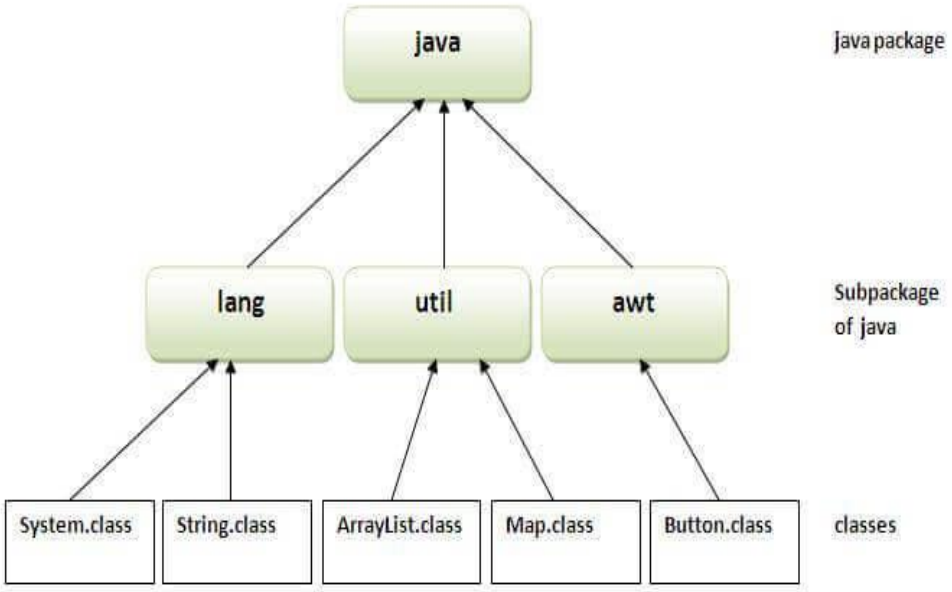
UNIT-II

1.Discuss about packages in Java.

A package is Java's style of bundling classes together. A package is a collection of related classes and interfaces. A package does not mean only predefined classes; a package may contain user defined classes also. A package is equivalent to a header file of C-lang. Packages can be compressed into JAR files for fast traversal in a network or to download from Internet.

ADVANTAGES OF PACKAGES

- Like header files, packages come with many advantages.
- With a single import statement, all the classes and interfaces can be obtained into our program.
- Unlike a header file, Java permits to import even a single class also.
- Avoids namespace problems. Two classes of the same name cannot be put in the same package but can be placed in two different packages.
- Access between the classes can be controlled. Using packages, restrictions can be imposed on the access of other package classes. Access specifiers work on package boundaries (between the classes of other packages).
- We can find all the related classes and interfaces in a single space. Searching and identification will be easier.
- Packages and sub-packages are the easiest way to organize the classes.



To access the package from outside the package.(using import keyword)

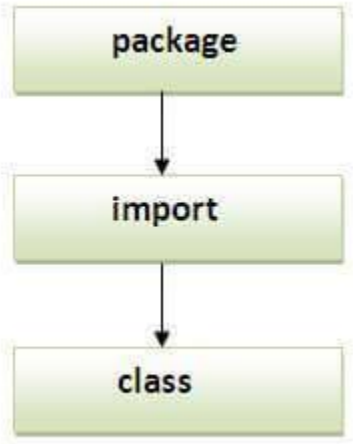
- import package.*;
- import package.classname; fully qualified name.

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages. **The import keyword** is used to make the classes and interface of another package accessible to the current package.

2.Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.



Subpackage in java Package inside the package is called the subpackage. It should be created to categorize the package further.

- Import mainpackage.subpackage.*; to access all classes in subpackage.
- Import mainpackage.subpackage.Classname; to access specific class in subpackage.

System Predefined packages in Java

Java provides various predefined classes and interfaces (API's) organized under packages. These are known as predefined packages, following is the list of predefined packages in java −

- java.lang − This package provides the language basics.
- java.util − This packages provides classes and interfaces (API's) related to collection frame work, events, data structure and other utility classes such as date.
- java.io − This packages provides classes and interfaces for file operations, and other input and output operations.
- java.math − This packages provides classes and interfaces for multiprecision arithmetics.
- java.nio − This packages provides classes and interfaces the Non-blocking I/O framework for Java
- java.net − This packages provides classes and interfaces related to networking.
- java.security − This packages provides classes and interfaces such as key generation, encryption and decryption which belongs to security frame work.

