UNIT-3

Interfaces: Defining an interface, implementing interfaces, nested interfaces, variables in interfaces and extending interfaces, multiple inheritance, class within interface and interface within class, up-casting and down-casting, marker interfaces, interfaces with default and static methods, Introduction to Collection framework, lambda expressions in Java.

Packages: Predefined packages: java.util.Date, java.util.Random, java.util.Arrays, creating and accessing user defined packages, package hierarchy, and multiple classes in single package and access controls with packages.

Interfaces:

An interface contains only abstract methods which are all incomplete methods. So it is not possible to create an object to an interface. In this case, we can create separate classes where we can implement all the methods of the interface. These classes are called implementation classes. Since, implementation classes will have all the methods with body; it is possible to create objects to the implementation classes.

Syntax:

```
interface MyInter
{
        constants
        abstract methods
}
```

Relationship between classes and Interfaces:

- One class can "implements" one or more interfaces.
- On interface can "extends" one interface.
- One class can "extends" another class.
- One interface cannot "extends" or "implements" another class.

Declaration of Interface:

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Syntax:

```
interface <interface_name>{

   // declare constant fields
   // declare methods that abstract
   // by default.
}
```

Multiple Interfaces:

Example: Write a Java Program to illustrate multiple inheritance using multiple interfaces.

```
interface Father
{
        double HT=6.2;
```

```
        void height();
}
interface Mother
{
        double HT=5.8;
        void color();
}
class Child implements Father, Mother
{
        public void height()
        {
                double ht=(Father.HT+Mother.HT)/2;
                System.out.println("Child's Height= "+ht);
        }
        public void color()
        {
                System.out.println("Child Color= brown");
        }
        public static void main(String[] args)
        {
                Child c=new Child();
                c.height();
                c.color();
        }
}
```

Output:

```
Child's Height= 6.0
Child Color= brown
```

Nested Interfaces:

An interface, i.e., declared within another interface or class, is known as a nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred to by the outer interface or class. It can't be accessed directly.

Points to remember:

- There are given some points that should be remembered by the java programmer.
- The nested interface must be public if it is declared inside the interface, but it can have any access modifier if declared within the class.
- Nested interfaces are declared static

Syntax:

```
interface interface_name
{
 ...
 interface nested_interface_name
```

```
{
  ...
 }
}
```

Example:

```
interface Showable
{
  void show();
  interface Message
{
   void msg();
  }
}
class TestNestedInterface1 implements Showable.Message
{
 public void msg()
{
System.out.println("Hello nested interface");
}
 public static void main(String args[])
{
Showable.Message message=new TestNestedInterface1();//upcasting here
message.msg();
 }
}
```

Output:

```
hello nested interface
```

Inheritance of Interfaces:

A class implements an interface, but one interface extends another interface.

Example:

```
interface Printable
{
void print();
}
interface Showable extends Printable
{
void show();
}
class TestInterface4 implements Showable
{
public void print()
{
```

```
System.out.println("Hello");
}
public void show()
{
System.out.println("Welcome");
}
public static void main(String args[])
{
TestInterface4 obj = new TestInterface4();
obj.print();
obj.show();
 }
}
```

Output:

```
Hello
Welcome
```

Default Methods in Interfaces:

Since Java 8, we have method body in interface. But we need to make it default method.

Example:

```
interface Drawable
{
void draw();
default void msg()
{
System.out.println("default method");
}
}
class Rectangle implements Drawable
{
public void draw()
{
System.out.println("drawing rectangle");
}
}
class TestInterfaceDefault
{
public static void main(String args[])
{
Drawable d=new Rectangle();
d.draw();
d.msg();
}
```

```
}
```

Output:

```
drawing rectangle
default method
```

Static Methods in Interface:

We have static method in interface.

Example:

```
interface Drawable
{
void draw();
static int cube(int x)
{
return x*x*x;
}
}
class Rectangle implements Drawable
{
public void draw()
{
System.out.println("drawing rectangle");
}
}
class TestInterfaceStatic
{
public static void main(String args[])
{
Drawable d=new Rectangle();
d.draw();
System.out.println(Drawable.cube(3));
}
}
```

Output:

```
drawing rectangle
27
```

Functional Interfaces:

A functional interface is an interface that contains only one abstract method. They can have only one functionality to exhibit.

