UNIT I

1. **JVM**
2. **JAVA FEATURES** **12 MARKS**
3. **DATA TYPES**
4. **TYPE CONVERSION, CASTING**
5. **CONDITIONAL STATEMENTS** **12 MARKS**
6. **LOOPS** **12 MARKS**
7. **CLASSES, OBJECTS** **12 MARKS**
8. **METHOD DECLARATION AND INVOCATION**
9. **METHOD OVERLOADING** **12 MARKS**

1. JVM (JAVA VIRTUAL MACHINE) ARCHITECTURE

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed. JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

What is JVM

It is:

1. **A specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Oracle and other companies.
2. **An implementation** Its implementation is known as JRE (Java Runtime Environment).
3. **Runtime Instance** Whenever we write java command on the command prompt to run the java class, an instance of JVM is created.

What it does

The JVM performs following operation:

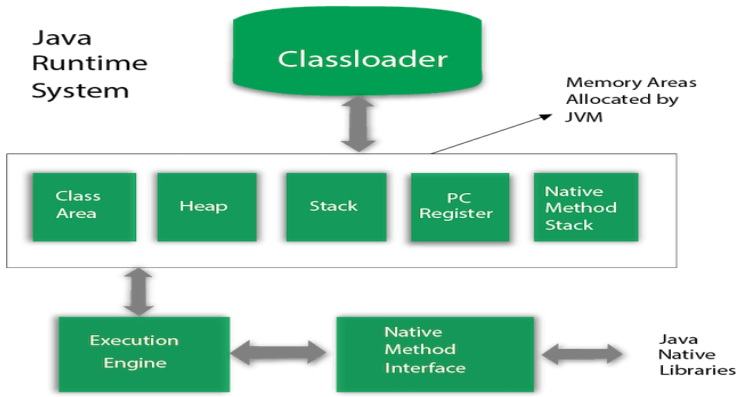
- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.

JVM Architecture

Let's understand the internal architecture of JVM. It contains classloader, memory area, execution engine etc.



2. FEATURES OF JAVA

The primary objective of Java programming language creation was to make it portable, simple and secure programming language. The features of Java are also known as java *buzzwords*.

A list of most important features of Java language is given below.

i. **Simple**
ii. **Object-Oriented**
iii. **Platform independent**
iv. **Secured**
v. **Robust**
vi. **Architecture neutral**
vii. **portable**
viii. **Compiled and Interpreted**
ix. **High Performance**
x. **Multithreaded**
xi. **Distributed**
xii. **Dynamic**

i. Simple

Java language is a simple programming language because:

- Java has removed many explicit pointers, operator overloading, etc.
- Java syntax is based on C++
- Automatic Garbage Collection in Java.

ii. Object-oriented

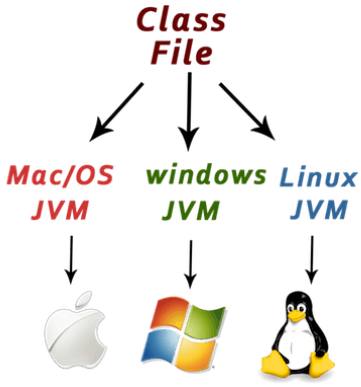
Java is an object-oriented programming language.

Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

Basic concepts of OOPs are:

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

iii. Platform Independent



Java is is different from other languages like C, C++, etc. which are compiled into platform specific machines while Java is a ***write once, run anywhere language***. A platform is the hardware or software environment in which a program runs.

There are two types of platforms software-based and hardware-based. Java provides a software-based platform.

1. Runtime Environment
2. API(Application Programming Interface)

iv. Secured

Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

- **No explicit pointer**
- **Java Programs run inside a virtual machine sandbox**

How java is secured

- **Classloader:** Classloader in Java is a part of the Java Runtime Environment(JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access right to objects.
- **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

v. Robust

Robust simply means strong. Java is robust because:

- It uses strong memory management.
- There is a lack of pointers

- Automatic garbage collection in java which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- Exception handling and the type checking mechanism in Java.

vi. Architecture-neutral

Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

vii. Portable

Java is portable because it facilitates we to carry the Java bytecode to any platform. It doesn't require any implementation.

viii. compiled and interpreted

java is both compiled and interpred. Javac and java tools are used to compile and interpret .java file.

ix. High-performance

Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++). Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

x. Distributed

Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

xi. Multi-threaded

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

xii. Dynamic

Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

Java supports dynamic compilation and automatic memory management (garbage collection).

3. DATATYPES IN JAVA

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

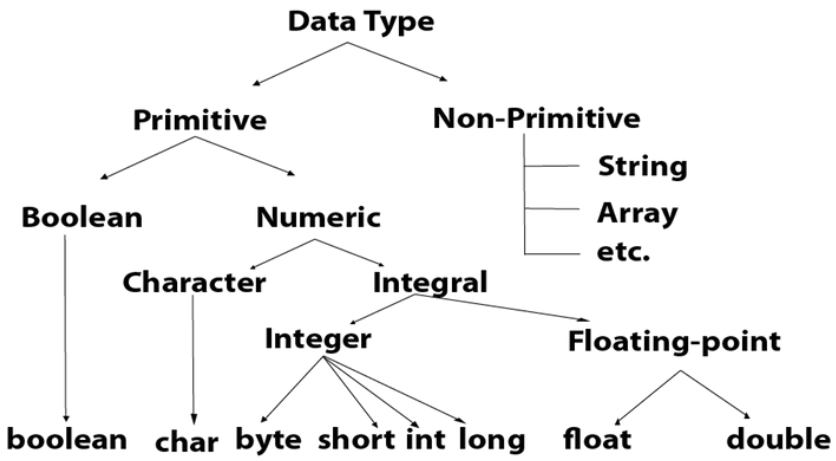
Java Primitive Data Types

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

Java is a statically-typed programming language. It means, all variables must be declared before its use. That is why we need to declare variable's type and name.

There are 8 types of primitive data types:

Datatype	size
boolean data type	1 bit
byte data type	1bytes
char data type	2 bytes
short data type	2 bytes
int data type	4 bytes
long data type	8 bytes
float data type	4 bytes
double data type	8 bytes



4. TYPE CONVERSION AND CASTING

Java will perform the conversion automatically known as Automatic Type Conversion and if not then they need to be casted or converted explicitly.

WIDENING or AUTOMATIC TYPE CONVERSION or TYPE CONVERSION

Automatic conversion takes place when two data types are automatically converted. This happens when:

- The two data types are compatible.
- When we assign value of a smaller data type to a bigger data type.

Widening or Automatic Conversion

Example:

```
int i = 100;

//automatic type conversion
long l = i;
```

NARROWING or EXPLICIT CONVERSION or TYPE CASTING

If we want to assign a value of larger data type to a smaller data type we perform explicit type casting or narrowing.

- This is useful for incompatible data types where automatic conversion cannot be done.
- Here, target-type specifies the desired type to convert the specified value to.

Narrowing or Explicit Conversion

char and number are not compatible with each other.

Example:

```
double d = 100.04;

//explicit type casting
long l = (long)d;

//explicit type casting
int i = (int)l;
```

5. CONDITIONAL STATEMENTS

The Java *if statement* is used to test the condition. It checks boolean condition: *true* or *false*. There are various types of if statement in java.

if statement

if-else statement

if-else-if statements

nested if statement

Java if Statement

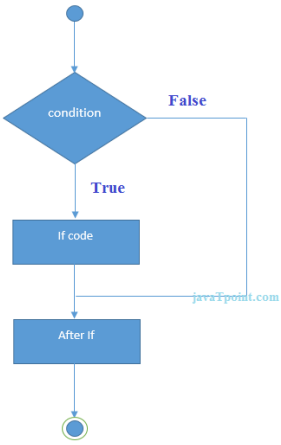
The Java if statement tests the condition. It executes the *if block* if condition is true.

Syntax:

if(condition){

//code to be executed

}



//Java Program to demonstate the use of if statement.

```
public class IfExample
{
    public static void main(String[] args)
    {
            int age=20;
            if(age>18)
            {
                    System.out.print("Age is greater than 18");
            }
    }
}
```

Output:

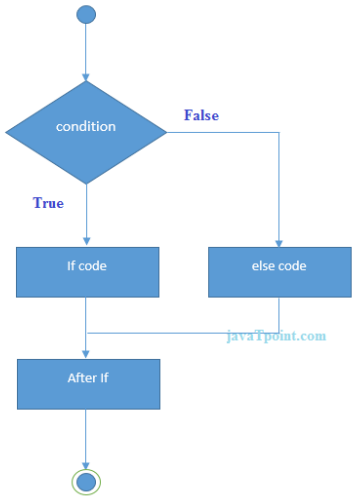
```
Age is greater than 18
```

Java if-else Statement

The Java if-else statement also tests the condition. It executes the *if block* if condition is true otherwise *else block* is executed.

Syntax:

```
if(condition){
//code if condition is true
}else{
//code if condition is false
}
```



Example:

```
//A Java Program to demonstrate the use of if-else statement.
public class IfElseExample
{
    public static void main(String[] args)
    {
            int number=13;
            if(number%2==0)
                    System.out.println("even number");
            else
                    System.out.println("odd number");
    }
}
```

Output:

```
odd number
```

Java if-else-if ladder Statement

The if-else-if ladder statement executes one condition from multiple statements.

Syntax:

```
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
...
else{
//code to be executed if all the conditions are false
}
```

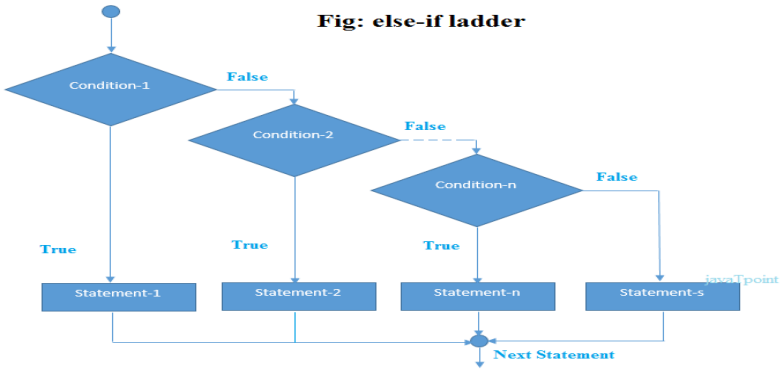


Fig: else-if ladder

Example:

```
//Java Program to demonstrate the use of If else-if ladder.
public class IfElseIfExample
{
    public static void main(String[] args)
    {
            int marks=65;
            if(marks<50)
                System.out.println("fail");
            else if(marks>=50 && marks<60)
                System.out.println("D grade");
            else if(marks>=60 && marks<70)
                System.out.println("C grade");
            else if(marks>=70 && marks<80)
                System.out.println("B grade");
           else if(marks>=80 && marks<90)
                System.out.println("A grade");
            else if(marks>=90 && marks<100)
                System.out.println("A+ grade");
            else
                System.out.println("Invalid!");
    }
}
```

Output: C grade

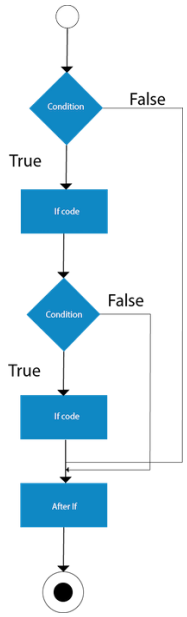
Java Nested if statement

The nested if statement represents the *if block within another if block*. Here, the inner if block condition executes only when outer if block condition is true.

Syntax:

```
if(condition){
    //code to be executed
        if(condition){
          //code to be executed

    }
}
```



//Java Program to demonstrate the use of Nested If Statement.

```
public class JavaNestedIfExample
{
        public static void main(String[] args)
        {
                int age=20;
                int weight=80;
                if(age>=18)
                if(weight>50)
                System.out.println("We are eligible to donate blood");
        }
}
```

Output: We are eligible to donate blood

Switch Statement

The Java *switch statement* executes one statement from multiple conditions. It is like if-else-if ladder statement.

In other words, the switch statement tests the equality of a variable against multiple values.

Syntax:

```
switch(expression)
{
    case value1:
            //code to be executed;
     break;
     case value2:
            //code to be executed;
     break;  .....
    default:
            code to be executed if all cases are not matched;
}
```

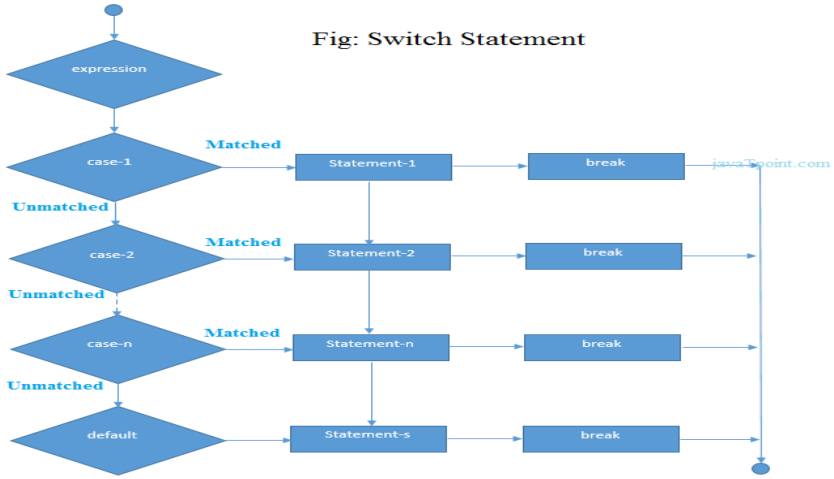


Fig: Switch Statement

Example:

```
public class SwitchExample
{
    public static void main(String[] args)
    {
            int number=20;
            switch(number)
            {
                    case 10: System.out.println("10");
                    break;
                    case 20: System.out.println("20");
                    break;
                    case 30: System.out.println("30");
                    break;
                    default:System.out.println("Not in 10, 20 or 30");
            }
    }
}
```

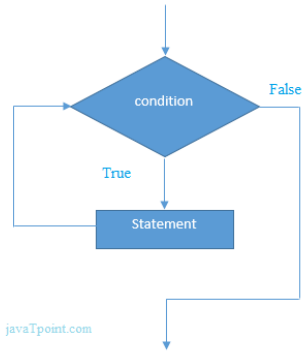
6. LOOPS OR CONTROL STATEMENTS

Java While Loop

The Java *while loop* is used to iterate a part of the program several times. As long as the while loop condition is true it executes the statement as and when the condition becomes false the control comes out from loop to next statement.

Syntax:

```
while(condition){
//code to be executed
}
```



Example:

```
public class WhileExample
{
    public static void main(String[] args)
    {
            int i=1;
            while(i<=10)
            {
                    System.out.println(i);
                    i++;
            }
    }
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

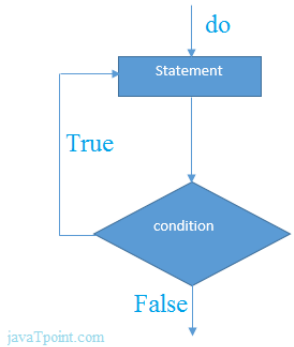
Java do-while Loop

The Java *do-while loop* is used to iterate a part of the program several times. The statement part of the loop is executed at least once,

The Java *do-while loop* is executed at least once because condition is checked after loop body.

Syntax:

```
do{
//code to be executed
}while(condition);
```



Example:

```
public class DoWhileExample
{
    public static void main(String[] args)
    {
            int i=1;
            do
            {
                    System.out.println(i);
                    i++;
            }while(i<=10);
    }
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

For Loop

A simple for loop can initialize the variable, check condition and increment/decrement value. It consists of four parts:

Initialization: it is executed once when the loop starts. we can initialize the variable,

Condition: It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either true or false. It is an optional condition.

