class in Java

- A class that is declared with abstract keyword, is known as abstract class in java.
- It can have abstract and non-abstract methods.

- It cannot be instantiated. Abstract classes can have Constructors, Member variables and Normal methods.
- When you extend Abstract class with abstract method, you must define the abstract method in the child class, or make the child class abstract.

Syntax :

abstract class *class_name* { }

abstract method

A method that is declared as abstract and does not have implementation is known as abstract method.

Syntax :

abstract return_type *function_name* (); // No definition

Example of Abstract class

```
abstract class A
{
 abstract void callme();
}
class B extends A
{
 void callme()
 {
  System.out.println("this is callme.");
 }
 public static void main(String[] args)
 {
  B b = new B();
```

```
 b.callme();
}
}
```

Output : this is callme.

Interface in Java

- Interface is a pure abstract class.
- They are syntactically similar to classes, but you cannot create instance of an Interface and their methods are declared without any body.

Properties of Interface

- An interface does not contain any constructors.
- The methods declared in interface are by default abstract (only method signature, no body)
- The variables declared in an interface are public, static & final by default.

Use of interface in Java

- Interface is used to achieve complete abstraction in Java.
- By using Interface, you can achieve multiple inheritance in java.
- It can be used to achieve loose coupling.

Syntax :

interface *interface_name* { }

Declaring Interfaces

The **interface** keyword is used to declare an interface.

Example of Interface

```
interface Person
{
 datatype variablename=value;
 //Any number of final, static fields
 returntype methodname(list of parameters or no parameters)
```

```
//Any number of abstract method declarations
}
```

Implementing Interfaces

A class uses the implements keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

Example:

```
interface Person
{
void run();  // abstract method
}
class A implements Person
{
public void run()
{
System.out.println("Run fast");
}
public static void main(String args[])
 {
 A obj = new A();
 obj.run();
 }
}
```

Output: Run fast

Interface extends other Interface

Classes implements interfaces, but an interface extends other interface.

Example

```
interface NewsPaper
{
 news();
}
interface Magazine extends NewsPaper
{
 colorful();
}
```

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.

Example:

```
interface Developer
{
void disp();
}
interface Manager
{
void show();
}
class Employee implements Developer, Manager
{
```

```
public void disp()
{
System.out.println("Hello Good Morning");
}
public void show()
{
System.out.println("How are you ?");
}
public static void main(String args[])
{
Employee obj=new Employee();
obj.disp();
obj.show();
}
}
```

Output: Hello Good Morning

How are you?

Final Keyword in Java

Final is a keyword in Java that is used to restrict the user and can be used in many respects. It is a non-access modifier. The final keyword can be used in following way:

1.) **Final at variable level:** When a variable is declared with final keyword, it's value can't be modified, essentially, a constant.

Example:

```
class FinalVariable
{
  public static void main(String[] args)
  {
```

```
    final int hours=24;
    System.out.println("Hours in 6 days = " + hours * 6);
  }
}
```

Output: Hours in 6 days = 144

2.) **Final at method level:** When a method is declared with final keyword, method cannot be overridden.

Example:

```
class X
{
 final void getMethod()
 {
  System.out.println("X method has been called");
 }
}
class Y extends X
{
 void getMethod() //cannot override
 {
  System.out.println("Y method has been called");
 }
}
class FinalMethod
{
 public static void main(String[] args)
 {
  Y obj = new Y();
  obj.getMethod();
 }
}
```

Output: error: "getMethod() in Y cannot override getMethod() in X; overridden method is final."

3.) **Final at class level:** When a class is declared with final keyword, class cannot be extended (inherited).

Example:

```
final class X
```

```
{
  //properties and methods of class X
}
class Y extends X
{
  //properties and methods of class Y
}
class FinalClass
{
   public static void main(String args[]) {}
}
```

Output: error: cannot inherit from final X

Super Keyword in Java

Super keyword in java is a reference variable that is used to refer parent class object.

The use of super keyword

1) To access the data members of parent class when both parent and child class have member with same name

2) To explicitly call the no-arg and parameterized constructor of parent class

3) To access the method of parent class when child class has overridden that method.

1. Use of super with variables: When you have a variable in child class which is already present in the parent class then in order to access the variable of parent class, you need to use the super keyword.