Example:

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

```
                // create anonymous inner class object
                new Thread(new Runnable()
                {
                        @Override
                        public void run()
                        {
                                System.out.println("New thread created");
                        }
                }).start();
        }
}
```

Output:

New thread created

Up-casting and down-casting:

Up-casting:

It is a type of object typecasting in which a child object is type casted to a parent class object. By using this, we can easily access the variables and methods of the parent class to the child class. Here, we don't access all the variables and the method. We access only some specified variables and methods of the child class. It is also known as Generalization and Widening.

Example:

```
class  Parent{
  void PrintData() {
    System.out.println("method of parent class");
  }
}
class Child extends Parent {
  void PrintData() {
    System.out.println("method of child class");
  }
}
class UpcastingExample{
  public static void main(String args[]) {
    Parent obj1 = (Parent) new Child();
    Parent obj2 = (Parent) new Child();
    obj1.PrintData();
    obj2.PrintData();
  }
}
```

Output:

```
method of child class
method of child class
```

Downcasting:

Upcasting is another type of object typecasting. In Upcasting, we assign a parent class reference object to the child class. In Java, we cannot assign a parent class reference object to the child class, but if we perform downcasting, we will not get any compile-time error. However, when we run it, it throws the "ClassCastException". Now the point is if downcasting is not possible in Java, then why is it allowed by

the compiler? In Java, some scenarios allow us to perform downcasting. Here, the subclass object is referred by the parent class.

Example:

```
class Parent {
   String name;
   // A method which prints the data of the parent class
   void showMessage()
   {
      System.out.println("Parent method is called");
   }
}
// Child class
class Child extends Parent {
   int age;
   // Performing overriding      @Override
   void showMessage()
   {
      System.out.println("Child method is called");
   }
}
public class Downcasting{
      public static void main(String[] args)
   {
      Parent p = new Child();
      p.name = "Shubham";
      // Performing Downcasting Implicitly
      //Child c = new Parent(); // it gives compile-time error
      // Performing Downcasting Explicitly
      Child c = (Child)p;
      c.age = 18;
      System.out.println(c.name);
      System.out.println(c.age);
      c.showMessage();
   }
}
```

Marker interfaces:

An interface that does not contain methods, fields, and constants is known as marker interface. In other words, an empty interface is known as marker interface or tag interface. It delivers the run-time type information about an object. It is the reason that the JVM and compiler have additional information about an object. The Serializable and Cloneable interfaces are the example of marker interface. In short, it indicates a signal or command to the JVM.

The declaration of marker interface is the same as interface but the interface must be empty.

Example:

```
public interface Serializable
{

}
```

Cloneable Interface:

Cleanable interface is also a marker interface that belongs to java.lang package. It generates replica (copy) of an object with different name. We can implement the interface in the class of which class object to be cloned. It indicates the clone() method of the Object class. If we do not implement the Cloneable interface in the class and invokes the clone() method, it throws the ClassNotSupportedException.

Example:

```
import java.util.Scanner;
public class Product implements Cloneable
{
	int pid;
	String pname;
	double pcost;
	//Product class constructor
	public Product (int pid, String pname, double pcost)
	{
		this.pid = pid;
		this.pname = pname;
		this.pcost = pcost;
	}
	//method that prints the detail on the console
	public void showDetail()
	{
		System.out.println("Product ID: "+pid);
		System.out.println("Product Name: "+pname);
		System.out.println("Product Cost: "+pcost);
	}
	public static void main (String args[]) throws CloneNotSupportedException
	{
		//reading values of the product from the user
		Scanner sc = new Scanner(System.in);
		System.out.print("Enter product ID: ");
		int pid = sc.nextInt();
		System.out.print("Enter product name: ");
		String pname = sc.next();
		System.out.print("Enter product Cost: ");
		double pcost = sc.nextDouble();
		System.out.println("-------Product Detail--------");
		Product p1 = new Product(pid, pname, pcost);
		//cloning the object of the Product class using the clone() method
		Product p2 = (Product) p1.clone();
		//invoking the method to print detail
		p2.showDetail();
	}
}
```

Introduction to Collection framework:

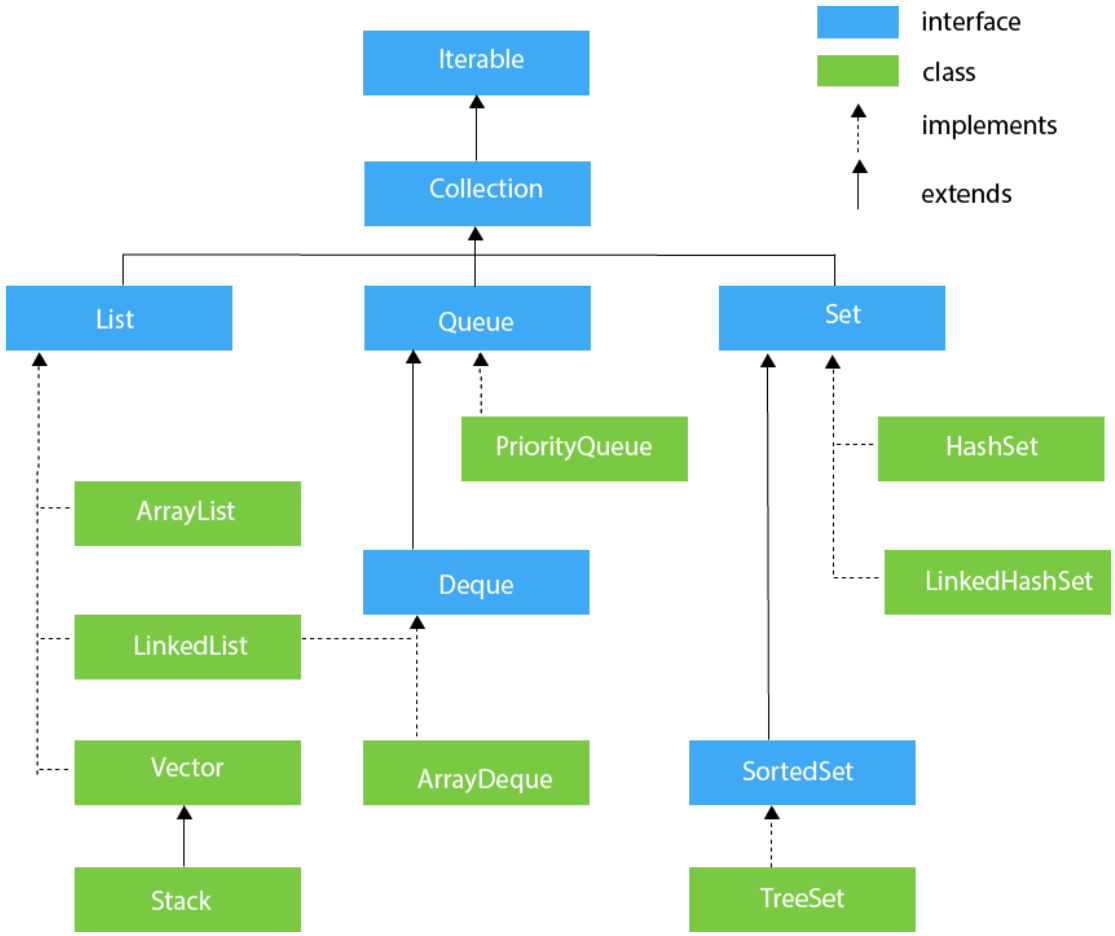
The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

What is Collection framework

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes
2. Algorithm



ArrayList:

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed.

```
import java.util.*;
class TestJavaCollection1{
public static void main(String args[])
{
        ArrayList<String> list=new ArrayList<String>();//Creating arraylist
        list.add("Ravi");//Adding object in arraylist
        list.add("Vijay");
        list.add("Ravi");
```

```
        list.add("Ajay");
        //Traversing list through Iterator
        Iterator itr=list.iterator();
        while(itr.hasNext()){
        System.out.println(itr.next());
    }
    }
    }
```

HashSet:

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

Example:

```
    import java.util.*;
    public class TestJavaCollection7{
    public static void main(String args[]){
    //Creating HashSet and adding elements
    HashSet<String> set=new HashSet<String>();
    set.add("Ravi");
    set.add("Vijay");
    set.add("Ravi");
    set.add("Ajay");
    //Traversing elements
    Iterator<String> itr=set.iterator();
    while(itr.hasNext()){
    System.out.println(itr.next());
    }
    }
    }
```

Output:

```
        Vijay
        Ravi
        Ajay
```

Lambda Expressions:

A lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

Syntax:

A single parameter and an expression:

```
        parameter -> expression
```

To use more than one parameter:

```
    (parameter1, parameter2) -> expression
```

Expressions are limited. They have to immediately return a value, and they cannot contain variables, assignments or statements such as if or for. In order to do more complex operations, a code block can be used with curly braces. If the lambda expression needs to return a value, then the code block should have a return statement.