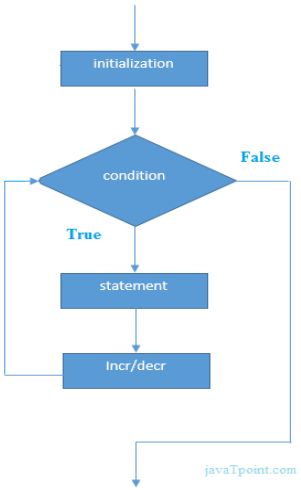
Statement: The statement of the loop is executed each time until the second condition is false.

Increment/Decrement: It increments or decrements the variable value to control entry exit from / to from loop

Syntax:

```
for(initialization;condition;incr/decr){
//statement or code to be executed
}
```

Flowchart:



Example:

```
public class ForExample {
public static void main(String[] args)
 {
   for(int i=1;i<=10;i++){
      System.out.println(i);
   }
}
}
```

7. OBJECT AND CLASS

OBJECT

An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

State: represents the data (value) of an object.

Behavior: represents the behavior (functionality) of an object such as deposit, withdraw, etc.

Identity: An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

Object Definitions:

An object is *a real-world entity*.

An object is *a runtime entity*.

The object is *an entity which has state and behavior*.

The object is *an instance of a class*.

CLASS

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

i. Fields
ii. Methods
iii. Constructors
iv. Blocks
v. Nested class and interface

Syntax to declare a class:

```
class <class_name>{
   field;
   method;
}
```

The following program demonstrates creating class and its object.

```
class Student
{
    int id;
    String name;
}
```

```
//Creating another class TestStudent1 which contains the main method
class TestStudent1
{
    public static void main(String args[])
    {
            Student s1=new Student();
            System.out.println(s1.id);
            System.out.println(s1.name);
    }
}
```

Output: 0

Null

8. METHOD DECLARATION AND INVOCATION

Defining Methods

Syntax

```
modifier returnType nameOfMethod (Parameter List) {
  // method body
}
```

The syntax shown above includes −

- **modifier** − It defines the access type of the method and it is optional to use.
- **returnType** − Method may return a value.
- **nameOfMethod** − This is the method name. The method signature consists of the method name and the parameter list.
- **Parameter List** − The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
- **method body** − The method body defines what the method does with the statements.

Example

Here is the source code of the above defined method called **min()**. This method takes two parameters num1 and num2 and returns the maximum between the two −

```
public static int minFunction(int n1, int n2)
{
        int min;
        if (n1 > n2)
                min = n2;
        else
                min = n1;
        return min;
}
```

Method Calling

For using a method, it should be called. There are two ways in which a method is called i.e., method returns a value or returning nothing (no return value).

The process of method calling is simple. When a program invokes a method, the program control gets transferred to the called method. This called method then returns control to the caller in two conditions, when −

- the return statement is executed.
- it reaches the method ending closing brace.

The methods returning void is considered as call to a statement. Lets consider an example −

```
System.out.println("This is tutorialspoint.com!");
```

The method returning value can be understood by the following example −

```
int result = sum(6, 9);
```

Following is the example to demonstrate how to define a method and how to call it −

Example

```
public class ExampleMinNumber
{
  public static void main(String[] args)
  {
            int a = 11;
            int b = 6;
            int c = minFunction(a, b);
            System.out.println("Minimum Value = " + c);
  }

  /** returns the minimum of two numbers */
  public static int minFunction(int n1, int n2)
  {
            int min;
            if (n1 > n2)
                        min = n2;
            else
                        min = n1;
            return min;
  }
}
```

This will produce the following result −

Output

```
Minimum value = 6
```

The void Keyword

The void keyword allows us to create methods which do not return a value..

Example

```
public class ExampleVoid
{
        public static void main(String[] args)
        {
                methodRankPoints(255.7);
        }
        public static void methodRankPoints(double points)
        {
                if (points >= 202.5)
                {
                        System.out.println("Rank:A1");
                }else if (points >= 122.4)
                {
                        System.out.println("Rank:A2");
                }else
                {
                        System.out.println("Rank:A3");
                }
         }
}
```

This will produce the following result −

Output Rank:A1

9. METHOD OVERLOADING

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose we have to perform addition of the given numbers but there can be any number of arguments, if we write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for we as well as other programmers to understand the behavior of the method because its name differs.

Advantage of method overloading

Method overloading *increases the readability of the program*.

There are two ways to overload the method in java

i. By changing number of arguments

ii. By changing the data type

In java method overloading is not possible by changing the return type only.

i. Method Overloading: changing no. of arguments

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers. We are creating static methods so that we don't need to create instance for calling methods.

```
class Adder
{
	static int add(int a,int b)
	{
		return a+b;
	}
	static int add(int a,int b,int c)
	{
		return a+b+c;
	}
}
class TestOverloading1
{
	public static void main(String[] args)
	{
		System.out.println(Adder.add(11,11));
		System.out.println(Adder.add(11,11,11));
	}
}
```

Output: 22
33

ii. Method Overloading: changing data type of arguments

In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
class Adder
{
	static int add(int a, int b)
	{
		return a+b;
	}
	static double add(double a, double b)
	{
		return a+b;
	}
}
class TestOverloading2
{
	public static void main(String[] args)
	{
		System.out.println(Adder.add(11,11));
		System.out.println(Adder.add(12.3,12.6));
	}
}
```

Output: 22 24.9

UNIT II

1. **CONSTRUCTORS** **12 marks**
    a. **Introduction**
    b. **Types of constructors**
    c. **Constructor overloading,**
2. **CLEANING-UP UNUSEDOBJECTS.**
3. **CLASS VARIABLES (OR) STATIC VARIABLES (OR) STATIC KEY WORD**
4. **THIS KEYWORD**
5. **ARAAYS** **12 marks**
6. **COMMAND-LINE ARGUMENTS** **12 marks**
7. **INNER CLASS**
8. **INHERITANCE** **12 marks**
    a. **Introduction**
    b. **types of inheritance,**
9. **METHOD OVERRIDING**
10. **SUPER**
11. **FINAL KEYWORD**
12. **ABSTRACT CLASSES** **12 marks**
13. **INTERFACES** **12 marks**
14. **ABSTRACT CLASSES VERSES INTERFACES.** **12 marks**
15. **CREATING AND USING PACKAGES** **12 marks**
16. **ACCESS PROTECTION** **12 marks**
17. **WRAPPER CLASSES**
18. **STRING CLASS**
19. **STRING BUFFER CLASS.**

1. CONSTRUCTORS

Constructors are used to initialize the object's state. Like methods, a constructor also contains **collection of statements(i.e. instructions)** that are executed at time of Object creation.

When is a Constructor called ?

Each time an object is created using **new()** keyword at least one constructor (it could be default constructor) is invoked to assign initial values to the **data members** of the same class. Constructor is invoked at the time of object or instance creation. For Example:

```
class Geek
{
  .......

  // A Constructor
  new Geek() {}

  .......
}

// We can create an object of above class
// using below statement. This statement
// calls above constructor.
Geek obj = new Geek();
```

Rules for writing Constructor:

- Constructor(s) of a class must has same name as the class name in which it resides.
- A constructor in Java can not be abstract, final, static and Synchronized.
- Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.

Types of Constructors

There are two type of constructor in Java:

1. **No-argument constructor:** A constructor that has no parameter is known as default constructor. If we don't define a constructor in a class, then compiler creates **default constructor(with no arguments)** for the class. And if we write a constructor with arguments or no-argument then compiler does not create default constructor. Default constructor provides the default values to the object like 0, null etc. depending on the type.

```
class Bike1
{
    Bike1()
```

```
    {
        System.out.println("Bike is created");
    }
    public static void main(String args[])
    {
        Bike1 b=new Bike1();
    }
}
```

Outpt : Bike is created

2. Parameterized Constructor: A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with wer own values, then use parameterized constructor.

```
class Student4
{
    int id;
    String name;

    Student4(int i,String n)
    {
        id = i;
        name = n;
    }
    void display()
    {
        System.out.println(id+" "+name);
    }

    public static void main(String args[])
    {
        Student4 s1 = new Student4(111,"Karan");
        Student4 s2 = new Student4(222,"Aryan");
        s1.display();
        s2.display();
    }
}
```

Output :

```
111 Karan
222 Aryan
```

Constructor overloading

Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

Example of Constructor Overloading

//Java program to overload constructors in java

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

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

Output :

```
111 Karan  0
222 Aryan 25
```

2. CLEANING-UP UNUSEDOBJECTS OR GARBAGE COLLECTION

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

In java it unused objects are clean up automatically. So, java provides better memory management.

Advantage of Garbage Collection

It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.

It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

How can an object be unreferenced?

There are many ways:

By nulling the reference
By assigning a reference to another
By anonymous object etc.

By nulling a reference:

```
Employee e=new Employee();
e=null;
```

By assigning a reference to another:

```
Employee e1=new Employee();
Employee e2=new Employee();
e1=e2;//now the first object referred by e1 is available for garbage collection
```

By anonymous object: **new Employee();**

finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

protected void finalize(){}

Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if we have created any object without new, we can use finalize method to perform cleanup processing (destroying remaining objects).

gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes. **public static void** gc(){}

Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

Simple Example of garbage collection in java

```
public class TestGarbage1
{
    public void finalize(){ System.out.println("object is garbage collected"); }
    public static void main(String args[])
    {
        TestGarbage1 s1=new TestGarbage1();
        TestGarbage1 s2=new TestGarbage1();
        s1=null;
        s2=null;
        System.gc();
    }
}
```

Output : object is garbage collected
object is garbage collected

3. CLASS VARIABLES (or) STATIC VARIABLES (OR) STATIC KEYWORD

The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.

If we declare any variable with static keyword, it is known as a static variable.

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

Understanding the problem without static variable

```
class Student
{
   int rollno;
   String name;
   String college="ITS";
}
```

Static variables are known as one for entire class objects and instance variables (non static variables are known as one for each object.

> *Java static property is shared to all objects.*

Example of static variable

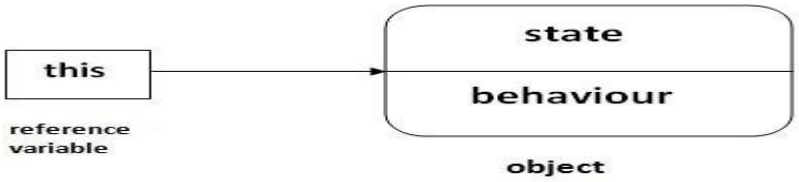
```
class Student
{
        int rollno;//instance variable
        String name;
        static String college ="ITS";//static variable
        Student(int r, String n)                    //constructor
        {
          rollno = r;
          name = n;
        }
        void display ()
        {
                System.out.println(rollno+" "+name+" "+college);
        }
}
public class TestStaticVariable1
{
        public static void main(String args[])
        {
                Student s1 = new Student(111,"Karan");
                Student s2 = new Student(222,"Aryan");
                 s1.display();
                s2.display();
        }
}
```

Output:

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

4. THIS KEYWORD

In java, this is a **reference variable** that refers to the current object.



Usage of java this keyword

i. this can be used to refer current class instance variable.
ii. this can be used to invoke current class method (implicitly)
iii. this() can be used to invoke current class constructor.
iv. this can be passed as an argument in the method call.
v. this can be passed as argument in the constructor call.
vi. this can be used to return the current class instance from the method.

this: to refer current class instance variable

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

Example :

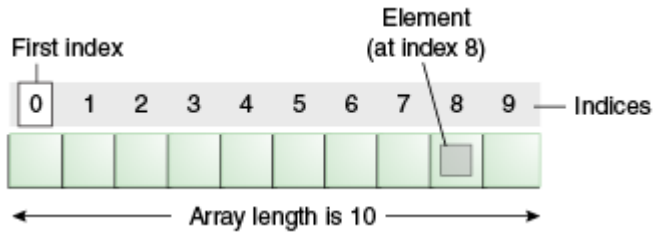
```
class Student
{
    int rollno;
    String name;
    float fee;
    Student(int rollno,String name,float fee)
    {
        this.rollno=rollno;
        this.name=name;
        this.fee=fee;
    }
    void display()
    {
        System.out.println(rollno+" "+name+" "+fee);
    }
}
class TestThis2
{
    public static void main(String args[])
    {
        Student s1=new Student(111,"ankit",5000f);
        Student s2=new Student(112,"sumit",6000f);
        s1.display();  s2.display();
    }
}
```

Output:

```
111 ankit 5000
112 sumit 6000
```

5. ARAAYS

Java array is an object which contains elements of a similar data type. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array. Array in java is index-based, the first element of the array is stored at the 0 index.



Advantages

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

Disadvantages

- **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in Java

Syntax to Declare an Array in Java

dataType[] arr; (or)

dataType []arr; (or)

dataType arr[];

Instantiation of an Array in Java

arrayRefVar=**new** datatype[size];

Example of Java Array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

//Java Program to illustrate how to declare, instantiate, initialize and traverse the Java array.

```
class Testarray
{
    public static void main(String args[])
    {
        int a[]=new int[5];//declaration and instantiation
        a[0]=10;//initialization
        a[1]=20;
        a[2]=70;
        a[3]=40;
        a[4]=50;
    //traversing array
        for(int i=0;i<a.length;i++)//length is the property of array
                System.out.println(a[i]);
    }
}
```

Output:

```
10
20
70
40
50
```

6. COMMAND-LINE ARGUMENTS

- The java command-line argument is an argument i.e. passed at the time of running the java program.
- The arguments passed from the console can be received in the java program and it can be used as an input.
- So, it provides a convenient way to check the behavior of the program for the different values. We can pass **N** (1,2,3 and so on) numbers of arguments from the command prompt.

Simple example of command-line argument in java

In this example, we are receiving only one argument and printing it. To run this java program, we must pass at least one argument from the command prompt.

```
class CommandLineExample
{
        public static void main(String args[])
        {
                System.out.println("Wer first argument is: "+args[0]);
        }
}
```

compile by > javac CommandLineExample.java

run by > java CommandLineExample sonoo

Output: Wer first argument is: sonoo

Example of command-line argument that prints all the values

In this example, we are printing all the arguments passed from the command-line. For this purpose, we have traversed the array using for loop.