- java.sql − This packages provides classes and interfaces for accessing/manipulating the data stored in databases and data sources.
- java.awt − This packages provides classes and interfaces to create GUI components in Java.
- java.text − This packages provides classes and interfaces to handle text, dates, numbers, and messages.
- java.rmi − Provides the RMI package.
- java.time − The main API for dates, times, instants, and durations.
- java.beans − The java.beans package contains classes and interfaces related to JavaBeans components.

2. How to create used defined package in Java.

Creating a package

You can create a package and add required classes/interfaces in it just by declaring the package on the top of the Class/Interface files using the keyword package as –

```
package package_name;
```

To compile this program (programs with packages), you need to use the –d option of the javac command. At this you need to specify the path where you need to create the package.

javac -d E:\ Simple.java

//save as Simple.java

package mypack; →user defined package name

public class Simple{

public static void main(String args[]){

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

}

}

To Compile:

e:\sources> javac -d c:\classes Simple.java

To Run:

To run this program from e:\source directory, you need to set classpath of the directory where the class file resides.

e:\sources> set classpath=c:\classes;.;

e:\sources> java mypack.Simple

2. write about Access Modifiers in Java

- Private access modifier
- Role of private constructor
- Default access modifier
- Protected access modifier
- Public access modifier
- Access Modifier with Method Overriding

There are two types of modifiers in Java: access modifiers and non-access modifiers.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.
5. 's understand the access modifiers in Java by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

3. What is an Exception and give types of 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, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Based on these, we have three categories of Exceptions.

Checked exceptions − A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called as compile time exceptions. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.

```
import java.io.File;
import java.io.FileReader;

public class FilenotFound_Demo {

   public static void main(String args[]) {
      File file = new File("E://file.txt");
      FileReader fr = new FileReader(file);
   }
}
```

In above code FileReader class throws FileNotFound exception it should be handled and checked by compiler.

Unchecked exceptions − An unchecked exception is an exception that occurs at the time of execution. These are also called as Runtime Exceptions. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

```
public class Unchecked_Demo {

   public static void main(String args[]) {
      int num[] = {1, 2, 3, 4};
      System.out.println(num[5]);
   }
}
```

Here array size in not checked by compiler and hence error is occurs at runtime.

