ELECTRONICS AND COMPUTER ENGINEERING

LECTURE NOTES

OBJECT ORIENTED PROGRAMMING THROUGH JAVA

J.B. INSTITUTE OF ENGINEERING AND TECHNOLOGY

(UGC AUTONOMOUS)

Bhaskar Nagar, Moinabad Mandal , R.R. District, Hyderabad -500075

<p style="text-align:center;">**UNIT –I**</p>

<p style="text-align:center;">**I. INTRODUCTION TO JAVA**</p>

What is Java

Java is a high Level programming language and it is also called as a platform. Java is a secured and robust high level object-oriented programming language.

Platform: Any software or hardware environment in which a program runs is known as a platform. Java has its own runtime environment (JRE) and API so java is also called as platform.

Java fallows the concept of **Write Once, Run Anywhere.**

Application of java

1. Desktop Applications
2. Web Applications
3. Mobile
4. Enterprise Applications
5. Smart Card
6. Embedded System
7. Games
8. Robotics etc

History of Java

James Gosling, Patrick Naughton and Mike Sheridan initiated the Java language project in 1991. Team of sun engineers designed for small, embedded systems in electronic appliances like set-top boxes. Initially it was called "Greentalk" later it was called Oak .

Java is an open source software produced by Sunmicro system under the terms of the GNU General Public License (GPL) .

Features of Java:

- **Object Oriented** – Java implements basic concepts of Object oriented programming System (OOPS) ie Object, Class, Inheritance, Polymorphism, Abstraction, Encapsulation. In Java, everything is an Object. Java can be easily extended since it is based on the Object model.
- **Platform Independent** – Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on.
- **Simple** – Java fallows the basic Syntax of C,C++. If you understand the basic concept of OOPS then it is easy to master in java.
- **Secure** – With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.
- **Architecture-neutral** – Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.
- **Portable** – Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary, which is a POSIX subset.
- **Robust** – Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
- **Multithreaded** – With Java's multithreaded feature In java we can write programs that can perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that can run smoothly.
- **Interpreted** – Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.
- **High Performance** – With the use of Just-In-Time compilers, Java enables high performance.
- **Distributed** – Java is designed for the distributed environment of the internet.
- **Dynamic** – Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java

programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

Object Oriented Programming System(OOPS)

Object means a real word entity such as pen, chair, table etc. Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

If any language fallows the OOPS concepts that language we call it as object oriented language

Procedure to write simple java Program

To write a java program First we have install the JDK.

To create a simple java program, you need to create a class that contains main method. Let's understand the requirement first.

- install the JDK and install it.
- set path of the jdk
- create the java program
- compile and run the java program

Setting Up the Path for Windows

Assuming you have installed Java in *c:\Program Files\java\jdk* directory −

- Right-click on 'My Computer' and select 'Properties'.
- Click the 'Environment variables' button under the 'Advanced' tab.
- Now, alter the 'Path' variable so that it also contains the path to the Java executable. Example, if the path is currently set to 'C:\WINDOWS\SYSTEM32', then change your path to read 'C:\WINDOWS\SYSTEM32;c:\Program Files\java\jdk\bin'.

Setting Up the Path for Linux, UNIX, Solaris, FreeBSD

Environment variable PATH should be set to point to where the Java binaries have been installed. Refer to your shell documentation, if you have trouble doing this. For Example if you use *bash* as your shell, then you would add the following line to the end of your '.bashrc: export PATH = /path/to/java:$PATH'

Popular Java Editors

- **Notepad** – On Windows machine, you can use any simple text editor like Notepad (Recommended for this tutorial), TextPad.
- **Netbeans** – A Java IDE that is open-source and free which can be downloaded from **Eclipse** – A Java IDE developed by the eclipse open-source community and can be downloaded from

JVM (Java Virtual Machine)

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 Sun and other companies.
2. **An implementation** Its implementation is known as JRE (Java Runtime Environment).
3. **Runtime Instance** Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

What it does

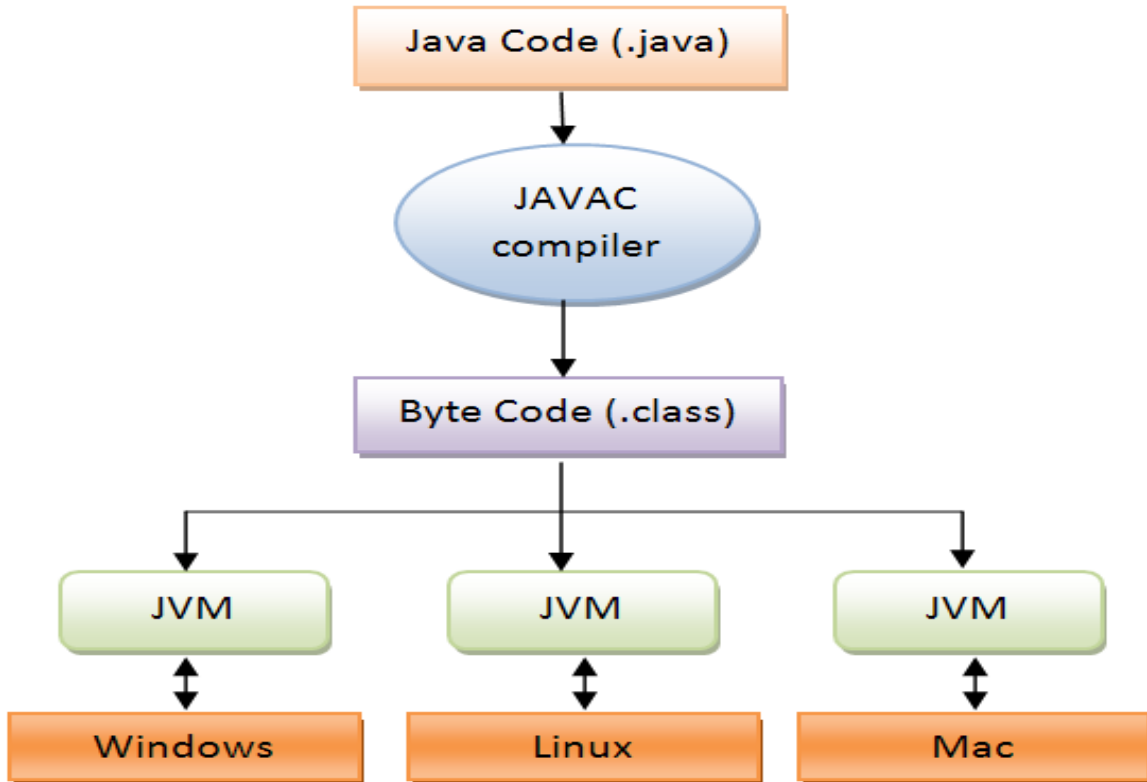The JVM performs following operation:

- 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.

JAVA VIRTUAL MACHINE

Java's Magic: The Bytecode output of a Java compiler is not executable code. Rather, it is bytecode. Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM).

Java complier translates the *java source* code into *byte code* or intermediate code ,not the executable file .JVM take the byte code and convert into executable code corresponding to Operating system

Because of the above feature java is portable

II.CLASS, OBJECT AND METHODS

Java program is a collection of objects that communicate via invoking each other's methods. We now briefly look into class, object, methods, and instance variables.

Class – A class can be defined as a template/blueprint that describes the behavior/state that the object of its type supports.

A class is declared by use of the class keyword. A simplified general form of a class definition is shown here:

class classname

{

 type instance-variable1; type instance-variable2;

 // ... type instance-variableN;

type methodname1(parameter-list)

{ // body of method }

type methodname2(parameter-list)

 { // body of method }

// ... type methodnameN(parameter-list)

    { // body of method } }

The data, or variables, defined within a class are called instance variables. The code is contained within methods. Collectively, the methods and variables defined within a class are called members of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class

Simple Class

Class Sample

{

 int len,   float ht

 void get()

 { // body

```
}  }
```

Here a class Sample contains two variable len and ht

Object in Java

Object is the physical as well as logical entity whereas class is the logical entity only.

An object has three characteristics:

- **state:** represents data (value) of an object.
- **behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.
- **identity:** Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each object uniquely.

Object is an instance of a class. Class is a template or blueprint from which objects are created. So object is the instance(result) of a class.

Object Definitions:

- Object is *a real world entity*.   Object is *a run time entity*.
- Object is *an entity which has state and behavior*.
- Object is *an instance of a class*.

Sample s=new Sample()   here s is an object for the class Sample

new operator is used to create an object

Methods – A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.

Let's create the Simple java program:

```
1. class Sample{
2.    public static void main(String args[]){
3.     System.out.println("How are you ");
4.    }
5. }
```

save this file as Sample.java

To compile: javac Sample.java

To execute: java Sample

Java Identifiers

All Java components require names. Names used for classes, variables, and methods are called identifiers.

In Java, there are several points to remember about identifiers. They are as follows −

- All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (_).
- After the first character, identifiers can have any combination of characters.
- A key word cannot be used as an identifier.
- Most importantly, identifiers are case sensitive.
- Examples of legal identifiers: age, $salary, _value, __1_value.
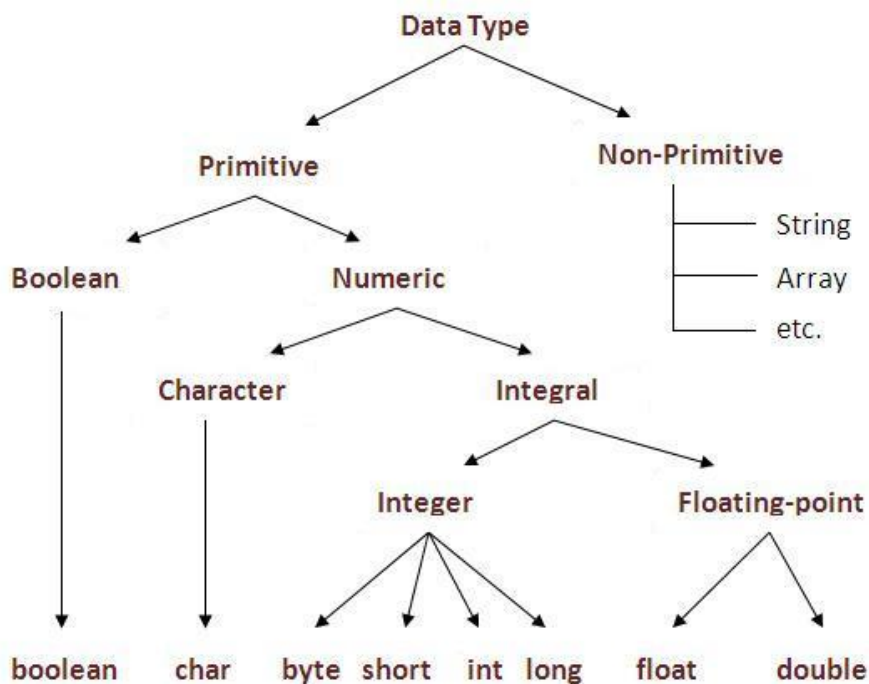- Examples of illegal identifiers: 123abc, -salary.