```
class A
```

```
{
        public static void main(String args[])
        {
                for(int i=0;i<args.length;i++)
                System.out.println(args[i]);
        }
}
```
compile by > javac A.java
run by > java A sonoo jaiswal 1 3 abc
Output: sonoo
jaiswal
1
3
Abc

7. INNER CLASS

Java inner class or nested class is a class which is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

Additionally, it can access all the members of outer class including private data members and methods.

Syntax of Inner class

```
class Java_Outer_class
{
        //code
        class Java_Inner_class
        {
        //code
        }
}
```

Advantage of java inner classes

There are basically three advantages of inner classes in java. They are as follows:

Nested classes represent a special type of relationship that is **it can access all the members (data members and methods) of outer class** including private.

Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.

Code Optimization: It requires less code to write.

8. INHERITANCE

Introduction

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviours of a parent object. It is an important part of OOPs The idea behind inheritance in Java is that new classes are created that are built upon existing classes. When a class inherits from an existing class, inheriting class can reuse methods and fields of the parent class. And it can also add new methods and fields.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Why use inheritance in java

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

Terms used in Inheritance

Class: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.

Sub Class/Child Class: Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.

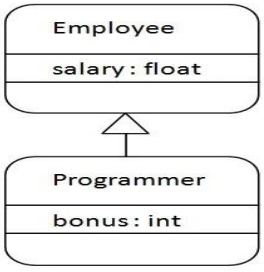
Super Class/Parent Class: Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

Reusability: reusability is a mechanism which facilitates we to reuse the fields and methods of the existing class when we create a new class. We can use the same fields and methods already defined in the previous class.

Syntax:

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

- The **extends keyword** indicates that we are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.
- In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

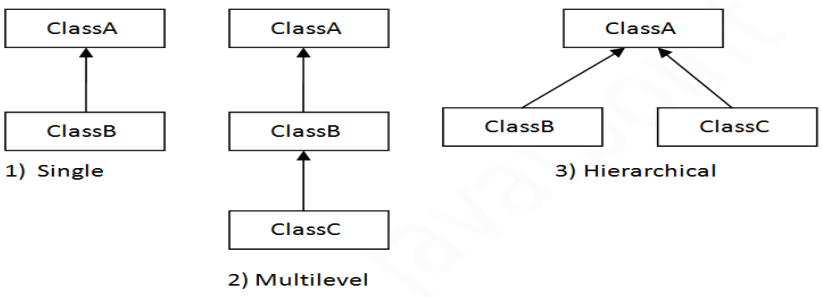
```
class Employee
{
        float salary=40000;
}
class Programmer extends Employee
{
                 int bonus=10000;
                public static void main(String args[])
                {
                        Programmer p=new Programmer();
                        System.out.println("Programmer salary is:"+p.salary);
                        System.out.println("Bonus of Programmer is:"+p.bonus);
                }
}
```

```
Programmer salary is:40000.0
 Bonus of programmer is:10000
```

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

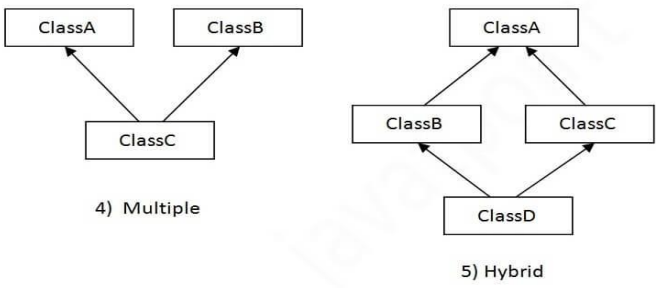
Types of Inheritance

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



Note : Multiple Inheritance is not supported in Java through class

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



Single Inheritance Example

```
class Animal
{
    void eat(){System.out.println("eating...");}
}
class Dog extends Animal
{
    void bark(){System.out.println("barking...");}
}
class TestInheritance
{
    public static void main(String args[])
    {
            Dog d=new Dog();
            d.bark();
            d.eat();
    }
}
```

Output: barking...
eating...

Multilevel Inheritance Example

```
class Animal
{
    void eat()
    {
        System.out.println("eating...");
    }
}
class Dog extends Animal
{
    void bark()
    {
        System.out.println("barking...");
    }
}
class BabyDog extends Dog
{
    void weep()
    {
        System.out.println("weeping...");
    }
}
class TestInheritance2
{
    public static void main(String args[])
    {
        BabyDog d=new BabyDog();
        d.weep();  d.bark();    d.eat();
    }
}
```

Output: weeping...
barking...
eating...

Hierarchical Inheritance Example

```
class Animal
{
    void eat(){System.out.println("eating...");}
}
class Dog extends Animal
{
    void bark(){System.out.println("barking...");}
}
class Cat extends Animal
{
    void meow(){System.out.println("meowing...");}
}

class TestInheritance3
{
    public static void main(String args[])
    {
            Cat c=new Cat();
            c.meow();
            c.eat();
            //c.bark();//C.T.Error
    }
}
```

Output: meowing...
eating...

Q) Why multiple inheritance is not supported in java?

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

```
class A
{
    void msg(){System.out.println("Hello");}
}
class B
{
    void msg(){System.out.println("Welcome");}
}
class C extends A,B
{
    public Static void main(String args[])
    {
            C obj=new C();
            obj.msg();//Now which msg() method would be invoked?
    }
}
```

Compile Time Error

9. METHOD OVERRIDING

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**. In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

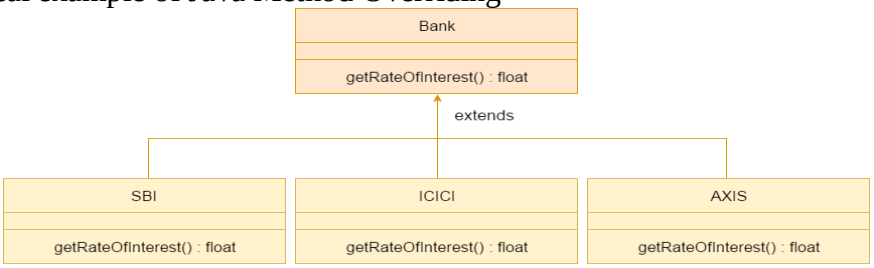
Usage of Java Method Overriding

Method overriding is used to provide the specific implementation of a method which is already provided by its superclass. Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

- The method must have the same name as in the parent class.
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).

A real example of Java Method Overriding



Java method overriding is mostly used in Runtime Polymorphism

```
//Java Program to demonstrate the real scenario of Java Method Overriding
class Bank
{
    int getRateOfInterest()
    {
            return 0;
    }
}
class SBI extends Bank              //Creating child classes.
{
    int getRateOfInterest()
    {
            return 8;
    }
}
class ICICI extends Bank
{
    int getRateOfInterest()
    {
            return 7;
    }
}
```

```
class AXIS extends Bank
{
    int getRateOfInterest()
    {
            return 9;
    }
}
class Test2                 //Test class to create objects and call the methods
{
    public static void main(String args[])
    {
            SBI s=new SBI();
            ICICI i=new ICICI();
            AXIS a=new AXIS();
            System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
            System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
            System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
    }
}
```

Output:

```
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

10. SUPER KEYWORD

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object. Whenever we create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

1. super can be used to refer immediate parent class instance variable.

   Ex: super.variablename;

2. super can be used to invoke immediate parent class method.

   Ex: super.methodname();

3. super() can be used to invoke immediate parent class constructor.

super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
class Animal
{
    String color="white";
}
```

```
class Dog extends Animal
{
    String color="black";
    void printColor()
    {
            System.out.println(color);//prints color of Dog class
            System.out.println(super.color);//prints color of Animal class
    }
}
class TestSuper1
{
    public static void main(String args[])
    {
            Dog d=new Dog();
            d.printColor();
    }
}
```

Output: black
white

11. FINAL KEYWORD

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

1. variable
2. method
3. class

1) Java final variable :If we make any variable as final, we cannot change the value of final variable (It will be constant).

2) if we make a method as final it cannot override.

3) if a class is declared as final it cannot be sub classed (extended or inherited).

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

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

Output:Compile Time Error

12. ABSTRACT CLASSES

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body). It needs to be extended and its method implemented. It cannot be instantiated.

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Abstraction lets we focus on what the object does instead of how it does it.

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Abstract Method in Java

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

Example of abstract method

```
abstract void printStatus();    //no method body and abstract
```

Example of Abstract class that has an abstract method

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

```
abstract class Bike
{
        abstract void run();
}
class Honda4 extends Bike
{
        void run()
        {
                System.out.println("running safely");
        }
        public static void main(String args[])
        {
                Bike obj = new Honda4();
                obj.run();
        }
}
```

Output : running safely

13. INTERFACES

An **interface in java** is a blueprint of a class. It has static constants and abstract methods. The interface in Java is *a mechanism to achieve abstraction*.

- Java Interface also **represents the IS-A relationship**.
- It cannot be instantiated just like the abstract class.
- Since Java 8, we can have **default and static methods** in an interface.
- Since Java 9, we can have **private methods** in an interface.

Why use Java interface?

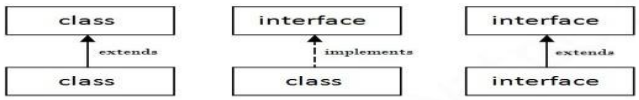
- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

How to declare an 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.
}
```

Example:
```
interface printable
{
        void print();
}
class A6 implements printable
{
        public void print()
        {
                System.out.println("Hello");
        }
        public static void main(String args[])
        {
                A6 obj = new A6();
                obj.print();
        }
}
```



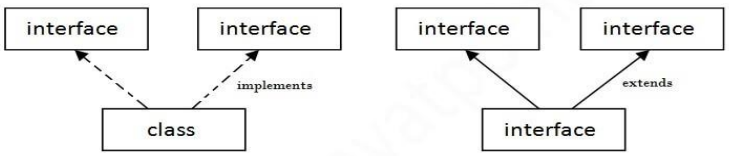
Interface inheritance
A class implements an interface, but one interface extends another interface.

```
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

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

```
interface Printable
{
        void print();
}
interface Showable
{
        void show();
}
```

```
class A7 implements Printable,Showable
{
	public void print()
	{
		System.out.println("Hello");
	}
	public void show()
	{
		System.out.println("Welcome");
	}
	public static void main(String args[])
	{
		A7 obj = new A7();
		obj.print();
		obj.show();
	}
}
```

14. ABSTRACT CLASSES VERSES INTERFACES

Difference between abstract class and interface

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class**can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class**can be extended using keyword "extends".	An **interface class**can be implemented using keyword "implements".
8) A Java**abstract class**can have class members like private, protected, etc.	Members of a Java interface are public by default.

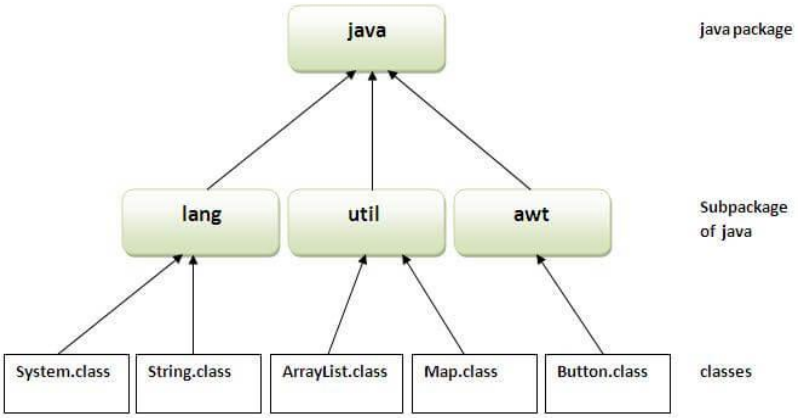
9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

15. CREATING AND USING PACKAGES

A **java package** is a group of similar types of classes, interfaces and sub-packages. Package in java can be categorized in two form, built-in package and user-defined package. There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package

- Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- Java package provides access protection.
- Java package removes naming collision.



Simple example of creating java package

The **package keyword** is used to ***create a package in java.***

//save as Simple.java

```
package mypack;
public class Simple
{
        public static void main(String args[])
        {
                System.out.println("Welcome to package");
        }
}
```

How to compile java package

If we are not using any IDE, we need to follow the **syntax** given below:

javac -d directory javafilename

For **example** javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file. We can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If we want to keep the package within the same directory, we can use . (dot).

How to run java package program

We need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

Output:Welcome to package

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

How to access package from another package?

There are three ways to access the package from outside the package.

import package.*;

import package.classname;

fully qualified name.

*Using packagename.**

If we 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.

Example of package that import the packagename.*

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

```
package mypack;
import pack.*;
 class B
{
        public static void main(String args[])
        {
                A obj = new A();
                obj.msg();
        }
}
```
Output:Hello

Using packagename.classname

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

Example of package by import package.classname

//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

16. ACCESS PROTECTION OR ACCESS SPECIFIERS

There are two types of modifiers in java: **access modifiers** and **non-access modifiers**. The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.

There are 4 types of java access modifiers:

i. private
ii. default
iii. protected
iv. public

There are many non-access modifiers such as static, abstract, synchronized, native, volatile, transient etc.

private access modifier

The private access modifier is accessible only within class.

```
class A
{
        private int data=40;
        private void msg()
        {
                System.out.println("Hello java");
        }
}

public class Simple
{
        public static void main(String args[])
        {
                A obj=new A();
                System.out.println(obj.data);//Compile Time Error
                obj.msg();//Compile Time Error
        }
}
```

Note : A class cannot be private or protected except nested class

default access modifier

If we don't use any modifier, it is treated as **default** bydefault. The default modifier is accessible only within package.

protected access modifier

The **protected access modifier** is accessible within package and outside the package but through inheritance only. The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

public access modifier

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

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

17. WRAPPER CLASSES

Wrapper classes works on primitive data types. Wrapper class in java **provides the mechanism *to convert primitive into object and object into primitive*.**

The list of eight wrapper classes of java.lang package are given below:

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Wrapper class Example: Primitive to Wrapper

```
public class WrapperExample1
{
        public static void main(String args[])
        {
                //Converting int into Integer
                int a=20;
                Integer i=Integer.valueOf(a);//converting int into Integer
                Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
                System.out.println(a+" "+i+" "+j);
        }
}
```

Output: 20 20 20

Wrapper class Example: Wrapper to Primitive