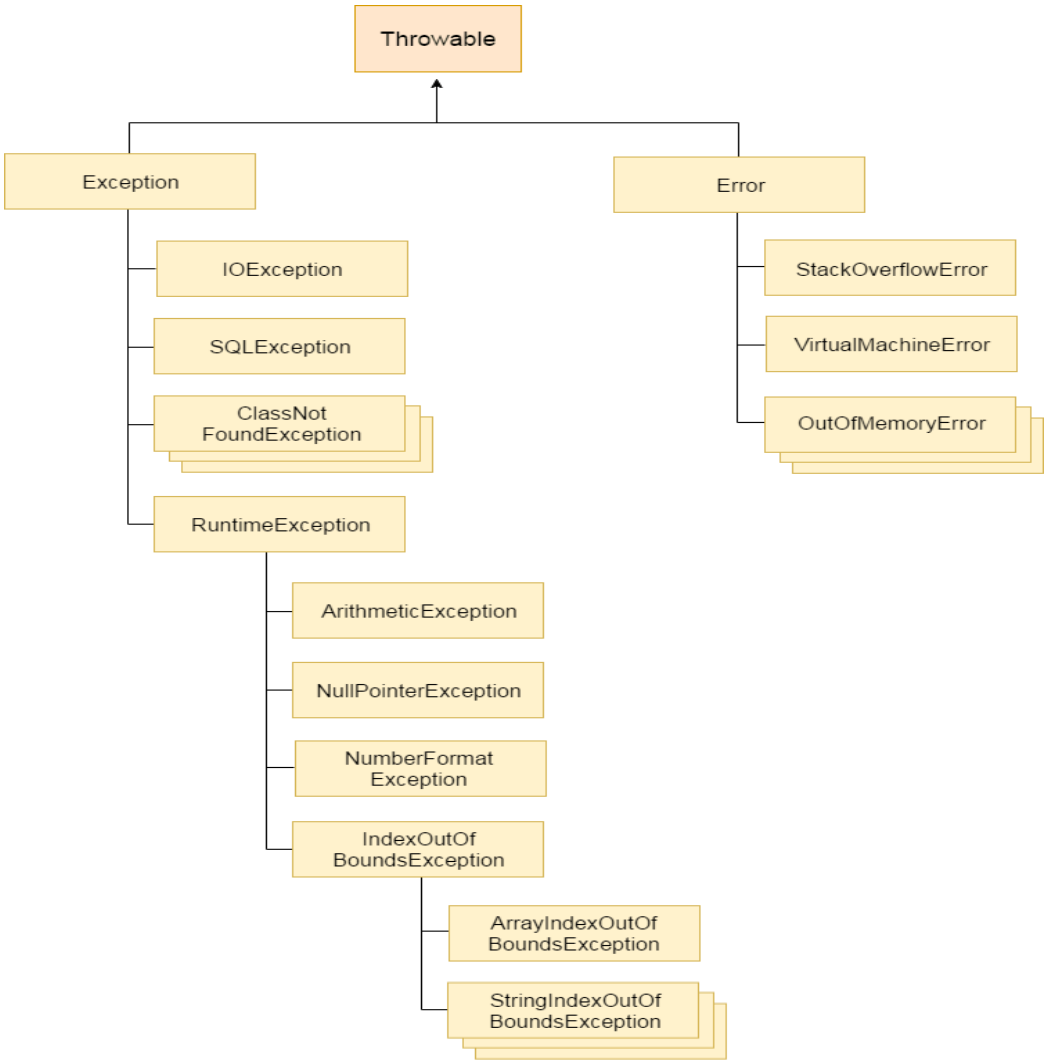
Errors − These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything

about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

4.Exception Hierarchy

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of memory. Normally, programs cannot recover from errors.



5. Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

6 Exception Class Methods

Exceptions Methods

Sr.No.	Method & Description
1	**public String getMessage()** Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2	**public Throwable getCause()** Returns the cause of the exception as represented by a Throwable object.
3	**public String toString()** Returns the name of the class concatenated with the result of getMessage().
4	**public void printStackTrace()** Prints the result of toString() along with the stack trace to System.err, the error output stream.

7.Exception handling Mechanism in Java.

In Java an exception handling is done using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception.

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following –

Syntax

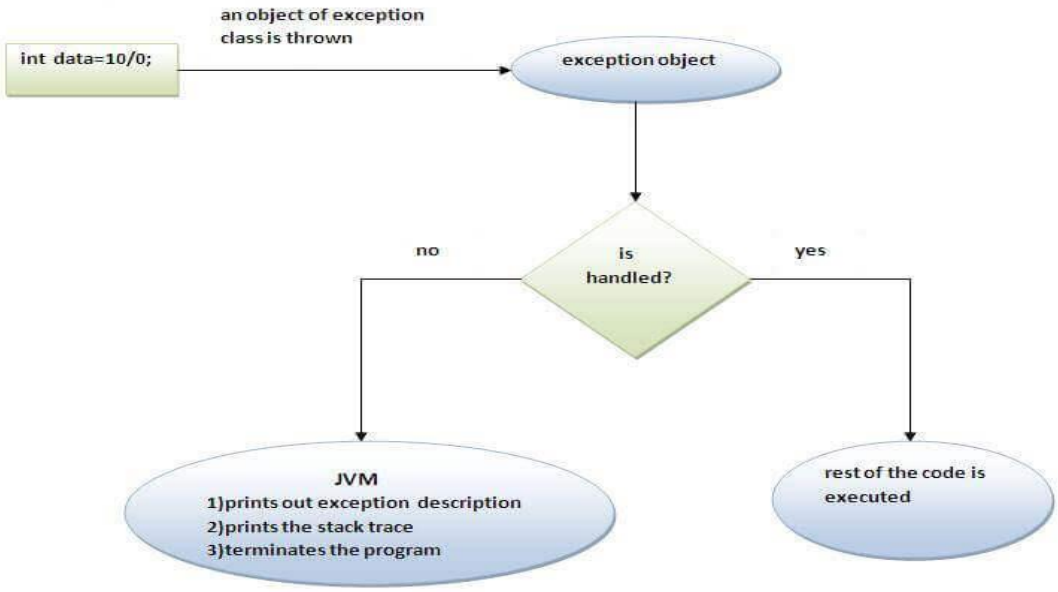
```
try {
   // Protected code
// that cause or raise exceptions

}
catch (ExceptionName e1)
 {
   // Catch block
}
```