Java Modifiers: There are two categories of modifiers −

- **Access Modifiers** − default, public , protected, private
- **Non-access Modifiers** − final, abstract, strictfp

III DATA TYPES

There are two data types available in Java −

- Primitive Data Types
- Non Primitive Types

Primitive Data Types

There are eight primitive data types supported by Java. Primitive data types are predefined by the language and named by a keyword. Let us now look into the eight primitive data types in detail.

byte

- Byte data type is an 8-bit signed two's complement integer
- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive)(2^7 -1)
- Default value is 0
- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an integer.
- Example: byte a = 100, byte b = -50

short

- Short data type is a 16-bit signed two's complement integer
- Minimum value is -32,768 (-2^{15})
- Maximum value is 32,767 (inclusive) (2^{15} -1)
- Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an integer
- Default value is 0.
- Example: short s = 10000, short r = -20000

int

- Int data type is a 32-bit signed two's complement integer.
- Minimum value is - 2,147,483,648 (-2^31)
- Maximum value is 2,147,483,647(inclusive) (2^31 -1)
- Integer is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0
- Example: int a = 100000, int b = -200000

long

- Long data type is a 64-bit signed two's complement integer
- Minimum value is -9,223,372,036,854,775,808(-2^63)
- Maximum value is 9,223,372,036,854,775,807 (inclusive)(2^63 -1)
- This type is used when a wider range than int is needed
- Default value is 0L
- Example: long a = 100000L, long b = -200000L

float

- Float data type is a single-precision 32-bit IEEE 754 floating point
- Float is mainly used to save memory in large arrays of floating point numbers
- Default value is 0.0f
- Float data type is never used for precise values such as currency
- Example: float f1 = 234.5f

double

- double data type is a double-precision 64-bit IEEE 754 floating point
- This data type is generally used as the default data type for decimal values, generally the default choice
- Double data type should never be used for precise values such as currency
- Default value is 0.0d
- Example: double d1 = 123.4

boolean

- boolean data type represents one bit of information
- There are only two possible values: true and false
- This data type is used for simple flags that track true/false conditions
- Default value is false
- Example: boolean one = true

char

- char data type is a single 16-bit Unicode character

- Minimum value is '\u0000' (or 0)
- Maximum value is '\uffff' (or 65,535 inclusive)
- Char data type is used to store any character
- Example: char letterA = 'A'

Java Literals

A literal is a source code representation of a fixed value. Literals can be assigned to any primitive type variable.

byte a = 68;   char a = 'A'

byte, int, long, and short can be expressed in decimal(base 10), hexadecimal(base 16) or octal(base 8) number systems as well.

Prefix 0 is used to indicate octal, and prefix 0x indicates hexadecimal when using these number systems for literals. For example −

int decimal = 100;   int octal = 0144;   int hexa = 0x64;

String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are −

Example

"Hello World"   "two\nlines"   "\"This is in quotes\""

String and char types of literals can contain any Unicode characters. For example −   char a = '\u0001';   String a = "\u0001";

Java language supports few special escape sequences for String and char

Notation	Character represented
\n	Newline (0x0a)
\r	Carriage return (0x0d)
\f	Formfeed (0x0c)
\b	Backspace (0x08)
\s	Space (0x20)
\t	Tab
\"	Double quote

\'	Single quote
\\	Backslash
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)

Java Variable Example: Add Two Numbers

```
1. class Sample{
2. public static void main(String[] args){
3. int i=50;
4. int j=60;
5. int k=a+b;
6. System.out.println(k);
7. } }
```

Java Variable Example: Widening

```
1. class Sample{
2. public static void main(String[] args){
3. int j=10;
4. float k=a;
5. System.out.println(i);
6. System.out.println(j);
7. }}
```

Unicode System

Unicode is a universal international standard character encoding that is capable of representing most of the world's written languages.

Before Unicode, there were many language standards:

- **ASCII** (American Standard Code for Information Interchange) for the United States.
- **ISO 8859-1** for Western European Language.
- **KOI-8** for Russian.
- **GB18030 and BIG-5** for chinese, and so on.

Java Tokens

Java Tokens are the smallest individual building block or smallest unit of a Java program, it is used by the Java compiler for constructing expressions and statements. Java program is collection different types of tokens, comments, and white spaces.

Java Supports Five Types of Tokens:

- Reserved Keywords   Identifiers   Literals
- Operators   Separators

Java Keywords can not be used as a variable name.

Abstract	Assert	boolean	break
Byte	Case	catch	char
Class	Const	continue	default
Do	Double	else	enum
extends	Final	finally	float
For	Goto	if	implements
Import	Instanceof	int	interface
Long	Native	new	package
private	Protected	public	return
Short	Static	strictfp	super
Switch	synchronized	this	throw
throws	Transient	try	void
volatile	While	true	false
Null			

Variable

Variable is name of *reserved area allocated in memory*. In other words, it is a *name of memory location*. It is a combination of "vary + able" that means its value can be changed.

There are three types of variables in java:

- local variable
- instance variable
- static variable
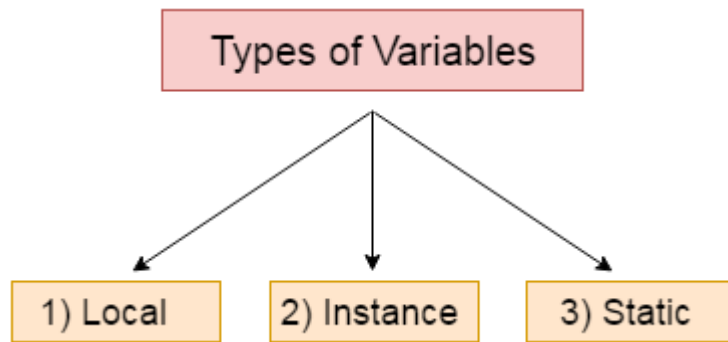
1) Local Variable

A variable which is declared inside the method is called local variable.