```
public class WrapperExample2
{
        public static void main(String args[])
        {
                //Converting Integer to int
                Integer a=new Integer(3);
                int i=a.intValue();//converting Integer to int
                int j=a;//unboxing, now compiler will write a.intValue() internally
                System.out.println(a+" "+i+" "+j);
        }
}
```

Output: 3 3 3

18. STRING CLASS

In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string. For example:

char[] ch={'j','a','v','a','t','p','o','i','n','t'};

String s=**new** String(ch);

is same as:

String s="lbc";

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

No.	Method	Description
1	char charAt(int index)	returns char value for the particular index
2	int length()	returns string length
3	static String format(String format, Object... args)	returns a formatted string.
4	static String format(Locale l, String format, Object... args)	returns formatted string with given locale.
5	String substring(int beginIndex)	returns substring for given begin index.
6	String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index.
7	boolean contains(CharSequence s)	returns true or false after matching the sequence of char value.
8	static String join(CharSequence delimiter, CharSequence... elements)	returns a joined string.
9	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	returns a joined string.
10	boolean equals(Object another)	checks the equality of string with the given object.
11	boolean isEmpty()	checks if string is empty.

12	String concat(String str)	concatenates the specified string.
13	String replace(char old, char new)	replaces all occurrences of the specified char value.
14	String replace(CharSequence old, CharSequence new)	replaces all occurrences of the specified CharSequence.
15	static String equalsIgnoreCase(String another)	compares another string. It doesn't check case.
16	String[] split(String regex)	returns a split string matching regex.
17	String[] split(String regex, int limit)	returns a split string matching regex and limit.
18	String intern()	returns an interned string.
19	int indexOf(int ch)	returns the specified char value index.
20	int indexOf(int ch, int fromIndex)	returns the specified char value index starting with given index.
21	int indexOf(String substring)	returns the specified substring index.
22	int indexOf(String substring, int fromIndex)	returns the specified substring index starting with given index.
23	String toLowerCase()	returns a string in lowercase.
24	String toLowerCase(Locale l)	returns a string in lowercase using specified locale.
25	String toUpperCase()	returns a string in uppercase.
26	String toUpperCase(Locale l)	returns a string in uppercase using specified locale.
27	String trim()	removes beginning and ending spaces of this string.
28	static String valueOf(int value)	converts given type into string. It is an overloaded method.

19. STRING BUFFER CLASS

Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

> *Note: Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.*

What is mutable string

A string that can be modified or changed is known as mutable string. StringBuffer and StringBuilder classes are used for creating mutable string.

Important Constructors of StringBuffer class

Constructor	Description
StringBuffer()	creates an empty string buffer with the initial capacity of 16.
StringBuffer(String str)	creates a string buffer with the specified string.
StringBuffer(int capacity)	creates an empty string buffer with the specified capacity as length.

Important methods of StringBuffer class

Modifier and Type	Method	Description
public synchronized StringBuffer	append(String s)	appends the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
public synchronized StringBuffer	insert(int offset, String s)	inserts the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public synchronized StringBuffer	replace(int startIndex, int endIndex, String str)	replaces the string from specified startIndex and endIndex.

public synchronized StringBuffer	delete(int startIndex, int endIndex)	deletes the string from specified startIndex and endIndex.
public synchronized StringBuffer	reverse()	reverses the string.
public int	capacity()	returns the current capacity.
public void	ensureCapacity(int minimumCapacity)	ensures the capacity at least equal to the given minimum.
public char	charAt(int index)	returns the character at the specified position.
public int	length()	returns the length of the string i.e. total number of characters.
public String	substring(int beginIndex)	returns the substring from the specified beginIndex.
public String	substring(int beginIndex, int endIndex)	returns the substring from the specified beginIndex and endIndex.

UNIT III

1. **EXCEPTION HANDLING : INTRODUCTION AND ITS TYPES**
2. **EXCEPTION HANDLING TECHNIQUES 12 marks**
3. **USER-DEFINED EXCEPTION.**
4. **MULTITHREADING: INTRODUCTION MAIN THREAD**
5. **CREATION OF NEW THREADS 12 marks**
    i. **INHERITING THE THREAD CLASS**

    OR EXTENDING THE THREAD CLASS

    ii. **IMPLEMENTING THE RUNNABLE INTERFACE**
6. **THREAD LIFECYCLE 12 marks**
7. **THREAD PRIORITY**
8. **THREAD SYNCHRONIZATION**
9. **INPUT/OUTPUT: INTRODUCTION**
10. **java.io PACKAGE**
11. **File CLASS**
12. **FileInputStream CLASS 12 marks**
13. **FileOutputStream CLASS 12 marks**
14. **Scanner CLASS**
15. **BufferedInputStream CLASS 12 marks**
16. **BufferedOutputStream CLASS 12 marks**
17. **RandomAccessFile CLASS. 12 marks**

1. EXCEPTION HANDLING : INTRODUCTION AND ITS TYPES

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

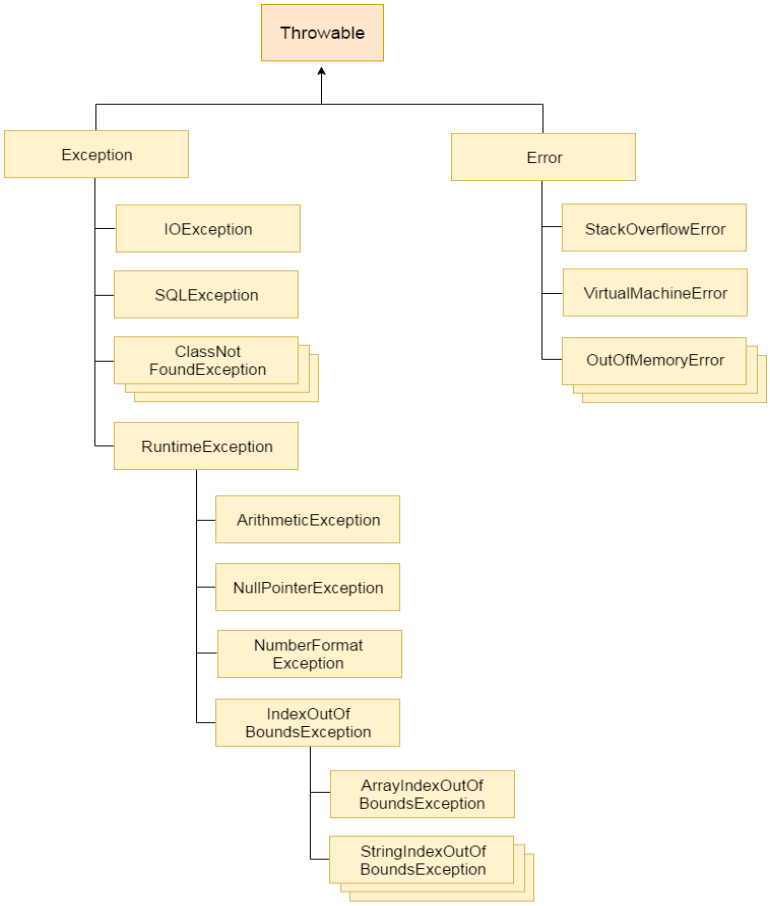
The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained. Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application that is why we use exception handling.

Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses: Exception and Error. A hierarchy of Java Exception classes are given below:



Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between Checked and Unchecked Exceptions

1) Checked Exception

The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Common Scenarios of Java Exceptions (or) Examples of exceptions

A scenario where ArithmeticException occurs

```
int a=50/0;//ArithmeticException
```

A scenario where NullPointerException occurs

```
String s=null;
System.out.println(s.length());//NullPointerException
```

A scenario where NumberFormatException occurs

```
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
```

A scenario where ArrayIndexOutOfBoundsException occurs

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsExceptio
```

2. EXCEPTION HANDLING TECHNIQUES

i. try catch Block

try block : Java try block is used to enclose the code that might throw an exception. It must be used within the method. Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

try{

//code that may throw exception

}**catch**(Exception_class_Name ref){}

Syntax of try-finally block

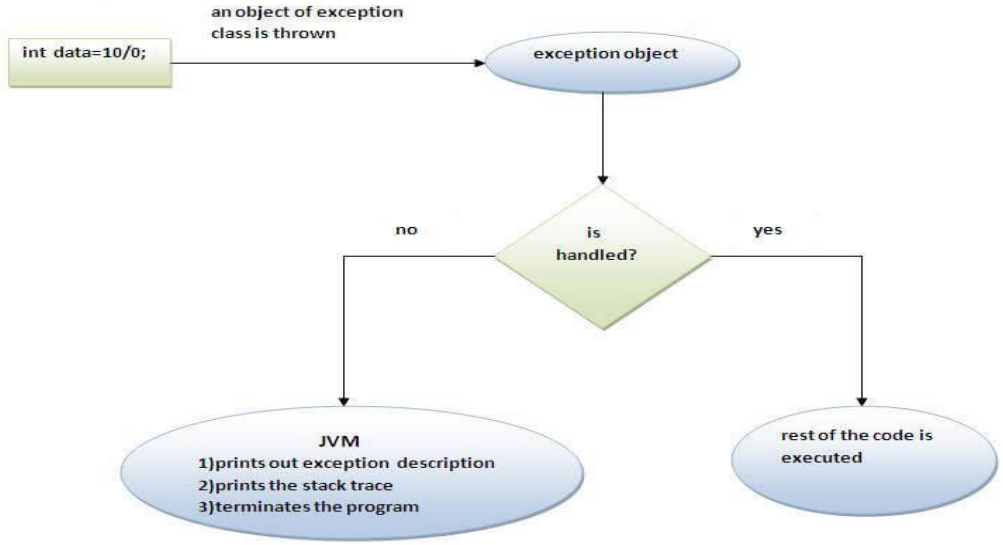
try{

//code that may throw exception

}**finally**{}

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

Internal working of java try-catch block



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.

But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

Demonstration of java try-catch block.

```
public class Testtrycatch2
{
        public static void main(String args[])
        {
                try
                {
                        int data=50/0;
                }
                catch(ArithmeticException e)
                {
                        System.out.println(e);
                }
                System.out.println("rest of the code...");
        }
}
```

Output: Exception in thread main java.lang.ArithmeticException:/ by zero

rest of the code...

Now, as displayed in the above example, rest of the code is executed i.e. rest of the code... statement is printed.

Java Multi catch block

If we have to perform different tasks at the occurrence of different Exceptions, use java multi catch block. simple example of java multi-catch block.

```
public class TestMultipleCatchBlock
{
        public static void main(String args[])
        {
                try
                {
                        int a[]=new int[5];
                        a[5]=30/0;
                }
                catch(ArithmeticException e){System.out.println("task1 is completed");}
                catch(ArrayIndexOutOfBoundsException e)
                        {System.out.println("task 2 completed");}
                catch(Exception e){System.out.println("common task completed");}
                 System.out.println("rest of the code...");
        }
}
```

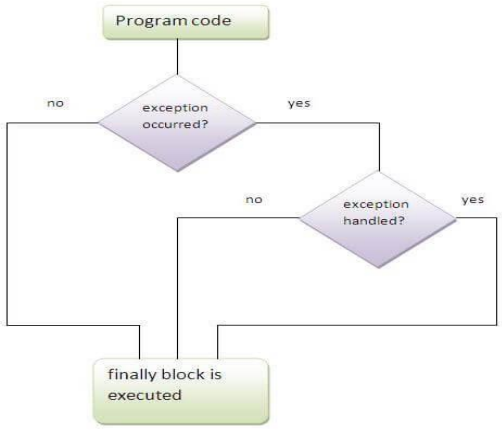
Output: task1 completed
rest of the code...

Rule: At a time only one Exception is occured and at a time only one catch block is executed.

Rule: All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception .

ii. 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.



Note: If we don't handle exception, before terminating the program, JVM executes finally block(if any).

Why use java finally

Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

Case 1 **Let's see the java finally example where** exception doesn't occur**.**

```
class TestFinallyBlock
{
    public static void main(String args[])
    {
        try
        {
            int data=25/5;
            System.out.println(data);
        }
        catch(NullPointerException e)
        {
            System.out.println(e);
        }
        finally
        {
            System.out.println("finally block is always executed");
        }
        System.out.println("rest of the code...");
    }
}
```

Output:
```
5
finally block is always executed
rest of the code...
```

Case 2 **Let's see the java finally example where** exception occurs and handled**.**

```
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...");
        }
}
```

Output: Exception in thread main java.lang.ArithmeticException:/ by zero
finally block is always executed
rest of the code...

Rule: For each try block there can be zero or more catch blocks, but only one finally block.Note: The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

iii. Java throw keyword

The Java throw keyword is used to explicitly throw an exception. We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

The syntax of java throw keyword is **throw** exception;
Let's see the example of throw IOException.

```
        throw new IOException("sorry device error);
```

java throw keyword example

```
public class TestThrow1
{       static void validate(int age)
        {
                if(age<18)      throw new ArithmeticException("not valid");
                else            System.out.println("welcome to vote");
        }
        public static void main(String args[])
        {
                 validate(13);
                 System.out.println("rest of the code...");
          }
}       Output:         Exception in thread main java.lang.ArithmeticException:not valid
```

Java throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Syntax of java throws

```
return_type method_name() throws exception_class_name{
//method code
}
```

Advantage of Java throws keyword

It provides information to the caller of the method about the exception.

Java throws example

Let's see the example of java throws clause which describes that checked exceptions can be propagated by throws keyword.

```
import java.io.IOException;
class Testthrows1
{
    void m()throws IOException
    {
            throw new IOException("device error");//checked exception
    }
    void n()throws IOException
    {
            m();
    }
    void p()
    {
            try
            {
                    n();
            }
            catch(Exception e)
            {
                System.out.println("exception handled");
            }
      }
    public static void main(String args[])
    {
            Testthrows1 obj=new Testthrows1();
            obj.p();
            System.out.println("normal flow...");
    }
}
```

Output: exception handled
normal flow...

Rule: If we are calling a method that declares an exception, we must either caught or declare the exception.

3. USER-DEFINED EXCEPTION

If we are creating wer own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need.

By the help of custom exception, we can have our own exception and message.

Let's see a simple example of java custom exception.

```
class InvalidAgeException extends Exception
{
        InvalidAgeException(String s)
        {
                super(s);
        }
}
class TestCustomException1
{
        static void validate(int age)throws InvalidAgeException
        {
        if(age<18)
                throw new InvalidAgeException("not valid");
        else
                System.out.println("welcome to vote");
        }
        public static void main(String args[])
        {
                try
                {
                        validate(13);
                }
                catch(Exception m){System.out.println("Exception occured: "+m);}
                System.out.println("rest of the code...");
  }
}
Output:Exception occured: InvalidAgeException:not valid
       rest of the code...
```

4. MULTITHREADING: INTRODUCTION MAIN THREAD

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

Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

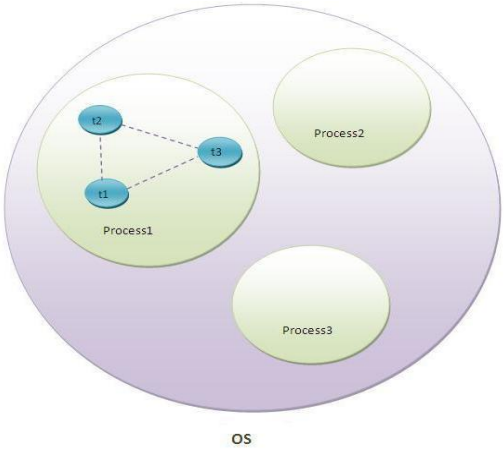
Java Multithreading is mostly used in games, animation etc.

Advantage of Java Multithreading

- It **doesn't block the user** because threads are independent and we can perform multiple operations at same time.
- We **can perform many operations together so it saves time**.
- Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

What is 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.



As shown in the above figure, thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

5. CREATION OF NEW THREADS

i. BY INHERITING THE THREAD CLASS (EXTENDING THREAD CLASS)

ii. IMPLEMENTING THE RUNNABLE INTERFACE

There are two ways to create a thread:

By extending Thread class

By implementing Runnable int

Thread class

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to

be executed by a thread. Runnable interface have only one method named run().

1. **public void run():** is used to perform action for a thread.

Starting a thread:

start() method of Thread class is used to start a newly created thread. It performs following tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

1) **Java Thread Example by extending Thread class**

```
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) **Java Thread Example by implementing Runnable interface**

```
class Multi3 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...

6. THREAD LIFECYCLE

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state. But for better understanding the threads, we are explaining it in the 5 states.

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

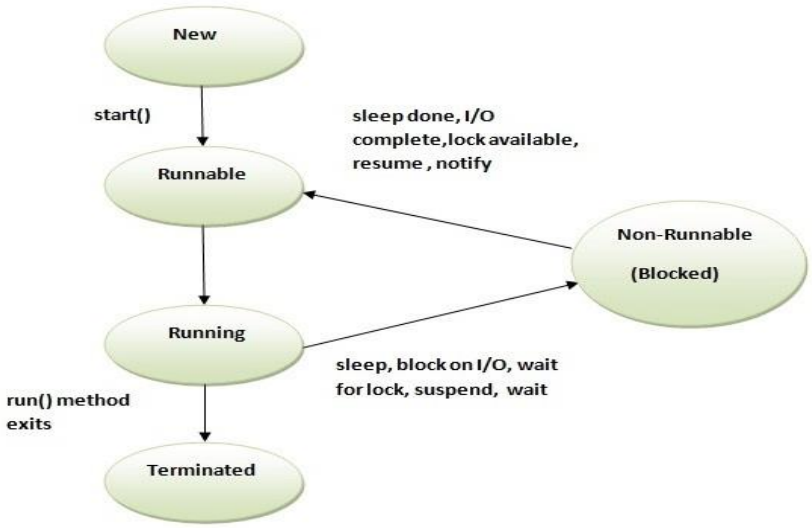
New

Runnable

Running

Non-Runnable (Blocked)

Terminated



1)New

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

2) Runnable

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

3) Running

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

4) Non-Runnable (Blocked)

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

5) Terminated

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

7. 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 reemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defiend in Thread class:

public static int MIN_PRIORITY

public static int NORM_PRIORITY

public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example of priority of a Thread:

```
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();

    }
}
```

```
Output:     running thread name is:Thread-0
            running thread priority is:10m
            running thread name is:Thread-1
            running thread priority is:1
```

8. THREAD SYNCHRONIZATION

Synchronization in java is the capability *to control the access of multiple threads to any shared resource*. Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization

1. To prevent thread interference.
2. To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
   - Synchronized method.
   - Synchronized block.
   - static synchronization.
2. Cooperation (Inter-thread communication in java)

Java synchronized method

- If we declare any method as synchronized, it is known as synchronized method.
- Synchronized method is used to lock an object for any shared resource.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

```
//example of java synchronized method
class Table
{
    synchronized void printTable(int n)
    {//synchronized method
        for(int i=1;i<=5;i++)
        {
            System.out.println(n*i);
            try
            {
                Thread.sleep(400);
            }catch(Exception e){System.out.println(e);}
        }

    }
```

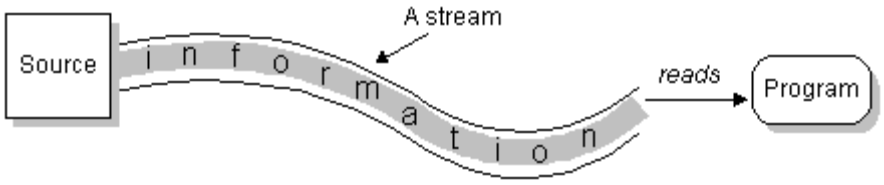
```
}
 class MyThread1 extends Thread
{
	Table t;
	MyThread1(Table t)
	{
		this.t=t;
	}
	public void run()
	{
		t.printTable(5);
	}

}
class MyThread2 extends Thread
{
	Table t;
	MyThread2(Table t)
	{
		this.t=t;
	}
	public void run()
	{
		t.printTable(100);
	}
}
public class TestSynchronization2
{
	public static void main(String args[])
	{
		Table obj = new Table();//only one object
		MyThread1 t1=new MyThread1(obj);
		MyThread2 t2=new MyThread2(obj);
		t1.start();
		t2.start();
	}
}
```

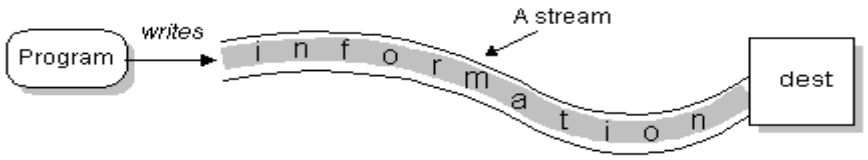
```
Output: 5
10
15
20
25
100
200
300
400
500
```

9. INPUT/OUTPUT: INTRODUCTION

Stream : To bring in information, a program opens a *stream* on an information source (a file, memory, a socket) and reads the information sequentially, as shown here:



Similarly, a program can send information to an external destination by opening a stream to a destination and writing the information out sequentially, like this:



No matter where the data is coming from or going to and no matter what its type, the algorithms for sequentially reading and writing data are basically the same:

Reading	**Writing**
`open a stream`	`open a stream`
`while more information`	`while more information`
`    read information`	`    write information`
`close the stream`	`close the stream`

10. java.io PACKAGE

The java.io package contains a collection of stream classes that support these algorithms for reading and writing. To use these classes, a program needs to import the java.io package. The stream classes are divided into two class hierarchies, based on the data type (either characters or bytes) on which they operate.