(parameter1, parameter2) -> { *code block* }

Example:

```
import java.util.ArrayList;
public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<Integer>();
    numbers.add(5);
    numbers.add(9);
    numbers.add(8);
    numbers.add(1);
    numbers.forEach( (n) -> { System.out.println(n); } );
  }
}
```

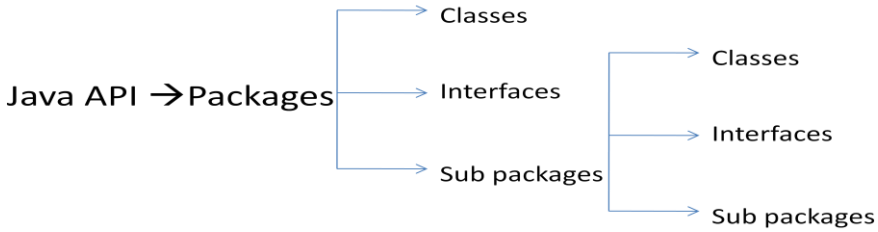
Output:

```
5
9
8
1
```

Introduction:

Learning java is nothing but learning about its concepts its library and its syntaxes.

- Library of java language is known as API. API of java is collection of packages.
- Packages are useful to arrange group of classes and interfaces into a group.
- It puts all together the classes and interfaces performing the same task in the same package.
- Packages hide the classes and interfaces into a sub directory.
- The classes and interfaces of a package are isolated from the classes and interfaces in another package. This means that you can use same name of a class of two different packages.
- A group of packages is called a library. The classes and interfaces of a package are like books in a library and can be reused several times.



The purpose of packages is to provide common classes and interfaces to the programmers

Definition:

A package is a collection of classes, interfaces and sub packages. A sub package in turns contains classes, interfaces and sub packages etc.,.

Types of packages:

1. Predefined /built in packages
2. User defined/custom defined
3. Third party

Predefined packages:

Predefined packages are those which are developed by sun micro systems and supplied as a path of JDK to deal with universal requirements.

User defined packages:

User defined packages are those which are developed by Java programmer and supplied as a part of project to deal with common requirements.

Third party packages:

This packages or developed by third party software Windows and release to this industry Java programmers to communicate with third party software product from the Java program.

Types of pre defined packages:

Predefined packages are classified into three types:

1. JSE/ core package
2. JEE/ advanced package
3. JME/ mobile packages

JSE/ core package:

This packages are used for developing client side applications

JEE/ advanced package:

These packages are used for developing server side applications

JME/ mobile packages:

These packages are used for developing wireless based applications

List of Pre defined JSE packages:

They are 8 essential packages. They are:

java. lang.*:

The basic aim of this language is to provide language functionality/ services/ facilities provided to the programmer.

→Some of the language or functionalities are:

1. Accessing the command line arguments
2. Displaying the data on the console
3. Data conversions
4. Provide the garbage collection facility
5. Developing thread based applications etc.

→This is the package which is imported by default to each and every Java program. hence this package is known as default package.

java.awt.*:

→The aim of this package is to design a GUI applications. to develop and GUI components and all this GUI components are given by Sun Micro system in the form of predefined classes such as label, button, checkbox, text field etc.

Ex: Button b=new Button("save");

java.awt.event.*:

→Here event is the one of the Sub package of a AWT package.

→Event package contains collection of predefined classes and interfaces which are used for providing functionality/ life/ behavior to GUI component.

→To develop component GUI application we need to import both java.awt.*(to design GUI application) &Java.awt.event.*(for GUI components functionality)

java.io.*:(file programming/streams)

→The aim of this package is to achieve data persistency through the concept of files. Data persistency is nothing but storing the data permanently and in the form of files are in the form of database.

java.applet.*:

→The aim of this package is to develop distributed applications. in the initial dish of sun micro systems sun developers has developed a concept called applets and whose basic aim is to develop distributed applications.