2) Instance Variable

A variable which is declared inside the class but outside the method, is called instance variable . It is not declared as static.

3) Static variable

A variable that is declared as static is called static variable. It cannot be local.

We will have detailed learning of these variables in next chapters.

IV OPERATORS & IF, Switch, loop Statements

Operators in java

Operator in java is a symbol that is used to perform operations. For example: +, -, *, / etc.

There are many types of operators in java which are given below:

- Unary Operator,
- Arithmetic Operator,
- shift Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator and
- Assignment Operator.

Java If-else Statement

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 ladder   nested if statement

Java IF Statement

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

1. if(condition){
2. //code to be executed
3. }

1. public class Example {
2. public static void main(String[] args) {
3.   int k=35;
4.   if(k>18){   System.out.print("Hello");
5.   } } }

IF-else Statement

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

Syntax:

1. if(condition){
2. //code if condition is true
3. }else{
4. //code if condition is false
5. }

```
public class Sample {
public static void main(String[] args) {
   int n=23;
   if(number%2==0){
      System.out.println("even ");
   }else{
      System.out.println("odd ");
   }  }
}
```

IF-else-if ladder Statement

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

Syntax:

```
1.  if(condition1){
2.  //code to be executed if condition1 is true
3.  }else if(condition2){
4.  //code to be executed if condition2 is true
5.  }
6.  else if(condition3){
7.  //code to be executed if condition3 is true
8.  }
9.  ...
10.      else{
11.      //code to be executed if all the conditions are false
12.      }
```

```
1.  public class Simple {
2.  public static void main(String[] args) {
3.      int marks=70;
4.
5.      if(marks<40){
6.          System.out.println("FAIL");
7.      }
8.      else if(marks>=40 && marks<50){
9.          System.out.println("D grade");
10.         }
11.          else if(marks>=50 && marks<60){
12.              System.out.println("C grade");
13.          }
14.          else if(marks>=60 && marks<70){
15.              System.out.println("B grade");
16.          }
17.          else if(marks>=70 && marks<80){
18.              System.out.println("A grade");
19.          }else if(marks>=80 && marks<100){
20.              System.out.println("A+ grade");
21.          }else{
22.              System.out.println("Invalid!");
23.          }
24.      }
25.      }
```

Switch Statement

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

Syntax:

1. switch(expression){
2. case value1:
3.   //code to be executed;
4.   break; //optional
5. case value2:
6.   //code to be executed;
7.   break; //optional
8. ......
9.
10.   default:
11.   code to be executed if all cases are not matched;
12.   }

Example:

1. public class Sample {
2. public static void main(String[] args) {
3.   int k=20;
4.   switch(k){
5.   case 10: System.out.println("10");break;
6.   case 20: System.out.println("20");break;
7.   case 30: System.out.println("30");break;
8.   default:System.out.println("Not in 10, 20 or 30");
9.   } }
10.   }

Java For Loop

The Java *for loop* is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop.

There are three types of for loop in java.

- Simple For Loop
- For-each or Enhanced For Loop
- Labeled For Loop

Java Simple For Loop

The simple for loop is same as C/C++. We can initialize variable, check condition and increment/decrement value.

Syntax:

1. for(initialization;condition;incr/decr){
2. //code to be executed
3. }

Example:

1. public class Sample {
2. public static void main(String[] args) {
3.   for(int i=1;i<=20;i++){
4.   System.out.println(i);
5.   }
6. }
7. }

Java While Loop

The Java *while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.

Syntax:

1. while(condition){
2. //code to be executed
3. }

1. public class Sample {
2. public static void main(String[] args) {
3.   int j=1;
4.   while(j<=10){
5.   System.out.println(j);
6.   j++;
7.   }
8. } }

Java do-while Loop

The Java *do-while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.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);

   public class Example {

   public static void main(String[] args) {

       int j=1;

       do{   System.out.println(j);

          j++;

       }while(j<=10);

          } }

Java Break Statement

The Java *break* is used to break loop or switch statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop.

Example:

```
1. public class Simple {
2. public static void main(String[] args) {
3.     for(int i=1;i<=10;i++){
4.         if(i==5){
5.             break;
6.         }
7.         System.out.println(i);
8.     }
9. }
10.        }
```

Java Continue Statement

The Java *continue statement* is used to continue loop. It continues the current flow of the program and skips the remaining code at specified condition. In case of inner loop, it continues only inner loop.

Example:

```
1. public class Sample {
```

```
2.  public static void main(String[] args) {
3.     for(int k=1;k<=10;k++){
4.        if(k==5){
5.           continue;
6.        }
7.        System.out.println(k);
8.     }
9. }
10.       }
```

V ARRAYS & COMMENTS in JAVA

Array in java

Array is group of elements of similar types occupying contiguous memory locations

Advantage of Java Array

- **Code Optimization   Random access**

    Disadvantage of Java Array is it has fixed size it cannot grow

There are two types of array in java .