Streams are two types they are Byte streams, character streams

Byte Streams

To read and write 8-bit bytes, programs should use the byte streams. Reads from the source into bytes or writes to destinations in bytes. These streams are typically used to read and write binary data such as images and sounds. Two of the byte stream classes

Input Stream and Output Stream are abstract super classes for all byte based streams

The following table lists java.io's streams and describes what they do. Note that many times, java.io contains character streams and byte streams that perform the same type of I/O but for different data types.

I/O Streams		
Type of I/O	**Streams**	**Description**
Memory	CharArrayReader CharArrayWriter ByteArrayInputStream ByteArrawetputStream	Use these streams to read from and write to memory. We create these streams on an existing array and then use the read and write methods to read from or write to the array.
	StringReader StringWriter StringBufferInputStream	Use StringReader to read characters from a String in memory. Use StringWriter to write to a String. StringWriter collects the characters written to it in a StringBuffer, which can then be converted to a String. StringBufferInputStream is similar to StringReader, except that it reads bytes from a StringBuffer.
Pipe	PipedReader PipedWriter PipedInputStream PipedOutputStream	Implement the input and output components of a pipe. Pipes are used to channel the output from one thread into the input of another.
File	FileReader FileWriter FileInputStream FileOutputStream	Collectively called file streams, these streams are used to read from or write to a file on the native file system.
Concatenation	*N/A* SequenceInputStream	Concatenates multiple input streams into one input stream.
Object Serialization	*N/A* ObjectInputStream ObjectOutputStream	Used to serialize objects
Data Conversion	N/A DataInputStream DataOutputStream	Read or write primitive data types in a machine-independent format.
Counting	LineNumberReader LineNumberInputStream	Keeps track of line numbers while reading.
Peeking Ahead	PushbackReader PushbackInputStream	These input streams each have a pushback buffer. When reading data from a stream, it is sometimes useful to peek at the next few bytes or characters in the stream to decide what to do next.

Printing	PrintWriter PrintStream	Contain convenient printing methods. These are the easiest streams to write to, so we will often see other writable streams wrapped in one of these.
Buffering	BufferedReader BufferedWriter BufferedInputStream BufferedOutputStream	Buffer data while reading or writing, thereby reducing the number of accesses required on the original data source. Buffered streams are typically more efficient than similar nonbuffered streams and are often used with other streams.
Filtering	FilterReader FilterWriter FilterInputStream FilterOutputStream	These abstract classes define the interface for filter streams, which filter data as it's being read or written.
Converting between Bytes and Characters	InputStreamReader OutputStreamWriter	A reader and writer pair that forms the bridge between byte streams and character streams.An InputStreamReader reads bytes from an InputStream and converts them to characters, using the default character encoding or a character encoding specified by name.An OutputStreamWriter converts characters to bytes, using the default character encoding or a character encoding specified by name and then writes those bytes to an OutputStream..We can get the name of the default character encoding by calling System.getProperty("file.encoding").

11. File CLASS

The file class is not I/O. It provides just an identifier of files and directories. Files and directories are accessed and manipulated thorugh java.io.file class.

File class following constructors

File MyFile=new File(“c:/java/file_name.txt”);

Gives filename and path as one string parameter.

File MyFile=new File(“c:/java”, file_name.txt”);

One stirng parameter is a path and another one is file name.

File MyFile=new File(“java”, file_name.txt”);

One is sub directory and another one is file name.

File Class has following methods

boolean exists()	:	returns true if file already exists
boolean canWrite()	:	return true if file is writable.
boolean canRead()	:	return true if file is readable
boolean isFile()	:	return true if reference is a file and false for directories references.
Boolean isDirectory	:	return true if reference is a directory.
String getAbsolutePath()	:	return the absolue path to the application.

Program to demonstrate file class

```
import java.io.*;
class FileDemo
{
  public static void main(String args[]) throws IOException
  {
    File f=null;
    try
    {
      f=new File(args[0]);
    }catch(ArrayIndexOutOfBoundsException e)
    {
      System.out.println("we should give a file name in the command line");
    }
    if(f.exists())
    {
      String cr=f.canRead() ? "yes" : "no";
      String cw=f.canWrite() ? "yes" : "no";
      String isF=f.isFile() ? "yes" : "no";
      String isHid=f.isHidden() ? "yes" : "no";
      String isDir=f.isDirectory() ? "yes" : "no";
      System.out.println("Readable : " +cr);
      System.out.println("Writable : " +cw);
      System.out.println("is File : " +isF);
      System.out.println("is directory : " +isDir);
      System.out.println("is hidden : " +isHid);
    }
    else        System.out.println(" file does not exist");
  }
}
```

12. FileInputStream CLASS

Java FileInputStream class obtains input bytes from a file. It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc. We can also read character-stream data. But, for reading streams of characters, it is recommended to use FileReader class.

Java FileInputStream class declaration
Let's see the declaration for java.io.FileInputStream class:

public class FileInputStream **extends** InputStream

Java FileInputStream class methods

Method	Description
int available()	It is used to return the estimated number of bytes that can be read from the input stream.
int read()	It is used to read the byte of data from the input stream.
int read(byte[] b)	It is used to read up to **b.length** bytes of data from the input stream.
int read(byte[] b, int off, int len)	It is used to read up to **len** bytes of data from the input stream.
long skip(long x)	It is used to skip over and discards x bytes of data from the input stream.
FileChannel getChannel()	It is used to return the unique FileChannel object associated with the file input stream.
FileDescriptor getFD()	It is used to return the FileDescriptor object.
protected void finalize()	It is used to ensure that the close method is call when there is no more reference to the file input stream.
void close()	It is used to closes the stream.

Java FileInputStream example 1: read single character

```
import java.io.FileInputStream;
public class DataStreamExample
{
    public static void main(String args[]){
        try{
          FileInputStream fin=new FileInputStream("D:\\testout.txt");
          int i=fin.read();
          System.out.print((char)i);

          fin.close();
        }catch(Exception e){System.out.println(e);}
       }
      }
```

13. FileOutputStream CLASS

Java FileOutputStream is an output stream used for writing data to a file.

If we have to write primitive values into a file, use FileOutputStream class. We can write byte-oriented as well as character-oriented data through FileOutputStream class. But, for character-oriented data, it is preferred to use FileWriter than FileOutputStream.

FileOutputStream class declaration

Let's see the declaration for Java.io.FileOutputStream class:

public class FileOutputStream **extends** OutputStream

FileOutputStream class methods

Method	Description
protected void finalize()	It is used to clean up the connection with the file output stream.
void write(byte[] ary)	It is used to write **ary.length** bytes from the byte array to the file output stream.
void write(byte[] ary, int off, int len)	It is used to write **len** bytes from the byte array starting at offset **off** to the file output stream.
void write(int b)	It is used to write the specified byte to the file output stream.
FileChannel getChannel()	It is used to return the file channel object associated with the file output stream.
FileDescriptor getFD()	It is used to return the file descriptor associated with the stream.
void close()	It is used to closes the file output stream.

Java FileOutputStream Example 1: write byte

```
import java.io.FileOutputStream;
```

```
public class FileOutputStreamExample {
   public static void main(String args[]){
        try{
         FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
         fout.write(65);
         fout.close();
         System.out.println("success...");
        }catch(Exception e){System.out.println(e);}
    }
}
```

Output:

```
Success...
```

The content of a text file **testout.txt** is set with the data **A**.

testout.txt

```
A
```

14. Scanner CLASS

Java Scanner class comes under the java.util package. Java has various ways to read input from the keyboard, the java.util.Scanner class is one of them.

The Java Scanner class breaks the input into tokens using a delimiter that is whitespace by default. It provides many methods to read and parse various primitive values.

Java Scanner class is widely used to parse text for string and primitive types using a regular expression.

Java Scanner class extends Object class and implements Iterator and Closeable interfaces.

Java Scanner Class Declaration

```
public final class Scanner
        extends Object
        implements Iterator<String>
```

Java Scanner Class Constructors

SN	Constructor	Description
1)	Scanner(File source)	It constructs a new Scanner that produces values scanned from the specified file.
2)	Scanner(File source, String charsetName)	It constructs a new Scanner that produces values scanned from the specified file.
3)	Scanner(InputStream source)	It constructs a new Scanner that produces values scanned from the specified input stream.

Java Scanner Class Methods
The following are the list of Scanner methods:

SN	Modifier & Type	Method	Description
1)	void	close()	It is used to close this scanner.
2)	pattern	delimiter()	It is used to get the Pattern which the Scanner class is currently using to match delimiters.
3)	Stream <MatchResult>	findAll()	It is used to find a stream of match results that match the provided pattern string.
4)	String	findInLine()	It is used to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.
5)	string	findWithinHorizon()	It is used to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.
6)	boolean	hasNext()	It returns true if this scanner has another token in its input.

7)	boolean	hasNextBigDecimal()	It is used to check if the next token in this scanner's input can be interpreted as a BigDecimal using the nextBigDecimal() method or not.
8)	boolean	hasNextBigInteger()	It is used to check if the next token in this scanner's input can be interpreted as a BigDecimal using the nextBigDecimal() method or not.
9)	boolean	hasNextBoolean()	It is used to check if the next token in this scanner's input can be interpreted as a Boolean using the nextBoolean() method or not.
10)	boolean	hasNextByte()	It is used to check if the next token in this scanner's input can be interpreted as a Byte using the nextBigDecimal() method or not.
11)	boolean	hasNextDouble()	It is used to check if the next token in this scanner's input can be interpreted as a BigDecimal using the nextByte() method or not.
12)	boolean	hasNextFloat()	It is used to check if the next token in this scanner's input can be interpreted as a Float using the nextFloat() method or not.
13)	boolean	hasNextInt()	It is used to check if the next token in this scanner's input can be interpreted as an int using the nextInt() method or not.
14)	boolean	hasNextLine()	It is used to check if there is another line in the input of this scanner or not.
15)	boolean	hasNextLong()	It is used to check if the next token in this scanner's input can be interpreted as a Long using the nextLong() method or not.
16)	boolean	hasNextShort()	It is used to check if the next token in this scanner's input can be interpreted as a Short using the nextShort() method or not.

17)	IOException	ioException()	It is used to get the IOException last thrown by this Scanner's readable.
18)	Locale	locale()	It is used to get a Locale of the Scanner class.
19)	MatchResult	match()	It is used to get the match result of the last scanning operation performed by this scanner.
20)	String	next()	It is used to get the next complete token from the scanner which is in use.
21)	BigDecimal	nextBigDecimal()	It scans the next token of the input as a BigDecimal.
22)	BigInteger	nextBigInteger()	It scans the next token of the input as a BigInteger.
23)	boolean	nextBoolean()	It scans the next token of the input into a boolean value and returns that value.
24)	byte	nextByte()	It scans the next token of the input as a byte.
25)	double	nextDouble()	It scans the next token of the input as a double.
26)	float	nextFloat()	It scans the next token of the input as a float.
27)	int	nextInt()	It scans the next token of the input as an Int.
28)	String	nextLine()	It is used to get the input string that was skipped of the Scanner object.
29)	long	nextLong()	It scans the next token of the input as a long.
30)	short	nextShort()	It scans the next token of the input as a short.
31)	int	radix()	It is used to get the default radix of the Scanner use.

32)	void	remove()	It is used when remove operation is not supported by this implementation of Iterator.
33)	Scanner	reset()	It is used to reset the Scanner which is in use.
34)	Scanner	skip()	It skips input that matches the specified pattern, ignoring delimiters
35)	Stream<String>	tokens()	It is used to get a stream of delimiter-separated tokens from the Scanner object which is in use.
36)	String	toString()	It is used to get the string representation of Scanner using.
37)	Scanner	useDelimiter()	It is used to set the delimiting pattern of the Scanner which is in use to the specified pattern.
38)	Scanner	useLocale()	It is used to sets this scanner's locale object to the specified locale.
39)	Scanner	useRadix()	It is used to set the default radix of the Scanner which is in use to the specified radix.

Example 1

```
import java.util.*;

public class ScannerClassExample1 {

    public static void main(String args[]){
       String s = "Hello, This is Lbc.";
       //Create scanner Object and pass string in it
       Scanner scan = new Scanner(s);
       //Check if the scanner has a token
       System.out.println("Boolean Result: " + scan.hasNext());
       //Print the string
       System.out.println("String: " +scan.nextLine());
       scan.close();
       System.out.println("--------Enter Wer Details-------- ");
       Scanner in = new Scanner(System.in);
       System.out.print("Enter wer name: ");
       String name = in.next();
       System.out.println("Name: " + name);
       System.out.print("Enter wer age: ");
```

```
        int i = in.nextInt();
        System.out.println("Age: " + i);
        System.out.print("Enter wer salary: ");
        double d = in.nextDouble();
        System.out.println("Salary: " + d);
        in.close();
        }
}
```

Output:

```
Boolean Result: true
String: Hello, This is Lbc.
-------Enter Wer Details---------
Enter wer name: Abhishek
Name: Abhishek
Enter wer age: 23
Age: 23
Enter wer salary: 25000
Salary: 25000.0
```

15. BufferedInputStream CLASS

Java BufferedInputStream class is used to read information from stream. It internally uses buffer mechanism to make the performance fast.

The important points about BufferedInputStream are:

- When the bytes from the stream are skipped or read, the internal buffer automatically refilled from the contained input stream, many bytes at a time.
- When a BufferedInputStream is created, an internal buffer array is created.