→To fulfill the concept of applets they released a predefined class called applet and it represents in java.applet.*; package. This package contains only one class and whose fully qualified name is java.applet.Applet ;

java.remote.net.*:(Networking programming)

The basic aim of this package is to develop client/server applications. This application development contains two types of programs. they are

1. Client side program
2. Server side program

Client:

A client-side program is a program which is always used for making a request to the server side program for getting the services.

→For developing the client side program we required some predefined classes and interfaces which are present in java.net.*.

Server:

Server side java program is a program which will receive client request and process the client request and gives response back to the client side program.

→For developing the server side program we required some predefined classes and interfaces which are present in java.net.*.

java.text in.*:(Text Processing)

The aim of this package is to format dates, times, numerical roundings etc. In the day report generations.

java.util.*:(collection framework):

The aim of this package is to improve the performance of any Java application.

Collection framework:

It is one of the standardized mechanisms developed by Sun Micro system and supplied as a part of our software and it allows us to group or are gathered different types of values or similar types of values are both the type of values in a single variable with dynamic size. this single variable is known as collection framework variable.

User defined packages:

User defined packages are those which are developed by Java developer and supplied as a part of their project to deal with common requirements.

Syntax for package:

package package name;/package pack 1 [.pack 2….. [. packn]]];

In the above syntax:

- Package is a Keyword used for developing user defined packages

pack 1,Pack 2….. pack n represents Java valid variable names are treated as user defined packages.

- pack 1,Pack 2….. pack n represents name of the sub packages and whose specification is optional, because a package may or may not contain subpackages.
- pack1 represents an upper/outer packages and whose specification is mandatory.

Steps/guidelines for placing a class or interface in the package:

- Choose an appropriate package name for placing common classes and interfaces and ensure that the package statement must first executable statement.
- Choose an appropriate class name/interface name and ensure whose modifier must be public.
- The modifier of the constructor of the class which is present in the package must be public.
- The class/interface which we are placing in the package must be given as a filename with an extension ".java"

Syntax for compiling package classes and interfaces:

javac -d filename.java

Explanation:

- Here -d is an option which gives an indication to Java compiler saying that take the package name and makes the operating system to create it as a directory or folder.
- no errors present in filename.java
- The package name should not be created as a directory full stop the created directory is known as current directory and it is referred by dot(.).filename.java program will be compiled and ensure that filename.class file will be generated " . " reverse current directory.

Example:

```
package myPackage;
public class MyClass
{
   public void getNames(String s)
   {
System.out.println(s);
   }
}
```

Importing Packages and Classes into Programs:

To refer the package class and interface in Java programming we have two approaches they are:

1. by using import statement
2. by using fully qualified name approach

By using import statement:

import is one of the keyword used for referring The Other all the classes and interfaces of a specific package (or) a specific class (or) a specific interface of a specific package in our current Java program.

Syntax:

import pack1[.pack 2[.-------[.packn]]].*;

By using fully qualified name approach:

Syntax:

import pack1[.pack 2[.-------[.packn]]]. class/interface name;

Example:

```
import myPackage.MyClass;
public class PrintName
{
  public static void main(String args[])
  {
    // Initializing the String variable with a value
    String name = "GeeksforGeeks";
    // Creating an instance of class MyClass in the package.
MyClass obj = new MyClass();
obj.getNames(name);
  }
}
```

Access controls:

Access controls are those which are applied before data members and methods of a class. In Java programming we have 4 types of access controls.

They are:

1. Private
2. default(not a keyword)
3. Protected
4. public

Access specifier provides features “accessing control mechanism” among the classes and interfaces.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	YES	NO	NO	NO
Default	YES	YES	NO	NO
Protected	YES	YES	YES	NO
Public	YES	YES	YES	YES

Note:

- private access specifier is also known as a native access specifier
- default access specifier is also known as package access specifier

- protected access specifier is also known as inherited access specifier
- public access specifier is also known as universal access specifier.

Path and class path:

PATH

- The path environment variable is used to specify the set of directories which contains exceptional programs.
- When you try to execute a program from command line, the operating system searches for the specified program in the current directly, if available, executes it.
- In case the programs are not available in the current directory, operating system verifies in the set of directories specified in the 'PATH ' environment variable.