Example:

```
class Employee
{
```

```
float salary=10000;
}
class HR extends Employee
{
float salary=20000;
void display()
{
System.out.println("Salary: "+super.salary);//print base class salary
}
}
class Supervarible
{
public static void main(String[] args)
{
HR obj=new HR();
obj.display();
}
}
```

Output: Salary: 10000.0

2. Use of super with methods: The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

Example:

```
class Student
{
```

```
void message()
{
System.out.println("Good Morning Sir");
}
}
class Faculty extends Student
{
void message()
{
System.out.println("Good Morning Students");
}
void display()
{
message();//will invoke or call current class message() method
super.message();//will invoke or call parent class message() method
}
public static void main(String args[])
{
Student s=new Student();
s.display();
}
}
```

Output: Good Morning Students

Good Morning Sir

3. Use of super with constructors: The super keyword can also be used to invoke the parent class constructor.

Example:

```
class Employee
{
Employee()
{
System.out.println("Employee class Constructor");
}
}
class HR extends Employee
{
HR()
{
super(); //will invoke or call parent class constructor
System.out.println("HR class Constructor");
}
}
class Supercons
{
public static void main(String[] args)
{
HR obj=new HR();
}
```

```
}
```

Output: Employee class Constructor

HR class Constructor

transient keyword in Java

transient is a variables modifier used in serialization. At the time of serialization, if we don't want to save value of a particular variable in a file, then we use transient keyword.

Example of Java Transient Keyword

```
import java.io.Serializable;
public class Student implements Serializable{
 int id;
 String name;
 transient int age;//Now it will not be serialized
 public Student(int id, String name,int age) {
  this.id = id;
  this.name = name;
  this.age=age;
 }
}
```

Static keyword in java

The static keyword is used in java mainly for memory management. static is a non-access modifier in Java which is applicable for the blocks, variables, methods, nested classes.

1) **static variables:** When a variable is declared as static, then a single copy of variable is created and shared among all objects at class level. Static variables are, essentially, global variables. All instances of the class share the same static variable.

   Syntax for declare static variable:

public static *variableName*;

2) **static method:** When a method is declared with static keyword, it is known as static method. The most common example of a static method is main( ) method. Any static member can be accessed before any objects of its class are created, and without reference to any object. Methods declared as static have several restrictions:

- They can only directly call other static methods.
- They can only directly access static data.
- They cannot refer to *this* or *super* in any way.

Syntax for declare static method:

```
public static  void methodName()
{
.......
.......
}
```

3) **static block:** static block is used for initializing the static variables. This block gets executed when the class is loaded in the memory. A class can have multiple Static blocks, which will execute in the same sequence in which they have been written into the program.

Example of static block

```
class A{
  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

Volatile Keyword in Java

Volatile keyword in Java is used as an indicator to Java compiler and Thread that do not cache value of this variable and always read it from main memory.

Java volatile keyword cannot be used with method or class and it can only be used with variable.

If the variable keeps on changing such type of variables we have to declare with volatile modifier.

If a variable declared as volatile then for every thread a separate local copy will be created.

Example:

```
public class MyClass
 {
  private volatile int value;
  public int getValue() {
    return value;
  }
  public void setValue(int value)
{
    this.value = value;
  }
}
```

Synchronization in java is the capability to control the access of multiple threads to any shared resource. If a method or block declared as a Synchronized then at a time only one thread is allowed to operate on the given object. The synchronization is mainly used to

- To prevent thread interference.
- To prevent consistency problem.

The syntax for a synchronized method is as follows:

```
<method modifier> synchronized <method signature> {
    // synchronized code block
}
```

The syntax for a synchronized statement is as follows:

```
synchronized (expression) {
    // synchronized code block
}
```

Examples:

The following code example shows instance methods are synchronized:

```
class BankAccount {
   private double balance;
   synchronized void withdraw(double amount) {
      this.balance -= amount;
   }
   synchronized void deposit(double amount) {
      this.balance += amount;
   }
}
```

The following code example shows a synchronized statement is applied for a code block, not a method:

```
Object lock = new Object();
synchronized (lock) {
   System.out.println("Synchronized statement");
}
```

String Handling in Java

The basic aim of String Handling concept is storing the string data in the main memory (RAM), manipulating the data of the String, retrieving the part of the String etc. String Handling provides a lot of concepts that can be performed on a string such as concatenation of string, comparison of string, find sub string etc.

String: String is a sequence of characters enclosed within double quotes (" ") is known as String.

Example: "Java Programming".

How to create String object?

There are two ways to create String object:

1. **By string literal:** Java String literal is created by using double quotes.

For Example: String s="welcome";

2. **By new keyword:** In such case, JVM will create a new string object.

For Example: String s=new String("Welcome");

In java programming to store the character data we have a fundamental datatype called char. Similarly to store the string data and to perform various operations on String data, we have three predefined classes they are:

1. **String class:** It is a predefined class in java.lang package can be used to handle the String. String class is immutable that means whose content can not be changed at the time of execution of program.String class object is immutable that means when we create an object of String class it never changes in the existing object.

    Example:

```
class StringHandling
{
public static void main(String arg[])
{
String s=new String("java");
s.concat("software");
System.out.println(s);
}
}
```

    Output: java

Methods of String class

- **length():** This method is used to get the number of character of any string.
- **charAt():** This method is used to get the character at a given index value.
- **toUpperCase():** This method is use to convert lower case string into upper case.
- **toLowerCase():** This method is used to convert lower case string into upper case.
- **concat():** This method is used to combined two string.
- **equals():** This method is used to compare two strings, It return true if strings are same otherwise return false. It is case sensitive method.
- **equalsIgnoreCase():** This method is case insensitive method, It return true if the contents of both strings are same otherwise false.
- **compareTo():** This method is used to compare two strings by taking unicode values, It return 0 if the string are same otherwise return +ve or -ve integer values.
- **compareToIgnoreCase():** This method is case insensitive method, which is used to compare two strings similar to compareTo().
- **startsWith():** This method return true if string is start with given another string, otherwise it returns false.
- **endsWith():** This method return true if string is end with given another string, otherwise it returns false.
- **subString():** This method is used to get the part of given string.
- **indexOf():** This method is used find the index value of given string. It always gives starting index value of first occurrence of string.
- **lastIndexOf():** This method used to return the starting index value of last occurence of the given string.
- **trim():** This method remove space which are available before starting of string and after ending of string.
- **split():** This method is used to divide the given string into number of parts based on delimiter (special symbols like @ space ,).
- **replace():** This method is used to return a duplicate string by replacing old character with new character.

2. **StringBuffer:** It is a predefined class in java.lang package can be used to handle the String, whose object is mutable that means content can be modify. StringBuffer class is working with thread safe mechanism that means multiple thread are not allowed simultaneously to perform operation of StringBuffer. StringBuffer class object is mutable that means when we create an object of StringBuilder class it can be change.

   Example:

```
class StringHandling

{

public static void main(String arg[])

{

StringBuffer sb=new StringBuffer("java");

sb.append("software");

System.out.println(sb);