Java BufferedInputStream class declaration

Let's see the declaration for Java.io.BufferedInputStream class:

1. **public class** BufferedInputStream **extends** FilterInputStream

Java BufferedInputStream class constructors

Constructor	Description
BufferedInputStream(InputStream IS)	It creates the BufferedInputStream and saves it argument, the input stream IS, for later use.
BufferedInputStream(InputStream IS, int size)	It creates the BufferedInputStream with a specified buffer size and saves it argument, the input stream IS, for later use.

Java BufferedInputStream class methods

Method	Description
int available()	It returns an estimate number of bytes that can be read from the input stream without blocking by the next invocation method for the input stream.
int read()	It read the next byte of data from the input stream.
int read(byte[] b, int off, int ln)	It read the bytes from the specified byte-input stream into a specified byte array, starting with the given offset.
void close()	It closes the input stream and releases any of the system resources associated with the stream.
void reset()	It repositions the stream at a position the mark method was last called on this input stream.
void mark(int readlimit)	It sees the general contract of the mark method for the input stream.
long skip(long x)	It skips over and discards x bytes of data from the input stream.
boolean markSupported()	It tests for the input stream to support the mark and reset methods.

Example of Java BufferedInputStream

```
import java.io.*;
public class BufferedInputStreamExample{
 public static void main(String args[]){
  try{
    FileInputStream fin=new FileInputStream("D:\\testout.txt");
    BufferedInputStream bin=new BufferedInputStream(fin);
    int i;
    while((i=bin.read())!=-1){
     System.out.print((char)i);
    }
    bin.close();
    fin.close();
  }catch(Exception e){System.out.println(e);}
 }
}
```

Here, we are assuming that we have following data in **"testout.txt"** file: lbc

Output: lbc

16. BufferedOutputStream CLASS

Java BufferedOutputStream class is used for buffering an output stream. It internally uses buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast. For adding the buffer in an OutputStream, use the BufferedOutputStream class. Syntax for adding the buffer in an OutputStream:

OutputStream os= **new** BufferedOutputStream(**new** FileOutputStream("D:\\testout.txt"));

Java BufferedOutputStream class declaration

Declaration : **public class** BufferedOutputStream **extends** FilterOutputStream

Java BufferedOutputStream class constructors

Constructor	Description
BufferedOutputStream(OutputStream os)	It creates the new buffered output stream which is used for writing the data to the specified output stream.
BufferedOutputStream (OutputStream os, int size)	It creates the new buffered output stream which is used for writing the data to the specified output stream with a specified buffer size.

Java BufferedOutputStream class methods

Method	Description
void write(int b)	It writes the specified byte to the buffered output stream.
void write(byte[] b, int off, int len)	It write the bytes from the specified byte-input stream into a specified byte array, starting with the given offset
void flush()	It flushes the buffered output stream.

Example of BufferedOutputStream class:

In this example, we are writing the textual information in the BufferedOutputStream object which is connected to the FileOutputStream object. The flush() flushes the data of one stream and send it into another. It is required if we have connected the one stream with another.

```
package com.lbc;
import java.io.*;
public class BufferedOutputStreamExample
{
        public static void main(String args[])throws Exception{
    FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
    BufferedOutputStream bout=new BufferedOutputStream(fout);
    String s="Welcome to lbc.";
    byte b[]=s.getBytes();
    bout.write(b);
    bout.flush();
    bout.close();
    fout.close();
    System.out.println("success");
}
}
```

Output:

```
Success
```

testout.txt

```
Welcome to lbc.
```

17. RandomAccessFile CLASS

The character and byte stream are all sequential access streams whose contents must be read or written sequentially. In contrast, Random Access File lets you randomly access the contents of a file. The Random Access File allows files to be accessed at a specific point in the file. They can be opened in read/write mode, which allows updating of a current file.

The Random Access File class is not derived from InputStream and OuputStream. Instead it implements DataInput and DataOutput and thus provides a set of methods from both DataInputStream and DataOutputStream.

An object of this class should be created when access to binary data is required but in non-sequential form. The same object can be used to both read and write the file.

To create a new Random Access file pass the name of file and mode to the constructor. The mode is either “r (read only) or “rw” (read and write).

Following are RandomAccessFile **constructors**

RandomAccessFile(String s, String mode) throws IOException

s is the file name and mode is “r” and “rw”

RandomAccessFile(File f, String mode) throws IOException

f is an File Object and mode is “r” and “rw”

Methods

i. long length() returns length of bytes in afile

ii. void seek(long offset) throws IOException position the file pointer at a particular point in a file. The new position is current position plus the offset. The offset may be positive or negative.

iii. Long getFilePointer() throws IOException : returns the index of the current read write position within the file.

iv. void close() throws IOException : close file and free the resource to system.

*Program to demonstrate **RandomAccessFile***

```
import java.io.*;
class RAFile{
public static void  main (String args[]) throws Exception{
        RandomAccessFile ras = new RandomAccessFile("ras.txt","rw");
        byte b[] = {1,2,3,4,5,65,6,3,33,52,32,2,2,33,4,4};
        writeRecord(ras,"x",100,10000F,b);
        System.out.println(" record pointer is at " + ras.getFilePointer());
        writeRecord(ras,"y",101,10000F,b);
        System.out.println(" record pointer is at " + ras.getFilePointer());
        writeRecord(ras,"a",102,10000F,b);
        writeRecord(ras,"b",103,10000F,b);
        writeRecord(ras,"c",104,10000F,b);
        printRecords(ras,1);
        printRecords(ras,2);
        printRecords(ras,20);
        ras.close();
}
private static void writeRecord(RandomAccessFile r,String name,int eno,
float sal,byte photo[]) throws Exception{
        r.writeUTF(name);
        r.writeInt(eno);
        r.writeFloat(sal);
        r.write(photo,0,photo.length);
        r.writeUTF("\n");
}
private static void printRecords(RandomAccessFile r,int recno)
throws Exception{
        long curpos = r.getFilePointer();
        long newpos = (recno-1) * 30;
        r.seek(newpos);
        System.out.print(r.readUTF()+"  ");
        System.out.print(r.readInt()+"  ");
        System.out.println(r.readFloat());
        r.seek(curpos);
}
}
```

UNIT IV

1. **APPLETS: INTRODUCTION, EXAMPLE**
2. **LIFE CYCLE OF APPLET** **12 marks**
3. **APPLET CLASS**
4. **COMMON METHODS USED IN DISPLAYING THE OUTPUT USING APPLET**
5. **EVENT HANDLING: INTRODUCTION TYPES OF EVENTS, EXAMPLE** **12 marks**
6. **AWT: INTRODUCTION, COMPONENTS, CONTAINERS** **12 marks**
7. **BUTTON, LABEL, CHECKBOX, RADIO BUTTONS** **12 marks**
8. **CONTAINER CLASS**
9. **DIFFERENCES BETWEEN SWING AND AWT** **12 marks**
10. **COMPONENTS IN SWINGS** **12 marks**
11. **DATABASE HANDLING USING JDBC: INTRODUCTION**
12. **TYPES OF JDBC DRIVERS** **12 marks**
13. **LOAD THE DRIVER AND ESTABLISH CONNECTION** **12 marks**
14. **CREATE STATEMENT AND EXECUTE QUERY** **12 marks**
15. **ITERATE RESULTSET**
16. **SCROLLABLE RESULTSET** **12 marks**

1. APPLETS: INTRODUCTION, EXAMPLE

An **applet** is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.

There are some important differences between an applet and a standalone Java application, including the following −

- An applet is a Java class that extends the java.applet.Applet class.
- A main() method is not invoked on an applet, and an applet class will not define main().
- Applets are designed to be embedded within an HTML page.
- When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.
- A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.
- The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.
- Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.
- Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file.

A "Hello, World" Applet

Following is a simple applet named HelloWorldApplet.java −

```
import java.applet.*;
import java.awt.*;

public class HelloWorldApplet extends Applet {
  public void paint (Graphics g) {
    g.drawString ("Hello World", 25, 50);
  }
}
```

These import statements bring the classes into the scope of our applet class −

- java.applet.Applet
- java.awt.Graphics

Without those import statements, the Java compiler would not recognize the classes Applet and Graphics, which the applet class refers to.

Invoking an Applet

An applet may be invoked by embedding directives in an HTML file and viewing the file through an applet viewer or Java-enabled browser.

The <applet> tag is the basis for embedding an applet in an HTML file. Following is an example that invokes the "Hello, World" applet −

```
<html>
  <title>The Hello, World Applet</title>
  <hr>
  <applet code = "HelloWorldApplet.class" width = "320" height = "120">
    If your browser was Java-enabled, a "Hello, World"
    message would appear here.
  </applet>
  <hr>
</html>
```

2. LIFE CYCLE OF APPLET

Lifecycle of Java Applet

Applet is initialized.

Applet is started.

Applet is painted.

Applet is stopped.

Applet is destroyed.

Lifecycle methods for Applet:

The java.applet.Applet class 4 life cycle methods and java.awt.Component class provides 1 life cycle methods for an applet.

java.applet.Applet class

For creating any applet java.applet.Applet class must be inherited. It provides 4 life cycle methods of applet.

public void init(): is used to initialized the Applet. It is invoked only once.

public void start(): is invoked after the init() method or browser is maximized. It is used to start the Applet.

public void stop(): is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.

public void destroy(): is used to destroy the Applet. It is invoked only once.

java.awt.Component class

The Component class provides 1 life cycle method of applet.

public void paint(Graphics g): is used to paint the Applet. It provides Graphics class object that can be used for drawing oval, rectangle, arc etc.

Java Plug in software is responsible to manage the life cycle of an applet

---

How to run an Applet?

There are two ways to run an applet

By html file.

By appletViewer tool (for testing purpose).

Simple example of Applet by html file:

To execute the applet by html file, create an applet and compile it. After that create an html file and place the applet code in html file. Now click the html file.

//First.java

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.*;
/*<applet code="AppletLifeCycle.class" width="350" height="150"> </applet>*/
public class AppletLifeCycle extends Applet
{
    public void init()
    {
      setBackground(Color.CYAN);
      System.out.println("init() called");
    }
    public void start(){ System.out.println("Start() called"); }
    public void paint(Graphics g){ System.out.println("Paint(() called"); }
    public void stop() { System.out.println("Stop() Called"); }
    public void destroy()   { System.out.println("Destroy)() Called"); }
}
```

Note: class must be public because its object is created by Java Plugin software that resides on the browser.

3. APPLET CLASS

Every applet is an extension of the *java.applet.Applet class*. The base Applet class provides methods that a derived Applet class may call to obtain information and services from the browser context.

These include methods that do the following −

- Get applet parameters
- Get the network location of the HTML file that contains the applet
- Get the network location of the applet class directory
- Print a status message in the browser

- Fetch an image
- Fetch an audio clip
- Play an audio clip
- Resize the applet

Additionally, the Applet class provides an interface by which the viewer or browser obtains information about the applet and controls the applet's execution. The viewer may −

- Request information about the author, version, and copyright of the applet
- Request a description of the parameters the applet recognizes
- Initialize the applet
- Destroy the applet
- Start the applet's execution
- Stop the applet's execution

The Applet class provides default implementations of each of these methods. Those implementations may be overridden as necessary.

The "Hello, World" applet is complete as it stands. The only method overridden is the paint method.

4. COMMON METHODS USED IN DISPLAYING THE OUTPUT USING APPLET

Commonly used methods of Graphics class to display

1. **public abstract void drawString(String str, int x, int y):** is used to draw the specified string.
2. **public void drawRect(int x, int y, int width, int height):** draws a rectangle with the specified width and height.
3. **public abstract void fillRect(int x, int y, int width, int height):** is used to fill rectangle with the default color and specified width and height.
4. **public abstract void drawOval(int x, int y, int width, int height):** is used to draw oval with the specified width and height.
5. **public abstract void fillOval(int x, int y, int width, int height):** is used to fill oval with the default color and specified width and height.
6. **public abstract void drawLine(int x1, int y1, int x2, int y2):** is used to draw line between the points(x1, y1) and (x2, y2).
7. **public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer):** is used draw the specified image.
8. **public abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used draw a circular or elliptical arc.
9. **public abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used to fill a circular or elliptical arc.
10. **public abstract void setColor(Color c):** is used to set the graphics current color to the specified color.
11. **public abstract void setFont(Font font):** is used to set the graphics current font to the specified font.

Example of Graphics in applet:

```
import java.applet.Applet;
import java.awt.*;

public class GraphicsDemo extends Applet{

public void paint(Graphics g){
g.setColor(Color.red);
g.drawString("Welcome",50, 50);
g.drawLine(20,30,20,300);
g.drawRect(70,100,30,30);
g.fillRect(170,100,30,30);
g.drawOval(70,200,30,30);

g.setColor(Color.pink);
g.fillOval(170,200,30,30);
g.drawArc(90,150,30,30,30,270);
g.fillArc(270,150,30,30,0,180);

}
}
```

myapplet.html

```
<html>
<body>
<applet code="GraphicsDemo.class" width="300" height="300">
</applet>
</body>
</html>
```

5. EVENT HANDLING: INTRODUCTION TYPES OF EV ENTS

Changing the state of an object is known as an event. For example, click on button, dragging mouse etc. The java.awt.event package provides many event classes and Listener interfaces for event handling.

Java Event classes and Listener interfaces

Event Classes	Listener Interfaces
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener

MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

Steps to perform Event Handling

Following steps are required to perform event handling:

Register the component with the Listener

Registration Methods

For registering the component with the Listener, many classes provide the registration methods. For example:

Button

public void addActionListener(ActionListener a){}

MenuItem

public void addActionListener(ActionListener a){}

TextField

public void addActionListener(ActionListener a){}
public void addTextListener(TextListener a){}

TextArea

public void addTextListener(TextListener a){}

Checkbox

public void addItemListener(ItemListener a){}

Choice

public void addItemListener(ItemListener a){}

List

public void addActionListener(ActionListener a){}
public void addItemListener(ItemListener a){}

Java event handling by implementing ActionListener

```
import java.awt.*;
import java.awt.event.*;
class AEvent extends Frame implements ActionListener
{
    TextField tf;
    AEvent(){

//create components
    tf=new TextField();
    tf.setBounds(60,50,170,20);
    Button b=new Button("click me");
    b.setBounds(100,120,80,30);