Example:-

```
public class JavaExceptionExample{
  public static void main(String args[]){
   try{
      //code that may raise exception
      int data=100/0;
   }catch(ArithmeticException e){System.out.println(e);}
   //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

Exception Examples:

```
int a=50/0;//ArithmeticException
String s=null;
System.out.println(s.length());//NullPointerException
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```



The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

Java Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

```
public class MultipleCatchBlock1 {

   public static void main(String[] args) {

        try{
            int a[]=new int[5];
            a[5]=30/0;
           }
           catch(ArithmeticException e)
              {
               System.out.println("Arithmetic Exception occurs");
              }
           catch(ArrayIndexOutOfBoundsException e)
              {
               System.out.println("ArrayIndexOutOfBounds Exception occurs");
```

```
            }
          catch(Exception e)
            {
             System.out.println("Parent Exception occurs");
            }
          System.out.println("rest of the code");
   }
}
```

Java finally block

Java finally block is a block that is used to execute important code such as closing connection, stream etc. Java finally block is always executed whether exception is handled or not. Java finally block follows try or catch block.

```
public class TestFinallyBlock2{
  public static void main(String args[]){
  try{
   int data=25/0;
   System.out.println(data);
  }
  catch(ArithmeticException e){System.out.println(e);}
  finally{
System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

8.Discuss about throw & throws keyword.

If a method does not handle a checked exception, the method must declare it using the throws keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the throw keyword.

Try to understand the difference between throws and throw keywords, throws is used to postpone the handling of a checked exception and throw is used to invoke an exception explicitly.

The following method declares that it throws a RemoteException –

```
import java.io.*;
public class className {

   public void deposit(double amount) throws RemoteException {
      // Method implementation
      throw new RemoteException();
   }
   // Remainder of class definition
}
```

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException –

```
import java.io.*;
public class className {

  public void withdraw(double amount) throws RemoteException,
    InsufficientFundsException {
    // Method implementation
  }
  // Remainder of class definition
```

9. Write about User-defined Exceptions

Exceptions created or customized for user applications are known as user defined Exceptions. All exceptions must be a child of Throwable.

If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.

If you want to write a runtime exception, you need to extend the RuntimeException class.

We can define our own Exception class as below −

```
class MyException extends Exception {
}
// File Name InsufficientFundsException.java
import java.io.*;

public class InsufficientFundsException extends Exception {
  private double amount;