}

}
```

Output: javasoftware

Methods of StringBuffer class

- **reverse():** This method is used to reverse the given string and also the new value is replaced by the old string.
- **insert():** This method is used to insert either string or character or integer or real constant or boolean value at a specific index value of given string.
- **append():** This method is used to add the new string at the end of original string.
- **replace():** This method is used to replace any old string with new string based on index value.
- **deleteCharAt():** This method is used to delete a character at given index value.
- **delete():** This method is used to delete string form given string based on index value.
- **toString():** This method is used to convert mutable string value into immutable string.

3. **StringBuilder:** It is a predefined class in java.lang package can be used to handle the String. StringBuilder class is almost similar to to StringBuffer class. It is also a mutable object. The main difference StringBuffer and StringBuilder class is StringBuffer is thread safe that means only one threads allowed at a time to work on the String where as StringBuilder is not thread safe that means multiple threads can work on same String value.

Exceptions

An exception (or exceptional event) is a problem that arises during the execution of a program. When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally.

Error VS Exception

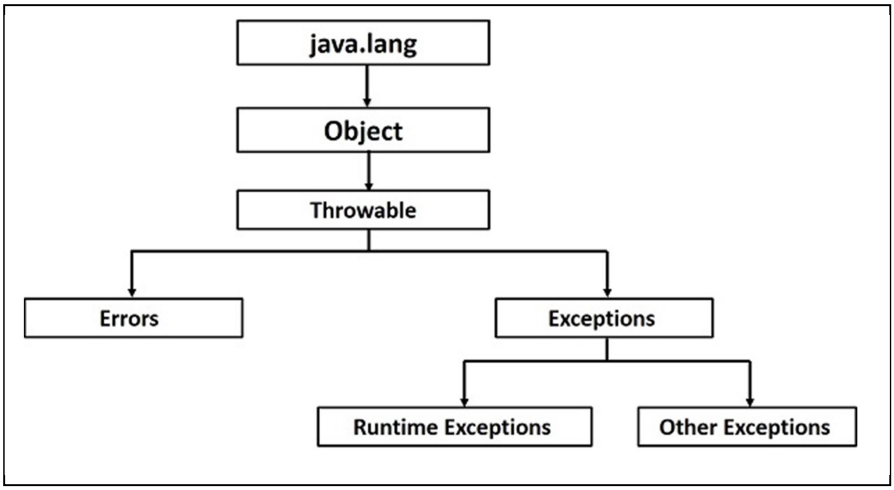
Error: An Error indicates serious problem that a reasonable application should not try to catch.

Exception: Exception indicates conditions that a reasonable application might try to catch.

Exception Handling in Java

The exception handling in java is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.

Hierarchy of Java Exception classes



Types of Exception

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. **Checked Exception:** The exception that can be predicted by the programmer at the compile time. Example: File that need to be opened is not found. These types of exceptions must be checked at compile time.
2. **Unchecked Exception:** Unchecked exceptions are the class that extends RuntimeException. Unchecked exceptions are ignored at compile time. Example: ArithmeticException, NullPointerException, Array Index out of Bound exception. Unchecked exceptions are checked at runtime.
3. **Error:** Errors are typically ignored in code because you can rarely do anything about an error. **Example:** if stack overflow occurs, an error will arise. This type of error cannot be handled in the code.

Exception Handling Mechanism

In java, exception handling is done using five keywords,

1. **try:** The try block contains set of statements where an exception can occur. try block must be followed by either catch or finally block.

   Syntax:

   ```
   try{
     //statements that may cause an exception
   }
   ```

2. **catch:** Java catch block is used to handle the Exception. It must be used after the try block only. You can use multiple catch block with a single try.

   Syntax:

   ```
   try
   {
      //statements that may cause an exception
   }
   catch (exception(type) e(object))
   {
      //error handling code
   }
   ```

3. **finally:** A finally block must be associated with a try block, you cannot use finally without a try block. You should place those statements in this block that must be executed always.

   Syntax:

   ```
   try {
      //Statements that may cause an exception
   }
   catch {
     //Handling exception
   }
   finally {
  //Statements to be executed
}
   ```

4. **throw:** throw keyword is used to throw an exception explicitly. Only object of Throwable class or its sub classes can be thrown.

   Syntax :

   throw *ThrowableInstance*

5. **Throws:** Throws clause is used to declare an exception, which means it works similar to the try-catch block.

Syntax :

```
type method_name(parameter_list) throws exception_list
{
 //definition of method
}
```

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

Example demonstrating try-catch-final keyword

```
class Example

{

  public static void main(String args[]) {

    try{

          int num=121/0;

          System.out.println(num);

    }

    catch(ArithmeticException e){

      System.out.println("Number should not be divided by zero");

    }

    /* Finally block will always execute

     * even if there is no exception in try block

     */

    finally{

          System.out.println("This is finally block");

    }

    System.out.println("Out of try-catch-finally");

  }
```

```
}
```

Output: Number should not be divided by zero

This is finally block

Out of try-catch-finally

Example demonstrating throw and throws Keyword