//register listener
    b.addActionListener(this);//passing current instance

//add components and set size, layout and visibility
    add(b);add(tf);
    setSize(300,300);
    setLayout(null);
    setVisible(true);
}
public void actionPerformed(ActionEvent e)
{
    tf.setText("Welcome");
}
public static void main(String args[])
{
    new AEvent();
}
}
```

6. AWT: INTRODUCTION, COMPONENTS, CONTAINERS

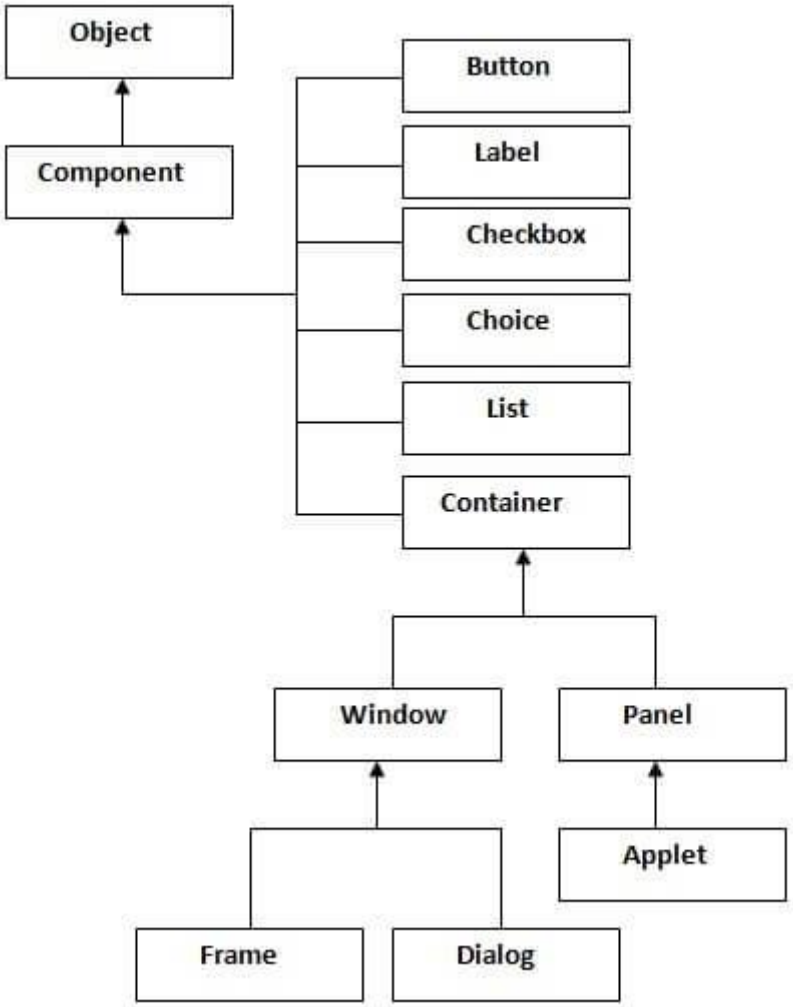
Java AWT (Abstract Window Toolkit) is *an API to develop GUI or window-based applications* in java.

Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavyweight i.e. its components are using the resources of OS.

The java.awt package provides classes for AWT api such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

Java AWT Hierarchy

The hierarchy of Java AWT classes are given below.



Container

The Container is a component in AWT that can contain another components like buttons, textfields, labels etc. The classes that extends Container class are known as container such as Frame, Dialog and Panel.

Window

The window is the container that have no borders and menu bars. You must use frame, dialog or another window for creating a window.

Panel

The Panel is the container that doesn't contain title bar and menu bars. It can have other components like button, textfield etc.

Frame

The Frame is the container that contain title bar and can have menu bars. It can have other components like button, textfield etc.

Useful Methods of Component class

Method	Description
public void add(Component c)	inserts a component on this component.
public void setSize(int width,int height)	sets the size (width and height) of the component.
public void setLayout(LayoutManager m)	defines the layout manager for the component.
public void setVisible(boolean status)	changes the visibility of the component, by default false.

Java AWT Example

To create simple awt example, you need a frame. There are two ways to create a frame in AWT.

By extending Frame class (inheritance)

By creating the object of Frame class (association)

AWT Example by Inheritance

Let's see a simple example of AWT where we are inheriting Frame class. Here, we are showing Button component on the Frame.

```
import java.awt.*;
class First extends Frame{
First(){
Button b=new Button("click me");
b.setBounds(30,100,80,30);// setting button position
add(b);//adding button into frame
setSize(300,300);//frame size 300 width and 300 height
setLayout(null);//no layout manager
setVisible(true);//now frame will be visible, by default not visible
}
public static void main(String args[]){
First f=new First();
}}
```

download this example

The setBounds(int xaxis, int yaxis, int width, int height) method is used in the above example that sets the position of the awt button.

7. BUTTON, LABEL, CHECKBOX, RADIO BUTTONS

Java AWT Button

The button class is used to create a labeled button that has platform independent implementation. The application result in some action when the button is pushed.

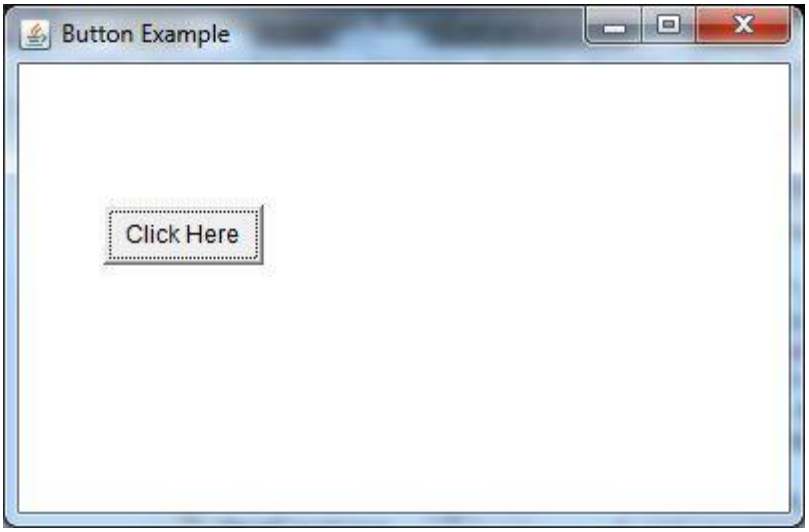
AWT Button Class declaration

public class Button **extends** Component **implements** Accessible

Java AWT Button Example

```
import java.awt.*;
public class ButtonExample {
public static void main(String[] args) {
    Frame f=new Frame("Button Example");
    Button b=new Button("Click Here");
    b.setBounds(50,100,80,30);
    f.add(b);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
}
}
```

Output:



Java AWT Label

The object of Label class is a component for placing text in a container. It is used to display a single line of read only text. The text can be changed by an application but a user cannot edit it directly.

AWT Label Class Declaration

public class Label **extends** Component **implements** Accessible

Java Label Example

```
import java.awt.*;
class LabelExample{
public static void main(String args[]){
    Frame f= new Frame("Label Example");
    Label l1,l2;
    l1=new Label("First Label.");
    l1.setBounds(50,100, 100,30);
    l2=new Label("Second Label.");
    l2.setBounds(50,150, 100,30);
    f.add(l1); f.add(l2);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
}
}
```

Output:

The Checkbox class is used to create a checkbox. It is used to turn an option on (true) or off (false). Clicking on a Checkbox changes its state from "on" to "off" or from "off" to "on".

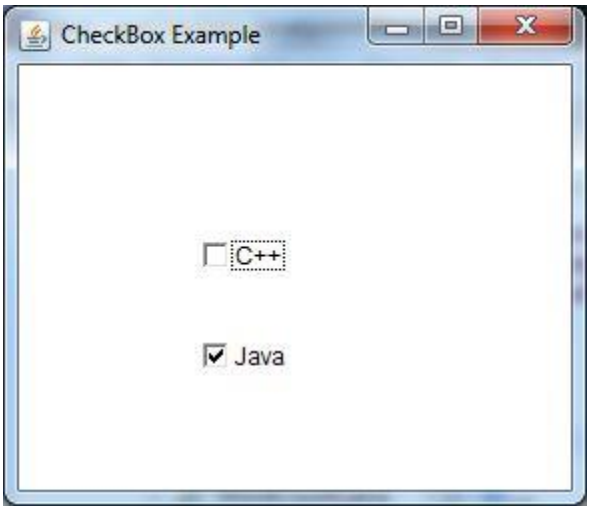
AWT Checkbox Class Declaration

public class Checkbox **extends** Component **implements** ItemSelectable, Accessible

Java AWT Checkbox Example

```
import java.awt.*;
public class CheckboxExample
{
    CheckboxExample(){
      Frame f= new Frame("Checkbox Example");
       Checkbox checkbox1 = new Checkbox("C++");
       checkbox1.setBounds(100,100, 50,50);
       Checkbox checkbox2 = new Checkbox("Java", true);
       checkbox2.setBounds(100,150, 50,50);
       f.add(checkbox1);
       f.add(checkbox2);
       f.setSize(400,400);
       f.setLayout(null);
       f.setVisible(true);
    }
public static void main(String args[])
{
   new CheckboxExample();
}
}
```

Output:



8. CONTAINER CLASS

The class **Container** is the super class for the containers of AWT. Container object can contain other AWT components.

Class declaration

Following is the declaration for **java.awt.Container** class:

```
public class Container    extends Component
```

Class constructors

S.N.	Constructor & Description
1	**Container()** This creates a new Container.

Class methods

S.N.	Method & Description
1	**Component add(Component comp)** Appends the specified component to the end of this container.
2	**Component add(Component comp, int index)** Adds the specified component to this container at the given position.
3	**void addContainerListener(ContainerListener l)** Adds the specified container listener to receive container events from this container.
4	**void addNotify()** Makes this Container displayable by connecting it to a native screen resource.
5	**void addPropertyChangeListener(PropertyChangeListener listener)** Adds a PropertyChangeListener to the listener list.
6	**boolean areFocusTraversalKeysSet(int id)** Returns whether the Set of focus traversal keys for the given focus traversal operation has been explicitly defined for this Container.
7	**Component findComponentAt(int x, int y)** Locates the visible child component that contains the specified position.
8	**float getAlignmentX()** Returns the alignment along the x axis.
9	**float getAlignmentY()** Returns the alignment along the y axis.
10	**Component getComponent(int n)** Gets the nth component in this container.

11	**Dimension getMinimumSize()** Returns the minimum size of this container.
12	**void paint(Graphics g)** Paints the container.
13	**void print(Graphics g)** Prints the container.
14	**void remove(Component comp)** Removes the specified component from this container.
15	**void removeAll()** Removes all the components from this container.
16	**void setFont(Font f)** Sets the font of this container.
17	**void update(Graphics g)** Updates the container.
18	**void validate()** Validates this container and all of its subcomponents.

Methods inherited

This class inherits methods from the following classes:

- java.awt.Component
- java.lang.Object

9. DIFFERENCES BETWEEN SWING AND AWT

There are many differences between java awt and swing that are given below.

No.	Java AWT	Java Swing
1)	AWT components are **platform-dependent**.	Java swing components are **platform-independent**.
2)	AWT components are **heavyweight**.	Swing components are **lightweight**.
3)	AWT **doesn't support pluggable look and feel**.	Swing **supports pluggable look and feel**.

4)	AWT provides **less components** than Swing.	Swing provides **more powerful components** such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT **doesn't follows MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing **follows MVC**.

10. COMPONENTS IN SWINGS

The class **Component** is the abstract base class for the non menu user-interface controls of AWT. Component represents an object with graphical representation.

Class Declaration

Following is the declaration for **java.awt.Component** class −