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in java
Syntax to Declare an Array in java

1. datatype[] arrayname (or)
2. datatype []arrayname; (or)
3. datatype arrayname[];

Array initialization in java

1. arrayname=new datatype[size];

Example

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

```
1. class Testarray{
2. public static void main(String args[]){
3. int a[]=new int[5];//declaration and instantiation
4. a[0]=10;    a[1]=20;   a[2]=70;   a[3]=40;  a[4]=50;
5.   for(int i=0;i<a.length;i++)//length is the property of array
6. System.out.println(a[i]);
7.   }}
```

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

```
1. int a[]={33,3,4,5};//declaration, instantiation and initialization
```

Let's see the simple example to print this array.

```
1. class Testarray1{
2. public static void main(String args[]){
3.
4. int a[]={33,3,4,5};//declaration, instantiation and initialization
5.
6. //printing array
7. for(int i=0;i<a.length;i++)//length is the property of array
8. System.out.println(a[i]);
9.
10.      }}
```

Multidimensional array in java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in java

```
1. dataType[][] arrayRefVar; (or)  dataType [][]arrayRefVar; (or)
2. dataType arrayRefVar[][]; (or)   dataType []arrayRefVar[];
```

Example to instantiate Multidimensional Array in java

```
1. int[][] arr=new int[3][3];//3 row and 3 column
```

Example to initialize Multidimensional Array in java

```
1. arr[0][0]=1;
2. arr[0][1]=2;
3. arr[0][2]=3;
4. arr[1][0]=4;
5. arr[1][1]=5;
6. arr[1][2]=6;
```

7. arr[2][0]=7;
8. arr[2][1]=8;
9. arr[2][2]=9;

Example of Multidimensional java array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

```
1.  class Ex{
2.  public static void main(String args[]){
3.
4.  //declaring and initializing 2D array
5.  int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
6.
7.  //printing 2D array
8.  for(int i=0;i<3;i++){
9.   for(int j=0;j<3;j++){
10.        System.out.print(arr[i][j]+" ");
11.       }
12.       System.out.println();
13.      }
14.       }  }
```

Java Comments

The java comments are statements that are not executed by the compiler and interpreter. The comments can be used to provide information or explanation about the variable, method, class or any statement. It can also be used to hide program code for specific time.

Types of Java Comments

There are 3 types of comments in java.

1. Single Line Comment
2. Multi Line Comment
3. Documentation Comment

1) Java Single Line Comment

The single line comment is used to comment only one line.

Syntax:

1. //This is single line comment

Example:

1. public class Sample{
2. public static void main(String[] args) {
3.   int j=10;//Here, i is a variable
4.   System.out.println(j);
5. }
6. }

2) Java Multi Line Comment

The multi line comment is used to comment multiple lines of code.

Syntax:

1. /*
2. This
3. is
4. multi line
5. comment
6. */

Example:

1. public class CommentExample2 {
2. public static void main(String[] args) {
3. /* Let's declare and
4. print variable in java. */
5.   int j=10;
6.   System.out.println(j);
7. }
8. }

3) Java Documentation Comment

The documentation comment is used to create documentation API. To create documentation API, you need to use **javadoc tool**.

Syntax:

1. /**
2. This
3. is
4. documentation
5. comment

6. */

VI CONSTRUCTORS

Constructor is special member function ,it has the same name as class name. It is called when an instance of object is created and memory is allocated for the object.

It is a special type of method which is used to initialize the object

Rules for creating java constructor

There are basically two rules defined for the constructor.

1. Constructor name must be same as its class name
2. Constructor must have no explicit return type

Types of java constructors

There are two types of constructors in java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

Types of Java Constructor

Default Constructor   Parameterized Constructor
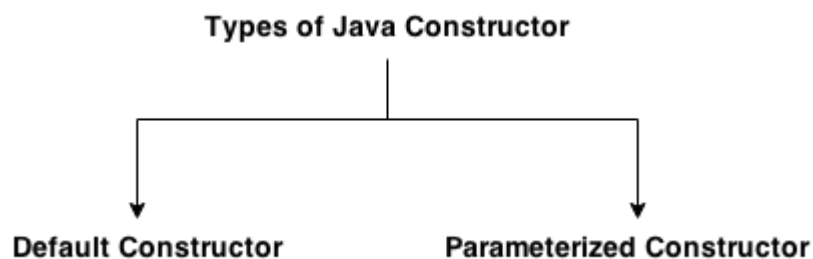
A constructor is called "Default Constructor" when it doesn't have any parameter.

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

A constructor which has a specific number of parameters is called parameterized constructor.

```
1. class Student4{
2.    int id;
3.    String name;
4.    Student4(int i,String n){
5.    id = i;
6.    name = n;
7.    }
8.    void display(){System.out.println(id+" "+name);}
9.
10.        public static void main(String args[]){
11.        Student4 s1 = new Student4(111,"Karan");
12.        Student4 s2 = new Student4(222,"Aryan");
13.        s1.display();
14.        s2.display();
15.      }
16.    }
```

Difference between constructor and method in java

There are many differences between constructors and methods. They are given below.

Java Constructor	Java Method
Constructor is used to initialize the state of an object.	Method is used to expose behaviour of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name may or may not be same as class name.

Java static keyword

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

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)

3. block
4. nested class

1) Java static variable

If you declare any variable as static, it is known static variable.

- The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees,college name of students etc.
- The static variable gets memory only once in class area at the time of class loading.

Advantage of static variable: It makes your program memory efficient (i.e it saves memory).

```
class Stud{
 int rollno;
 String name;
 static String college ="ITS";
 Stud(int r,String n){
 rollno = r;
 name = n;
 }
void display (){System.out.println(rollno+" "+name+" "+college);}

public static void main(String args[]){
Stud s1 = new Stud(11,"Krishna");
Stud s2 = new Stud(22,"Rama");
 s1.display();
s2.display();
 }    }
```

2) Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- static method can access static data member and can change the value of it.