```
class Test
{
 static void check() throws ArithmeticException
 {
  System.out.println("Inside check function");
  throw new ArithmeticException("demo");
 }
 public static void main(String args[])
 {
  try
  {
   check();
  }
  catch(ArithmeticException e)
  {
   System.out.println("caught" + e);
  }
 }
}
```

Output: Inside check function

caughtjava.lang.ArithmeticException: demo

Multithreading in Java

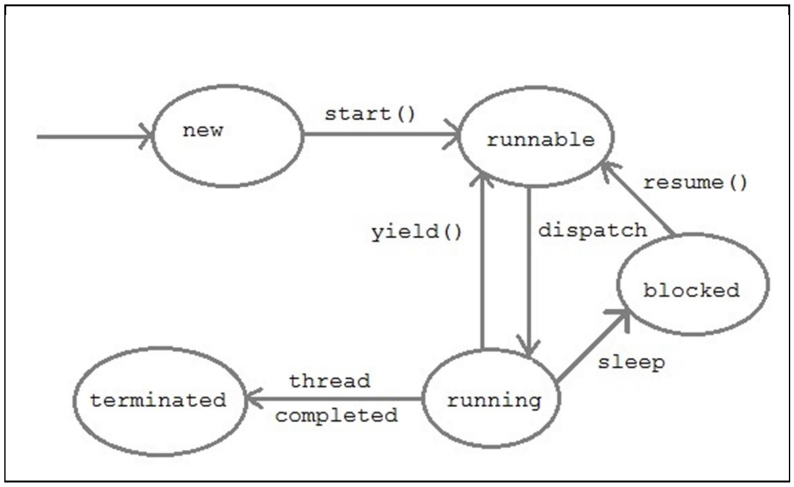
The process of executing multiple threads simultaneously is known as multithreading.

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread.

Thread in Java

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution. Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.

Life cycle of a Thread



New: A thread begins its life cycle in the new state. It remains in this state until the start() method is called on it.

Runnable: After invocation of start() method on new thread, the thread becomes runnable.

Running: A thread is in running state if the thread scheduler has selected it.

Waiting: A thread is in waiting state if it waits for another thread to perform a task. In this stage the thread is still alive.

Terminated: A thread enters the terminated state when it complete its task.

Types of Thread

User thread: This is created by the Java programmer or user. JVM wait for these threads to finish their task. These threads are foreground threads.

Daemon thread: This is created by the operating system. JVM doesn't wait for daemon threads to finish their task. These threads are used to perform some background tasks like garbage collection.

Thread Priorities

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java thread priorities are in the range between **MIN_PRIORITY** (a constant of 1) and **MAX_PRIORITY** (a constant of 10). By default, every thread is given priority **NORM_PRIORITY** (a constant of 5).

Thread class is the main class on which Java's Multithreading system is based. Thread class, along with its companion interface Runnable will be used to create and run threads for utilizing Multithreading feature of Java.

Constructors of Thread class

Thread ( )

Thread (String str)

Thread (Runnable r)

Thread (Runnable r, String str)

Thread class also defines many methods for managing threads. Some of them are,

Method	Description
setName()	to give thread a name
getName()	return thread's name
getPriority()	return thread's priority
isAlive()	checks if thread is still running or not
join()	Wait for a thread to end
run()	Entry point for a thread

sleep()	suspend thread for a specified time

Creating a thread in Java

There are two ways to create a thread in Java:

1) **By extending Thread class:** This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

   Example:

```
class Multi extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi t1=new Multi();
t1.start();
 }
}
```

   Output: thread is running...

2) **By implementing Runnable interface**: If your class is intended to be executed as a thread then you can achieve this by implementing a Runnable interface.

   Example:

```
class Multi implements Runnable{

public void run(){

System.out.println("thread is running...");

}

public static void main(String args[]){

Multi3 m1=new Multi3();

Thread t1 =new Thread(m1);

t1.start();

 }

}
```

   Output: thread is running...

Inter-thread Communication in Java

Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of Object class:

wait()-It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify().

notify()-It wakes up one single thread that called wait() on the same object.

notifyAll()-It wakes up all the threads that called wait() on the same object.

Difference between wait() and sleep()

wait()	sleep()
called from synchronised block	no such requirement
monitor is released	monitor is not released
gets awake when notify() or notifyAll() method is called.	does not get awake when notify() or notifyAll() method is called
not a static method	static method
wait() is generaly used on condition	sleep() method is simply used to put your thread on sleep.

Thread Deadlock

Deadlock is a situation of complete Lock, when no thread can complete its execution because lack of resources. Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Deadlock occurs when multiple threads need the same locks but obtain them in different order.

Example of deadlock

```
class Pen{}

class Paper{}

public class Write {

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

```
{
  final Pen pn =new Pen();
  final Paper pr =new Paper();
  Thread t1 = new Thread(){
       public void run()
       {
          synchronized(pn)
          {
             System.out.println("Thread1 is holding Pen");
           try{
              Thread.sleep(1000);
           }catch(InterruptedException e){}
             synchronized(pr)
          {  System.out.println("Requesting for Paper");
          }
       }
    };
   Thread t2 = new Thread(){
       public void run()
       {
          synchronized(pr)
          {
             System.out.println("Thread2 is holding Paper");
           try{
```

```
                    Thread.sleep(1000);
                  }catch(InterruptedException e){}
                    synchronized(pn)
                  {  System.out.println("requesting for Pen"); }
                        }
               }
            };
            t1.start();
            t2.start();
        }
      }
```

Output: Thread1 is holding Pen

Thread2 is holding Paper