  public InsufficientFundsException(double amount) {
    this.amount = amount;
  }

  public double getAmount() {
    return amount;
  }
}
```

10.What is a Thread and discuss about life cycle of a Thread.

Multithreading in java is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking. However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Advantages of Java Multithreading

1) It doesn't block the user because threads are independent and you can perform multiple operations at the same time.

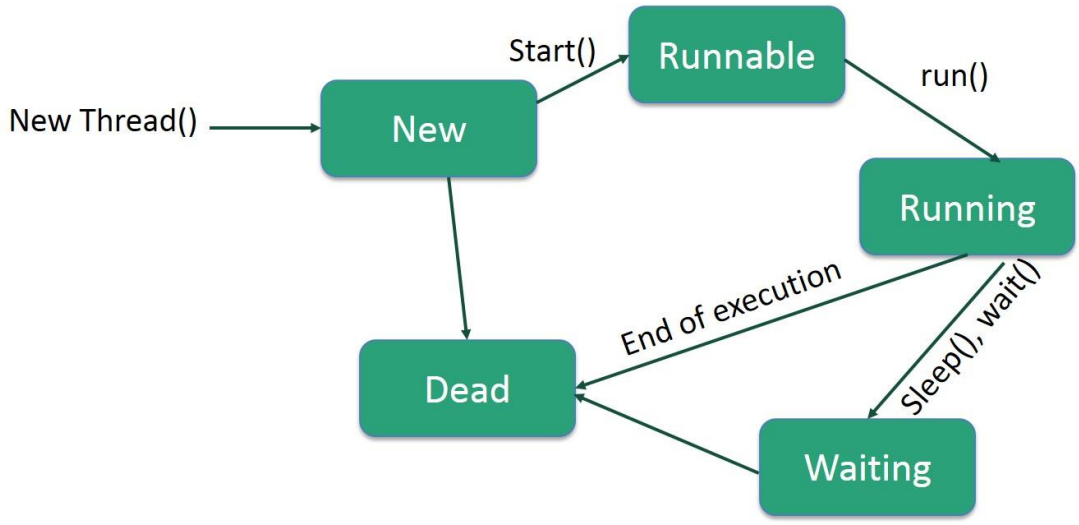
2) You can perform many operations together, so it saves time.

3) Threads are independent, so it doesn't affect other threads if an exception occurs in a single thread.

Life Cycle of a Thread

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.

Following are the stages of the life cycle –



New − A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.

Runnable − After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

Waiting − Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

Timed Waiting − A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

Terminated (Dead) − A runnable thread enters the terminated state when it completes its task or otherwise terminates.

In Java we use Thread class or Runnable Interface to create threads in a program.

11. Explain about creating Threads in Java.

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. So, threads are light-weight processes within a process.

Threads can be created by using two mechanisms :
1. Extending the Thread class
2. Implementing the Runnable Interface

Thread creation by extending the Thread class

We create a class that extends the java.lang.Thread class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

```
class MultithreadingDemo extends Thread
{
   public void run()
   {
      try
      {
         // Displaying the thread that is running
         System.out.println ("Thread " +
               Thread.currentThread().getId() +
               " is running");

      }
      catch (Exception e)
      {
         // Throwing an exception
         System.out.println ("Exception is caught");
      }
   }
}

// Main Class
```

```
public class Multithread
{
   public static void main(String[] args)
   {
      int n = 8; // Number of threads
      for (int i=0; i<8; i++)
      {
         MultithreadingDemo object = new MultithreadingDemo();
         object.start();
      }
   }
}
```

The Thread class defines several methods that help manage threads:

Method	**Meaning**
getName	Obtain thread's name
getPriority	Obtain thread's priority
isAlive	Determine if a thread is still running
join	Wait for a thread to terminate
run	Entry point for the thread
sleep	Suspend a thread for a period of time
start	Start a thread by calling its run method

Runnable Interface
The easiest way to create a thread is to create a class that implements the Runnable interface.

To implement Runnable interface, a class need only implement a single method called run( ), which is declared like this:

public void run( )
Inside run( ), we will define the code that constitutes the new thread. Example:

```
public class MyClass implements Runnable {
public void run(){
System.out.println("MyClass running");   } }
```

To execute the run() method by a thread, pass an instance of MyClass to a Thread in its constructor (A constructor in Java is a block of code similar to a method that's called when an instance of an object is created). Here is how that is done:

```
Thread t1 = new Thread(new MyClass ());
t1.start();
```

When the thread is started it will call the run() method of the MyClass instance instead of executing its own run() method. The above example would print out the text "MyClass running.

Multiple Threads

```
class MyThread implements Runnable {
String name;
Thread t;
   MyThread String thread){
   name = threadname;
   t = new Thread(this, name);
System.out.println("New thread: " + t);
t.start();
}
public void run() {
 try {
    for(int i = 5; i > 0; i--) {
    System.out.println(name + ": " + i);
     Thread.sleep(1000);
}
}catch (InterruptedException e) {
    System.out.println(name + "Interrupted");
}
    System.out.println(name + " exiting.");
}
}
class MultiThread {
public static void main(String args[]) {
    new MyThread("One");
    new MyThread("Two");
    new NewThread("Three");
try {
    Thread.sleep(10000);
} catch (InterruptedException e) {
     System.out.println("Main thread Interrupted");
}
     System.out.println("Main thread exiting.");
     }
}
```

Sleep method in java
The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

Syntax of sleep() method in java
The Thread class provides two methods for sleeping a thread:

public static void sleep(long miliseconds)throws InterruptedException
public static void sleep(long miliseconds, int nanos)throws InterruptedException

The join() method
The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

Syntax:
public void join()throws InterruptedException
public void join(long milliseconds)throws InterruptedException
Example of join() method