CLASSPATH

- JVM verifies the current directly for them, if not available it verifies the set of directories specified in the 'CLASSPATH' environment variable

Setting CLASSPATH:

- The CLASSPATH is an environment variable that tells the Java compiler where to look for class files to import. CLASSPATH is generally set to a directory or a JAR (Java Archive) file.
- To see what is there in currently in CLASSPATH variable in your system.

  echo %CLASSPATH%
- The class path environment variable is used to specify the location of the classes and packages.
- Suppose, preceding command has displayed class path as:C:\rnr;.
- This means the current class path is set to rnr directory in C: \ and also to the current directory represented by dot (**.**).

Java.lang Package and its Classes:

Provides classes that are fundamental to the design of the Java programming language. The most important classes are Object, which is the root of the class hierarchy, and Class, instances of which represent classes at run time.

Following are the Important Classes in Java.langpackage :

1. **Boolean**: The Boolean class wraps a value of the primitive type boolean in an object.
2. **Byte**: The Byte class wraps a value of primitive type byte in an object.
3. Character – **Set 1**, **Set 2:** The Character class wraps a value of the primitive type char in an object.
4. **Character.Subset**: Instances of this class represent particular subsets of the Unicode character set.
5. **Character.UnicodeBlock**: A family of character subsets representing the character blocks in the Unicode specification.
6. Class – **Set 1**, **Set 2** : Instances of the class Class represent classes and interfaces in a running Java application.
7. ClassLoader: A class loader is an object that is responsible for loading classes.
8. ClassValue: Lazily associate a computed value with (potentially) every type.
9. **Compiler**: The Compiler class is provided to support Java-to-native-code compilers and related services.
10. **Double**: The Double class wraps a value of the primitive type double in an object.
11. **Enum**: This is the common base class of all Java language enumeration types.

12. **Float:** The Float class wraps a value of primitive type float in an object.
13. **InheritableThreadLocal**: This class extends ThreadLocal to provide inheritance of values from parent thread to child thread: when a child thread is created, the child receives initial values for all inheritable thread-local variables for which the parent has values.
14. **Integer** :The Integer class wraps a value of the primitive type int in an object.
15. **Long**: The Long class wraps a value of the primitive type long in an object.

Enumerations:

Enumerations serve the purpose of representing a group of named constants in a programming language.

In Java, we can also add variables, methods and constructors to it. The main objective of enum is to define our own data types(Enumerated Data Types).

Declaration of enum in java :

Enum declaration can be done outside a Class or inside a Class but not inside a Method.

Example:

```
// A Java program to demonstrate that we can have main() inside enum class.
        enum Color
        {
        RED, GREEN, BLUE;
         // Driver method
        public static void main(String[] args)
        {
        Color c1 = Color.RED;
        System.out.println(c1);
        }
        }
```

class Math:

Java Math class provides several methods to work on math calculations like min(), max(), avg(), sin(), cos(), tan(), round(), ceil(), floor(), abs() etc.

If the size is int or long and the results overflow the range of value, the methods addExact(), subtractExact(), multiplyExact(), and toIntExact() throw an ArithmeticException.

```
public class JavaMathExample1
{
   public static void main(String[] args)
   {
      double x = 28;
      double y = 4;
       // return the maximum of two numbers
      System.out.println("Maximum number of x and y is: " +Math.max(x, y));
      // return the square root of y
      System.out.println("Square root of y is: " + Math.sqrt(y));
     //returns 28 power of 4 i.e. 28*28*28*28
```

```
        System.out.println("Power of x and y is: " + Math.pow(x, y));
        // return the logarithm of given value
        System.out.println("Logarithm of x is: " + Math.log(x));
        System.out.println("Logarithm of y is: " + Math.log(y));
        // return the logarithm of given value when base is 10
        System.out.println("log10 of x is: " + Math.log10(x));
        System.out.println("log10 of y is: " + Math.log10(y));
        // return the log of x + 1
        System.out.println("log1p of x is: " +Math.log1p(x));
        // return a power of 2
        System.out.println("exp of a is: " +Math.exp(x));
        // return (a power of 2)-1
        System.out.println("expm1 of a is: " +Math.expm1(x));
    }
}
```