Example of static method

```
1. //Program of changing the common property of all objects(static field).
2.
3. class Stud{
4.     int rollno;
5.     String name;
6.     static String college = "BEC";
7.
8.     static void change(){
9.     college = "JBIEIT";
10.        }
11.        Stud(int r, String n){
12.        rollno = r;
13.        name = n;
14.        }
15.
16.        void display (){System.out.println(rollno+" "+name+" "+college
   );}
17.        public static void main(String args[]){
18.        Stud.change();
19.        Stud s1 = new Stud (11,"Kiran");
20.        Stud s2 = new Stud (22,"Arjun");
21.        Stud s3 = new Stud (33,"srinu");
22.        s1.display();
23.        s2.display();
24.        s3.display();
25.        }
26.     }
```

this keyword in java

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

Usage of java this keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

1) 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.

```
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();   }}
```

2) this: to invoke current class method

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method

```
1.  class B{
2.  void m(){System.out.println("hello m");}
3.  void n(){
4.  System.out.println("hello n");
5.  //m();//same as this.m()
6.  this.m();
7.  }
8.  }
9.  class TestThis4{
10.       public static void main(String args[]){
11.       B b=new B();
12.       b.n();
13.       }}
```

3) this() : to invoke current class constructor

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

Calling default constructor from parameterized constructor:

```
1. class B{
2. B(){System.out.println("hello ");}
3. B(int x){
4. this();
5. System.out.println(x);
6. }
7. }
8. class Sample{
9. public static void main(String args[]){
10.      B a=new B(10);
11.      }}
```

```
1. class Student{
2. int rollno;
3. String name,course;
4. float fee;
5. Student(int rollno,String name,String course){
6. this.rollno=rollno;
7. this.name=name;
8. this.course=course;
9. }
10.      Student(int rollno,String name,String course,float fee){
11.      this(rollno,name,course);//reusing constructor
12.      this.fee=fee;
13.      }
14.      void display(){System.out.println(rollno+" "+name+" "+course+" "
   +fee);}
15.      }
16.      class SamplTest{
17.      public static void main(String args[]){
18.      Student s1=new Student(111,"ankit","java");
19.      Student s2=new Student(112,"sumit","java",6000f);
20.      s1.display();
21.      s2.display();
22.      }}
```

4) this: to pass as an argument in the method

The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

```
1. class S2{
2.   void m(S2 obj){
3.   System.out.println("method is invoked");
4.   }
5.   void p(){
6.   m(this);
```

```
7.    }
8.    public static void main(String args[]){
9.    S2 s1 = new S2();
10.         s1.p();
11.         }
12.         }
```

5) this: to pass as argument in the constructor call

We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes. Let's see the example:

```
1.  class B{
2.    A4 obj;
3.    B(A4 obj){
4.      this.obj=obj;
5.    }
6.    void display(){
7.      System.out.println(obj.data);//using data member of A4 class
8.    }
9.  }
10.
11.        class A4{
12.          int data=10;
13.          A4(){
14.            B b=new B(this);
15.            b.display();
16.          }
17.          public static void main(String args[]){
18.            A4 a=new A4();
19.          }
20.        }
```

6) this keyword can be used to return current class instance

We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example:

Syntax of this that can be returned as a statement

```
1.  return_type method_name(){
2.  return this;
3.  }
```

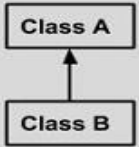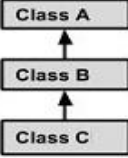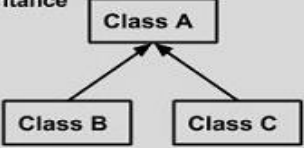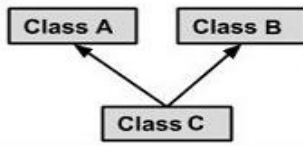**Example of this keyword that you return as a statement from the method**

1. class A{
2. A getA(){
3. return this;
4. }
5. void msg(){System.out.println("Hello java");}
6. }
7. class Test1{
8. public static void main(String args[]){
9. new A().getA().msg();
10.   }
11.   }

UNIT -II

VII. Inheritance

Inheritance can be defined as the procedure or mechanism of acquiring all the properties and behavior of one class to another, i.e. acquiring the properties and behavior of child class from the parent class. This concept was built in order to achieve the advantage of creating a new class that gets built upon an already existing class(es). It is mainly used for code reusability within a Java program. The class that gets inherited taking the properties of another class is the subclass or derived class or child class. Again, the class whose properties get inherited is the superclass or base class or parent class. The keyword **extends** is used to inherit the properties of the base class to derived class. The structure of using this keyword looks something like this:

```
class base
{
    .....
    .....
}
class derive extends base
 {
    .....
    .....
}
```

Single Inheritance	Class A ↑ Class B	public class A {
		....
		}
		public class B extends A {
		....
		}

Multi Level Inheritance	Class A ↑ Class B ↑ Class C	public class A { ....} public class B extends A {...................} public class C extends B {................... }
Hierarchical Inheritance	Class A ↗ ↖ Class B Class C	public class A { ....} public class B extends A {...................} public class C extends A {................... }

Multiple Inheritance	Class A Class B ↖ ↗ Class C	public class A { ....}
		
		public class B {...................}
		
		public class C extends A,B {
		....
		} // Java does not support mutiple Inheritance

```
class Person {
 void teach() {
  System.out.println("Teaching subjects");
 } }

class Person extends Teacher {
 void listen() {
  System.out.println("Listening to teacher");
 } }

class CheckForInheritance {
 public static void main(String args[]) {
Person s1 = new Students();
  s1.teach();
  s1.listen();
 }
}
```