```
class TestJoinMethod1 extends Thread{
 public void run(){
  for(int i=1;i<=5;i++){
   try{
    Thread.sleep(500);
   }catch(Exception e){System.out.println(e);}
  System.out.println(i);
  }
 }
public static void main(String args[]){
 TestJoinMethod1 t1=new TestJoinMethod1();
 TestJoinMethod1 t2=new TestJoinMethod1();
 TestJoinMethod1 t3=new TestJoinMethod1();
 t1.start();
 try{
  t1.join();
 }catch(Exception e){System.out.println(e);}

 t2.start();
 t3.start();
 }
}
```

12. Write about Priorities in Thread.

Priority of a Thread (Thread Priority):

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

public static int MIN_PRIORITY
public static int NORM_PRIORITY
public static int MAX_PRIORITY

```
class TestMultiPriority1 extends Thread{
 public void run(){
   System.out.println("running thread name is:"+Thread.currentThread().getName());
   System.out.println("running thread priority is:"+Thread.currentThread().getPriority());

  }
 public static void main(String args[]){
  TestMultiPriority1 m1=new TestMultiPriority1();
  TestMultiPriority1 m2=new TestMultiPriority1();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();
    }  }
```

1.Write short notes on Math class in Java.

Math Class

public final class **Math is in java.lang package which** extends Object

The class Math contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

Properties:

static double E

The double value that is closer than any other to e, the base of the natural logarithms.

static double PI

The double value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter.

Usage: Math.PI or Math.E

Methods:

`static double`	**`abs`**`(double a)` Returns the absolute value of a `double` value.
`static double`	**`cbrt`**`(double a)` Returns the cube root of a `double` value.
`static double`	**`ceil`**`(double a)` Returns the smallest (closest to negative infinity) `double` value that is greater than or equal to the argument and is equal to a mathematical integer.
`static double`	**`cos`**`(double a)` Returns the trigonometric cosine of an angle.
`static double`	**`cosh`**`(double x)` Returns the hyperbolic cosine of a `double` value.
`static double`	**`exp`**`(double a)` Returns Euler's number *e* raised to the power of a `double` value.
`static double`	**`floor`**`(double a)` Returns the largest (closest to positive infinity) `double` value that is less than or equal to the argument and is equal to a mathematical integer.
`static long`	**`floorMod`**`(long x, long y)` Returns the floor modulus of the `long` arguments.
`static double`	**`IEEEremainder`**`(double f1, double f2)` Computes the remainder operation on two arguments as prescribed by the IEEE 754 standard.
`static int`	**`incrementExact`**`(int a)` Returns the argument incremented by one, throwing an exception if the result

	overflows an `int`.
`static double`	**`log`**`(double a)` Returns the natural logarithm (base *e*) of a `double` value.
`static double`	**`log10`**`(double a)` Returns the base 10 logarithm of a `double` value.
`static int`	**`max`**`(int a, int b)` Returns the greater of two `int` values.
`static int`	**`min`**`(int a, int b)` Returns the smaller of two `int` values.
`static long`	**`min`**`(long a, long b)` Returns the smaller of two `long` values.
`static double`	**`pow`**`(double a, double b)` Returns the value of the first argument raised to the power of the second argument.
`static double`	**`random`**`()` Returns a `double` value with a positive sign, greater than or equal to `0.0` and less than `1.0`.
`static double`	**`sin`**`(double a)` Returns the trigonometric sine of an angle.
`static double`	**`sinh`**`(double x)` Returns the hyperbolic sine of a `double` value.
`static double`	**`sqrt`**`(double a)` Returns the correctly rounded positive square root of a `double` value.
`static double`	**`tan`**`(double a)` Returns the trigonometric tangent of an angle.
`static double`	**`tanh`**`(double x)` Returns the hyperbolic tangent of a `double` value.
`static double`	**`toDegrees`**`(double angrad)` Converts an angle measured in radians to an approximately equivalent angle measured in degrees.
`static double`	**`toRadians`**`(double angdeg)` Converts an angle measured in degrees to an approximately equivalent angle measured in radians.

Ex:- z=Math.sin(x) + Math.sqrt(x *x)+Math.log(y);

2. Discuss about String Handling features in Java.

The class String includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase

The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared.

Strings are created by using Literal or by Constructor.

By using Literal as below :

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
String str = new String(data)
```