```
public abstract class Component    extends Object
    implements ImageObserver, MenuContainer, Serializable
```

Following are the fields for **java.awt.Component** class −

- **static float BOTTOM_ALIGNMENT** − Ease-of-use constant for getAlignmentY.
- **static float CENTER_ALIGNMENT** − Ease-of-use constant for getAlignmentY and getAlignmentX.
- **static float LEFT_ALIGNMENT** − Ease-of-use constant for getAlignmentX.
- **static float RIGHT_ALIGNMENT** − Ease-of-use constant for getAlignmentX.
- **static float TOP_ALIGNMENT** − Ease-of-use constant for getAlignmentY().

Class Constructors

S.No.	Constructor & Description
1	**protected Component()** This creates a new Component.

Class Methods

Here is the list of methods in Swing Component class.

Methods Inherited

This class inherits methods from the following class −

- java.lang.Object

11. DATABASE HANDLING USING JDBC: INTRODUCTION

What is JDBC?

JDBC is the Java Database Connectivity API to integrate relational databases with Java programs. Java Database Connectivity is a set of relational database objects and methods for interacting with SQL data sources.

How Does JDBC Work?

JDBC is designed on the Call Level Interface model. A Java Program first opens a connection to a database, makes Statement Object, passes SQL statements to the underlying DBMS through the *Statement* object, and retrieves the results as well as information about the result sets.

What are the packages of JDBC API?

i) java.sql

This package contains classes and interfaces designed with traditional client-server architecture in mind. Its functionality is focused on basic database programming services such as creating connections, executing statements and prepared statements, and running batch queries. Advanced functions such as batch updates, scrollable resultsets, transaction isolation, and SQL data types are also available

ii) javax.sql

This package introduces major architectural changes to JDBC programming compared to java.sql, and provides better abstractions for connection management, distributed transactions, and legacy connectivity. This package also introduces container-managed connection pooling, distributed transactions, and rowsets.

What is Database Driver?

A database vendor typically provides a set of APIs for accessing the data managed by the database server. Popular database vendors such as Oracle, Sybase, and Informix provide proprietary APIs for client access. Client applications written in native languages such as C/C++ can use these APIs to gain direct access to the data.

12. TYPES OF JDBC DRIVERS

There are four different approaches to connect an application to a database server via a database driver. They are as following

i) Type 1 – JDBC-ODBC Bridge

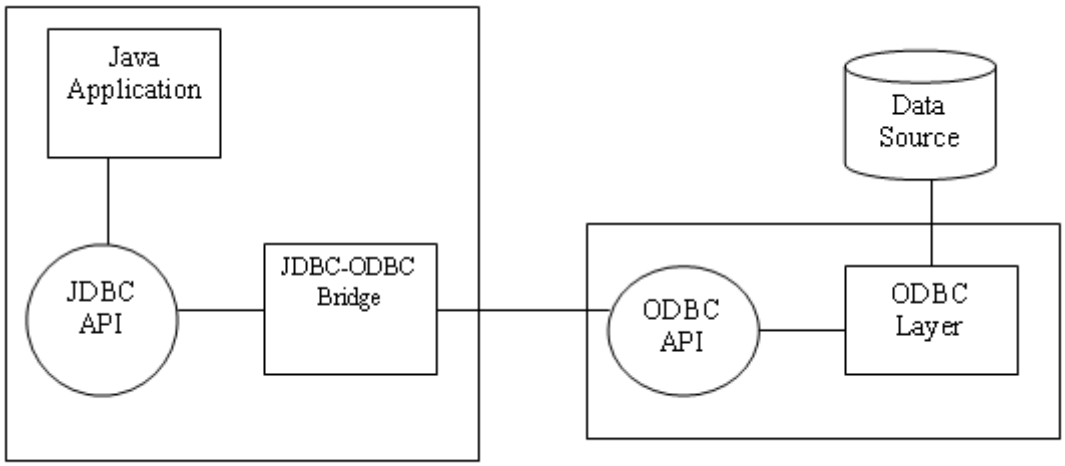
ii) Type 2 – Native API partly-Java driver

iii) Type 3 – Net Protocol Pure Java Driver (or) Intermediate Database Access Server

iv) Type 4 – Pure Java Drivers

i) Type 1 – JDBC-ODBC Bridge

This driver provides a bridge between the JDBC API and the ODBC API. The Bridge translates the standard JDBC calls to corresponding ODBC calls, and sends them to the ODBC data source via ODBC libraries.



JDBC-ODBC Bridge is invoked within the client application process, the ODBC layer executes in another process. This configuration requires every client that will run the application to have the JDBC-ODBC Bridge API, the ODBC driver, and the native language-level API such as the OCI library for Oracle, installed.

Purpose : the use of JDBC-ODBC should be considered for experimental purposes only. This solution for data access is inefficient for high-performance database access requirements because of the multiple layers of indirection for each data access call. In addition this solution limits the functionality of the JDBC API to that of the ODBC driver.

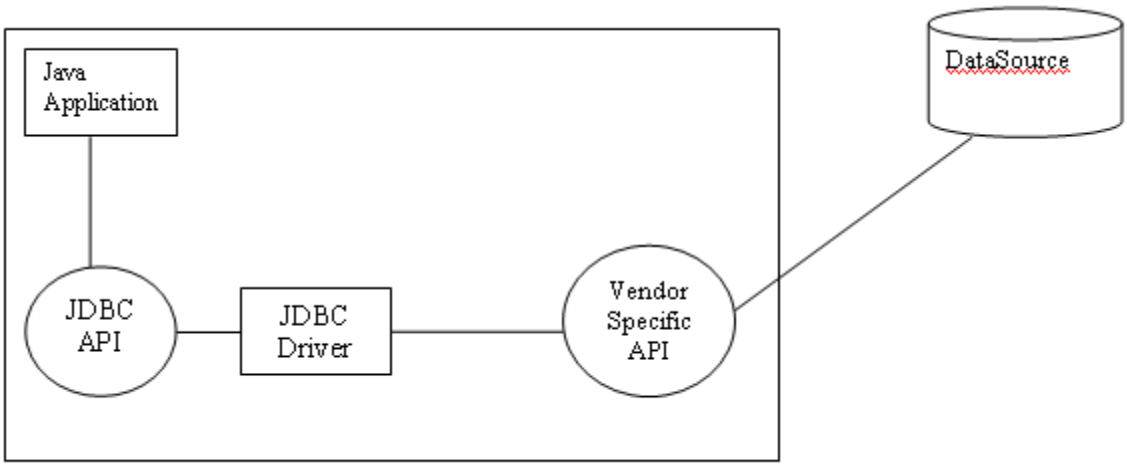
To establish connection of type 1 driver

1. loading driver : Class.forName("sun.jdbc.odbc.JdbcOdbcDriver")
2. making the connection :

   Connection con = DriverManager.getConnection("jdbc:odbc:empdsn","scott","tiger")

ii) Type 2 – Native API partly-Java driver

Type 2 driver use a mixture of Java implementation and vendor-specific native APIs to provide data access. JDBC database calls are translated into vendor-specific API calls. The database will process the request and send the results back through the API, which will in turn forward them back to the JDBC driver. The JDBC driver will translate the results to the JDBC standard and return them to the Java Application.



There is one layer fewer to go through than for a Type 1 driver and so in general a Type 2 driver will be faster than a Type 1 driver.

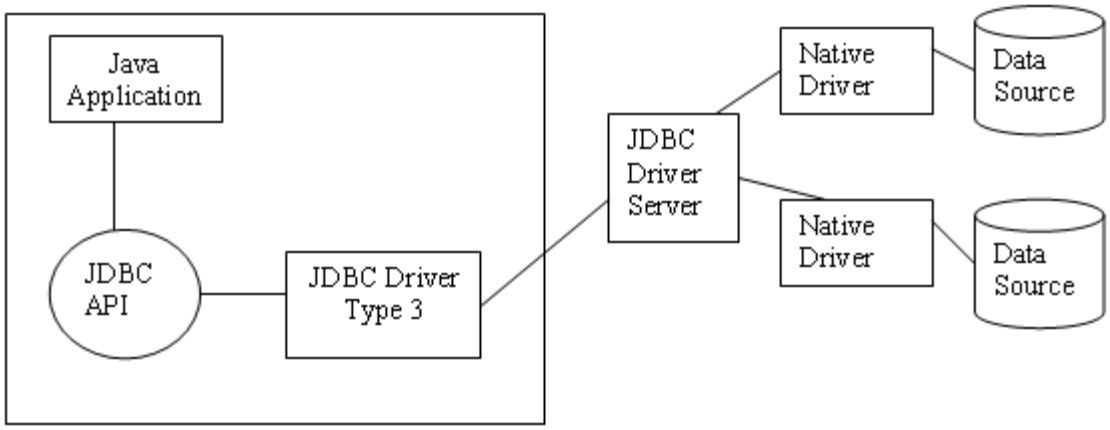
To establish connection of type 2 driver

1. loading driver : Class.forName("oracle.jdbc.driver.OracleDriver")
2. making the connection :

Connection con = DriverManager.getConnection("jdbc:oracle:oci8:@dbname","uname","pwd")

iii) Type 3 – Net Protocol Pure Java Driver (or) Intermediate Database Access Server

Type 3 drivers use Intermediate (middleware) database server that has the ability to connect multiple Java clients to multiple database servers.



Client Connect to database servers via an intermediate server component (such as a listener) that acts a gateway for multiple database servers. The Java client application sends a JDBC call through a JDBC driver (type 3) to the intermediate data access server, which completes the request to the Data Source using another driver (ex: type 2 or type 4 driver).

The protocol used to communicate between clients and the intermediate server depends on the middleware server vender but the intermediate server can use different native protocols to connect to different databases.

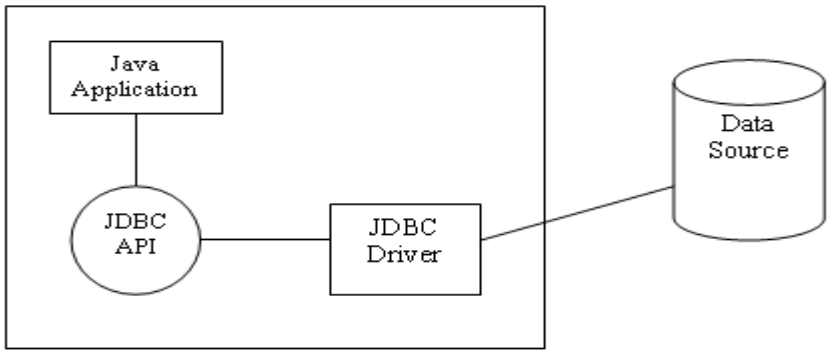
BEA WebLogic includes a type 3 driver. The advantage of this approach is that it removes the necessity to install network libraries on client machines.

To establish connection of type 3 driver

1. loading driver : Class.forName(“ids.sql.IDSDriver”)
2. making the connection : Connection con = DriverManager.getConnection(“jdbc:ids\\localhost:12\conn?dsn=dsnname”,”uname”,”pwd”)

iv) Type 4 – Pure Java Drivers

Type 4 drivers are a pure Java alternative to Type-2 driver. Type 4 drivers convert the JDBC API calls to direct network calls using vendor specific networking protocols. They do this by making direct socket connections with the database.



Type 4 drivers offer better performance than type 1 and type 2 drivers. Type 4 drivers are also the simplest drivers to deploy since there are additional libraries or middleware to install. All the major database vendors provide Type 4 JDBC drivers for their databases and they are also available from third party vendors.

To establish connection of type 2 driver

3. loading driver : Class.forName(“oracle.jdbc.driver.OracleDriver”)
4. making the connection : Connection con = DriverManager.getConnection(“jdbc:oracle:thin:@localhost:1521:dbname”,”uname”,”pwd”)

13. LOAD THE DRIVER and ESTABLISHING CONNECTION

Every Database Driver provider must be register an instance of the driver with the java.sql.DriverManager class before use. This Registration happens automatically when we load the driver class.

In JDBC, we try to load the database driver using the current java.lang.ClassLoader object. For example, you want to use the JDBC-ODBC Bridge driver, the following code will load it:

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

Your driver documentation will give you the class name to use. For instance, if the class name is jdbc.DriverXYZ , you would load the driver with the following line of code:

Class.forName("jdbc.DriverXYZ");

The ClassLoader locates and loads the class (ex: driver jdbc.DriverXYZ) from the classpath using the bootsrap class loader. While loading the class sthe class loader executes any static initialization code for the class (which registers the class).

DriverManager class in JDBC is to provide a common access layer on top of different database drivers used in an application. In this approach, instead of using individual Driver implementation class directly, applications use the DriverManager class to obtain the connections.

ESTABLISH CONNECTION

The second step in establishing a connection is to have the appropriate driver connect to the DBMS. The following line of code illustrates the general idea:

Connection con = DriverManager.getConnection(url,"myLogin", "myPassword");

Ex:

Connection con=DriverManager.getConnection("jdbc:odbc:Movies","username","pwd")

url The first part is url and its syntax is as follows

jdbc:<subprotocol>:<subname>

This has three parts

*Protoco*l: jdbc is the protocol. This is the only allowed protocol in JDBC

Sub-Protocol: this is used to identify a database driver, or the name of database connectivity mechanism, chosen by the database driver providers.

Sub-name:the syntax of the subname is driver specific. A driver may chose any syntax appropriate for its implementation.

Ex: jdbc:odbc:Movies

Jdbc is protocol , odbc is sub protocol and Movies is the dsn name of jdbc-odbc bridge (which is the specific way of syntax in driver implementation).

14. CREATE STATEMENT AND EXECUTE QUERY

Creating and Executing Statements

Statements that create a table, alter a table, or drop a table are all examples of DDL statements and are executed with the method executeUpdate. The method executeUpdate is also used to execute SQL statements that update a table.

Entering Data into a Table

The following code inserts one row of data, with Colombian in the column COF_NAME , 101 in SUP_ID , 7.99 in PRICE , 0 in SALES , and 0 in TOTAL. We will create a Statement object and then execute it using the method executeUpdate .

```
Ex:   Statement stmt = con.createStatement();
      stmt.executeUpdate( "INSERT INTO COFFEES " +
            "VALUES ('Colombian', 101, 7.99, 0, 0)");
      stmt.executeUpdate("INSERT INTO COFFEES " +
            "VALUES ('French_Roast', 49, 8.99, 0, 0)");
```

Getting Data from a Table

The following SQL statement selects the whole table:

```
SELECT * FROM COFFEES
```

The result, which is the entire table, will look similar to the following:

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	0	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

Here is another example of a SELECT statement; this one will get a list of coffees and their respective prices per pound:

```
SELECT COF_NAME, PRICE FROM COFFEES
```

The results of this query will look something like this:

COF_NAME	PRICE
Colombian	7.99
French_Roast	8.99
Espresso	9.99
Colombian_Decaf	8.99
French_Roast_Decaf	9.99

The following SQL statement limits the coffees selected to just those that cost less than $9.00 per pound:

```
SELECT COF_NAME, PRICE      FROM COFFEES WHERE PRICE < 9.00
```

The results would look similar to this:

COF_NAME	PRICE
Colombian	7.99
French_Roast	8.99
Colombian Decaf	8.99

Executing Qurey FROM RESULT SETS

JDBC returns results in a ResultSet object, so we need to declare an instance of the class ResultSet to hold our results. The following code demonstrates declaring the ResultSet object rs and assigning the results of our earlier query to it:

ResultSet rs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM COFFEES");

Using the Method next

The variable rs , which is an instance of ResultSet , contains the rows of coffees and prices shown in the result set example above. The method next of ResultSet is used to moves to next row and is called a cursor to the next row. Since the cursor is initially positioned just above the first row of a ResultSet object, the first call to the method next moves the cursor to the first row and makes it the current row. We can move the cursor backwards, to specific positions, and to positions relative to the current row in addition to moving the curs or forward.

Using the getXXX Methods

We use the getXXX method of the appropriate type to retrieve the value in each column. For example, the first column in each row of rs is COF_NAME , which stores a value of SQL type VARCHAR . The method for retrieving a value of SQL type VARCHAR is getString . The second column in each row stores a value of SQL type FLOAT , and the method for retrieving values of that type is getFloat . The following code accesses the values stored in the current row of rs and prints a line with the name followed by three spaces and the price. Each time the method next is invoked, the next row becomes the current row, and the loop continues until there are no more rows in rs .

```
String query = "SELECT COF_NAME, PRICE FROM COFFEES";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
   String s = rs.getString("COF_NAME");
   float n = rs.getFloat("PRICE");
   System.out.println(s + "   " + n);
}
```

The output will look something like this:

```
Colombian   7.99
French_Roast   8.99
Espresso   9.99
Colombian_Decaf   8.99
French_Roast_Decaf   9.99
```

15. ITERATE RESULTSET

- **Use of ResultSet.getXXX Methods to Retrieve JDBC Types**
- Following table shows which methods can legally be used to retrieve SQL types and, more important, which methods are recommended for retrieving the various SQL types. Note that this table uses the term "JDBC type" in place of "SQL type." Both terms refer to the generic SQL types defined in java.sql.Types , and they are interchangeable.

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
getByte	X	x	x	x	x	x	x	x	x	x	x	x	x						
getShort	x	X	x	x	x	x	x	x	x	x	x	x	x						
getInt	x	x	X	x	x	x	x	x	x	x	x	x	x						
getLong	x	x	x	X	x	x	x	x	x	x	x	x	x						
getFloat	x	x	x	x	X	x	x	x	x	x	x	x	x						
getDouble	x	x	x	x	x	X	X	x	x	x	x	x	x						
getBigDecimal	x	x	x	x	x	x	x	X	X	x	x	x	x						
getBoolean	x	x	x	x	x	x	x	x	x	X	x	x	x						
getString	x	x	x	x	x	x	x	x	x	x	X	X	x	x	x	x	x	x	x
getBytes														X	X	x			
getDate											x	x	x				X		x
getTime											x	x	x					X	x
getTimestamp											x	x	x				x	x	X

- An "x" indicates that the getXXX method may legally be used to retrieve the given JDBC type.
- An " *X* " indicates that the getXXX method is recommended for retrieving the given JDBC type.

16. SCROLLABLE RESULTSET

One of the new features in the JDBC 2.0 API is the ability to move a result set's cursor backward as well as forward. There are also methods that let you move the cursor to a particular row and check the position of the cursor. Scrollable result sets is used to create a GUI (graphical user interface) tool for browsing result sets. Another use is moving to a row in order to update it.

The following line of code illustrates one way to create a scrollable ResultSet object:

```
Statement stmt =  con.createStatement
(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM COFFEES");
```

The first argument is one of three constants added to the ResultSet API to indicate the type of a ResultSet object:

- *TYPE_FORWARD_ONLY :* creates a *nonscrollable* result set, that is, one in which the cursor moves only forward.
- *TYPE_SCROLL_INSENSITIVE :* creates *scrollable* result set, we can use it to move the cursor around in the result set.
- *TYPE_SCROLL_SENSITIVE* : creates *scrollable* result set, we can use it to move the cursor around in the result set.

The difference between TYPE_SCROLL_INSENSITIVE and TYPE_SCROLL_SENSITIVE is, whether a result set reflects changes that are made to it while it is open and whether certain methods can be called to detect these changes. Generally speaking, a result set that is TYPE_SCROLL_INSENSITIVE does not reflect changes made while it is still open and one that is TYPE_SCROLL_SENSITIVE does.

All three types of result sets will make changes visible if they are closed and then reopened. At this stage, you do not need to worry about the finer points of a ResultSet object's capabilities, and we will go into a little more detail later. Keep in mind, though, the fact that no matter what type of result set you specify, we are always limited by what your DBMS and driver actually provide.

The second argument is one of two ResultSet constants for specifying whether a result set is read-only or updatable:

CONCUR_READ_ONLY : result set is read only.

CONCUR_UPDATABLE : result set is updatable

The point to remember here is that if you specify a type, you must also specify whether it is read-only or updatable. Also, you must specify the type first, and because both parameters are of type int , the compiler will not complain if you switch the order.

Resultset has following methods

1. next : Moves the cursor to the next record. Returns false when the cursor goes beyond the result set.

2. previous: moves the cursor to the previous record. Returns false when the cursor goes beyond the result set.

3. absolute: The method absolute will move the cursor to the row number indicated in the argument passed to it. If the number is positive, the cursor moves the given number from the beginning, so calling absolute(1) puts the cursor on the first row. If the number is negative, the cursor moves the given number from the end, so calling absolute(-1) puts the cursor on the last row.

Ex : The following line of code moves the cursor to the fourth row of srs :

srs.absolute(4);

If srs has 500 rows, the following line of code will move the cursor to row 497:

srs.absolute(-4);

4. relative : we can specify how many rows to move from the current row and also the direction in which to move. A positive number moves the cursor forward the given number of rows; a negative number moves the cursor backward the given number of rows

For example, in the following code fragment, the cursor moves to the fourth row, then to the first row, and finally to the third row:

```
srs.absolute(4); // cursor is on the fourth row
. . .
srs.relative(-3); // cursor is on the first row
. . .
srs.relative(2); // cursor is on the third row
```

5. getRow : The method getRow lets you check the number of the row where the cursor is positioned. For example, you can use getRow to verify the current position of the cursor in the previous example as follows:

```
srs.absolute(4);
int rowNum = srs.getRow(); // rowNum should be 4
srs.relative(-3);
int rowNum = srs.getRow(); // rowNum should be 1
srs.relative(2);
int rowNum = srs.getRow(); // rowNum should be 3
```

6. *isFirst*: returns true if the cursor at the first record

7. *isLast* : returns true if the cursor at the last record.

8. *isBeforeFirst* : returns true if the cursor is before first.

9. *isAfterLast*: returns true if the cursor is after last.