In this type of inheritance, a derived class gets created from another derived class and can have any number of levels.

```
class Teacher {
 void teach() {
  System.out.println("Teaching subject");
 }
}
class Student extends Teacher {
 void listen() {
```

```
  System.out.println("Listening");
 }
}
class homeTution extends Student {
 void explains() {
  System.out.println("Does homework");
 }
}
class CheckForInheritance {
 public static void main(String argu[]) {
  homeTution h = new himeTution();
  h.explains();
  d.teach();
  d.listen();
 }
}
```

In this type of inheritance, there are more than 1 derived classes which get created from one single base class.

```
class Teacher {
 void teach() {
  System.out.println("Teaching subject");
 }
}
class Student extends Teacher {
 void listen() {
  System.out.println("Listening");
 }
}
class Principal extends Teacher {
 void evaluate() {
  System.out.println("Evaluating");
 }
}
class CheckForInheritance {
 public static void main(String argu[]) {
  Principal p = new Principal();
  p.evaluate();
  p.teach();
  // p.listen(); will produce an error
 }
}
```

Let us imagine a situation where there are three classes: A, B and C. The C class inherits A and B classes. In case, class A and class B have a method with same name and type and as a programmer, you have to call that

method from child class's (C) object, there-there will be ambiguity as which method will be called either of A or of B class.

So Java reduces this hectic situation by the use of interfaces which implements this concept and reduce this problem; as compile-time errors are tolerable than runtime faults in the program.

VIII Polymorphism

The word polymorphism means having multiple forms. The term Polymorphism gets derived from the Greek word where *poly + morphos* where *poly* means many and *morphos* means forms.

Polymorphism is another special feature of object-oriented programming (OOPs). The approach which lies beneath this concept is "single interface with multiple implementations." This offers a single interface for controlling access to a general class of actions.

Polymorphism can be achieved in two of the following ways:

- **Method Overloading**(Compile time Polymorphism)
- **Method Overriding**(Run time Polymorphism)

- Static Polymorphism is in other words termed as compile-time binding or early binding.
- Static binding occurs at compile time. Method overloading is a case of static binding and in this case binding of method call to its definition happens at the time of compilation.

- To call an overloaded method in Java, it is must use the type and/or the number of arguments to determine which version of the overloaded method to actually call.
- The overloaded methods may have varied return types and the return type single-handedly is insufficient to make out two versions of a method.
- As and when Java compiler encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.
- It permits the user to obtain compile time polymorphism with name method name.
- An overloaded method is able to throw different kinds of exceptions.
- A method which is overloaded can contain different access modifiers.

Overloading method's argument lists might differ in:

- Number of parameters passed
- Data type of actual parameters
- Sequence of data type of actual parameters

```java
class Mltply {
 void mul(int a, int b) {
  System.out.println("Sum of two=" + (a * b));
 }

 void mul(int a, int b, int c) {
  System.out.println("Sum of three=" + (a * b * c));
 }
}
class Polymorphism {
 public static void main(String args[]) {
  Mltply m = new Mltply();
  m.mul(6, 10);
  m.mul(10, 6, 5);
 } }
```

Rules to method overriding

- **Argument list**: The argument list at the time of overriding method need to be same as that of the method of the parent class. The data types of the arguments along with their sequence must have to be preserved as it is in the overriding method.
- **Access Modifier**: The Access Modifier present in the overriding method (method of subclass) cannot be more restrictive than that of an overridden method of the parent class.
- The private, static and final methods can't be overridden as they are local to the class.
- Any method which is overriding is able to throw any unchecked exceptions, in spite of whether the overridden method usually method of parent class might throw an exception or not.

```java
//method overriding
class parent {
 public void work() {
  System.out.println("Parent is under retirement from work.");
 }
}
class child extends parent {
 public void work() {
  System.out.println("Child has a job");
  System.out.println(" He is doing it well");
 }
 public static void main(String argu[]) {
  child c1 = new child();
  c1.work();
 } }
```

Advantage of method overriding

One major advantage of method overriding is that a class can give its own specific execution to an inherited method without having the modification in the parent class (base class).

super keyword in java

The **super** keyword in java is a reference variable which is used to refer immediate parent class object.

Whenever you 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.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

1) 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.

1. class Animal{
2. String color="white";
3. }
4. class Dog extends Animal{
5. String color="black";
6. void printColor(){
7. System.out.println(color);//prints color of Dog class
8. System.out.println(super.color);//prints color of Animal class
9. }
10.   }
11.   class TestSuper1{
12.   public static void main(String args[]){
13.   Dog d=new Dog();
14.   d.printColor();
15.   }}

2) super can be used to invoke parent class method

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

1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void eat(){System.out.println("eating bread...");}
6. void bark(){System.out.println("barking...");}
7. void work(){
8. super.eat();
9. bark();
10.   }
11.   }
12.   class TestSuper2{
13.   public static void main(String args[]){
14.   Dog d=new Dog();
15.   d.work();
16.   }}

3) super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

1. class Animal{
2. Animal(){System.out.println("animal is created");}
3. }
4. class Dog extends Animal{
5. Dog(){
6. super();
7. System.out.println("dog is created");
8. }
9. }
10.   class TestSuper3{
11.   public static void main(String args[]){
12.   Dog d=new Dog();
13.   }}

1. class Person{
2. int id;
3. String name;
4. Person(int id,String name){
5. this.id=id;

6. this.name=name;
7. }
8. }
9. class Emp extends Person{
10.   float salary;
11.   Emp(int id,String name,float salary){
12.   super(id,name);//reusing parent constructor
13.   this.salary=salary;
14.   }
15.   void display(){System.out.println(id+" "+name+" "+salary);}
16.   }
17.   class TestSuper5{
18.   public static void main(String[] args){
19.   Emp e1=new Emp(1,"ankit",45000f);
20.   e1.display();
21.   }}

Method Overloading in Java

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

Advantage of method overloading

Method overloading *increases the readability of the program.*

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

1) 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.