By using Constructor

String()
Initializes a newly created String object so that it represents an empty character sequence.
String(byte[] bytes)
Constructs a new String by decoding the specified array of bytes using the platform's default charset.
String(byte[] bytes, **Charset** charset)
Constructs a new String by decoding the specified array of bytes using the specified **charset**.

Methods:

char	**charAt**(int index)
	Returns the char value at the specified index.
int	**compareTo**(**String** anotherString)
	Compares two strings lexicographically.
int	**compareToIgnoreCase**(**String** str)
	Compares two strings lexicographically, ignoring case differences.

String	**concat**(**String** str) Concatenates the specified string to the end of this string.
boolean	**contains**(**CharSequence** s) Returns true if and only if this string contains the specified sequence of char values.
static **String**	**copyValueOf**(char[] data) Returns a String that represents the character sequence in the array specified.
boolean	**endsWith**(**String** suffix) Tests if this string ends with the specified suffix.
boolean	**equals**(**Object** anObject) Compares this string to the specified object.
boolean	**equalsIgnoreCase**(**String** anotherString) Compares this `String` to another `String`, ignoring case considerations.
byte[]	**getBytes**() Encodes this `String` into a sequence of bytes using the platform's default charset, storing the result into a new byte array.
byte[]	**getBytes**(**Charset** charset) Encodes this `String` into a sequence of bytes using the given **charset**, storing the result into a new byte array.
void	**getChars**(int srcBegin, int srcEnd, char[] dst, int dstBegin) Copies characters from this string into the destination character array.
int	**hashCode**() Returns a hash code for this string.
int	**indexOf**(int ch) Returns the index within this string of the first occurrence of the specified character.
int	**indexOf**(**String** str) Returns the index within this string of the first occurrence of the specified substring.
boolean	**isEmpty**() Returns `true` if, and only if, **length()** is `0`.

int	**lastIndexOf**(int ch) Returns the index within this string of the last occurrence of the specified character.
int	**length**() Returns the length of this string.
boolean	**matches**(**String** regex) Tells whether or not this string matches the given **regular expression**.
boolean	**regionMatches**(int toffset, **String** other, int ooffset, int len) Tests if two string regions are equal.
String	**replace**(char oldChar, char newChar) Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
String	**replace**(**CharSequence** target, **CharSequence** replacement) Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence.
String[]	**split**(**String** regex) Splits this string around matches of the given **regular expression**.
String[]	**split**(**String** regex, int limit) Splits this string around matches of the given **regular expression**.
boolean	**startsWith**(**String** prefix) Tests if this string starts with the specified prefix.
String	**substring**(int beginIndex) Returns a new string that is a substring of this string.
String	**substring**(int beginIndex, int endIndex) Returns a new string that is a substring of this string.
char[]	**toCharArray**() Converts this string to a new character array.
String	**toLowerCase**() Converts all of the characters in this String to lower case using the rules of the default locale.

String	**toString**() This object (which is already a string!) is itself returned.
String	**toUpperCase**() Converts all of the characters in this `String` to upper case using the rules of the default locale.
String	**toUpperCase**(**Locale** locale) Converts all of the characters in this `String` to upper case using the rules of the given `Locale`.
String	**trim**() Returns a copy of the string, with leading and trailing whitespace omitted.
static **String**	**valueOf**(boolean b) Returns the string representation of the `boolean` argument.
static **String**	**valueOf**(char c) Returns the string representation of the `char` argument.
static **String**	**valueOf**(double d) Returns the string representation of the `double` argument.
static **String**	**valueOf**(float f) Returns the string representation of the `float` argument.
static **String**	**valueOf**(int i) Returns the string representation of the `int` argument.

Ex:-

```
public class SubstringExample{

public static void main(String args[]){

String s1="KMMCollege";

System.out.println(s1.substring(2,4));//returns MC

System.out.println(s1.substring(2));//returns MCollege

}}
```