In this example, we are creating static methods so that we don't need to create instance for calling methods.

1. class Adder{
2. static int add(int a,int b){return a+b;}
3. static int add(int a,int b,int c){return a+b+c;}
4. }

5. class TestOverloading1{
6. public static void main(String[] args){
7. System.out.println(Adder.add(11,11));
8. System.out.println(Adder.add(11,11,11));
9. }}

2) 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.

1. class Adder{
2. static int add(int a, int b){return a+b;}
3. static double add(double a, double b){return a+b;}
4. }
5. class TestOverloading2{
6. public static void main(String[] args){
7. System.out.println(Adder.add(11,11));
8. System.out.println(Adder.add(12.3,12.6));
9. }}

Method Overriding in Java

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 subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. method must have same name as in the parent class
2. method must have same parameter as in the parent class.
3. must be IS-A relationship (inheritance).

1. class Vehicle{
2. void run(){System.out.println("Vehicle is running");}
3. }
4. class Bike2 extends Vehicle{
5. void run(){System.out.println("Bike is running safely");}
6.
7. public static void main(String args[]){
8. Bike2 obj = new Bike2();
9. obj.run();
10.   }

IX ABSTRACTION

Abstraction in Java

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

Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

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

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

Abstract class in Java

A class that is declared as abstract is known as **abstract class**. It needs to be extended and its method implemented. It cannot be instantiated.

Example abstract class

1. abstract class A{}

abstract method

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

known as abstract method.

Example abstract method

1. abstract void printStatus();//no body and abstract

Example of abstract class that has abstract method

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

```
1.  abstract class Bike{
2.   abstract void run();
3.  }
4.  class Honda4 extends Bike{
5.  void run(){System.out.println("running safely..");}
6.  public static void main(String args[]){
7.   Bike obj = new Honda4();
8.   obj.run();
9.  }
10.       }
```

```
1.  abstract class Shape{
2.  abstract void draw();
3.  }
4.  //In real scenario, implementation is provided by others i.e. unknown
    by end user
5.  class Rectangle extends Shape{
6.  void draw(){System.out.println("drawing rectangle");}
7.  }
8.  class Circle1 extends Shape{
9.  void draw(){System.out.println("drawing circle");}
10.       }
11.       //In real scenario, method is called by programmer or user
12.       class TestAbstraction1{
13.       public static void main(String args[]){
14.       Shape s=new Circle1();//In real scenario, object is provided thro
    ugh method e.g. getShape() method
15.       s.draw();
16.       }
17.       }
```

Interface in Java

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**. There can be only abstract methods in the java interface not method body. It is used to achieve abstraction and multiple inheritance in Java.

Java Interface also **represents IS-A relationship**.

It cannot be instantiated just like abstract class.

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

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

Java Interface Example

In this example, Printable interface has only one method, its implementation is provided in the A class.

```
1.  interface printable{
2.  void print();
3.  }
4.  class A6 implements printable{
5.  public void print(){System.out.println("Hello");}
6.
7.  public static void main(String args[]){
8.  A6 obj = new A6();
9.  obj.print();
10.        }
11.        }
```

Differences between abstract class and interface that are given below.

Abstract class	Interface
1) Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods. Since Java 8, it can have default and static methods also.
2) Abstract class doesn't support multiple inheritance.	Interface supports multiple inheritance.
3) Abstract class can have final,	Interface has only static and final

non-final, static and non-static variables.	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)Example:
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

1. interface A{
2. void a(); void b();
3. void c(); void d();
4. }
5. abstract class B implements A{
6. public void c(){System.out.println("I am c");}
7. }
8. class M extends B{
9. public void a(){System.out.println("I am a");}
10. public void b(){System.out.println("I am b");}
11. public void d(){System.out.println("I am d");}
12. }
13. class Test5{
14. public static void main(String args[]){
15. A a=new M();
16. a.a();
17. a.b();
18. a.c();
19. a.d();
20. }}

X PACKAGE

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.

Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.

Simple example of java package

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

1. //save as Simple.java
2. package mypack;
3. public class Simple{
4. public static void main(String args[]){
5.   System.out.println("Welcome to package");
6. }
7. }

How to compile java package

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

1. javac -d directory javafilename

For **example**

1. javac -d . Simple.java

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

How to access package from another package?

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

1. import package.*;
2. import package.classname;
3. fully qualified name.

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

1. //save by A.java
2. package pack;
3. public class A{
4.   public void msg(){System.out.println("Hello");}
5. }

1. //save by B.java
2. package mypack;
3. import pack.*;
4.
5. class B{
6.   public static void main(String args[]){
7.   A obj = new A();
8.   obj.msg();
9.   }
10.   }

2) Using packagename.classname

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

Example of package by import package.classname

1. //save by A.java
2.
3. package pack;
4. public class A{
5.   public void msg(){System.out.println("Hello");}
6. }

1. //save by B.java
2. package mypack;
3. import pack.A;
4.
5. class B{
6.   public static void main(String args[]){
7.   A obj = new A();
8.   obj.msg();
9.   }
10.   }

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

1. //save by A.java
2. package pack;
3. public class A{
4.   public void msg(){System.out.println("Hello");}
5. }
6.

1. //save by B.java
2. package mypack;
3. class B{
4.   public static void main(String args[]){
5.   pack.A obj = new pack.A();//using fully qualified name
6.   obj.msg();
7.   }
8. }

Access Modifiers in java

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:

1. private
2. default
3. protected
4. public

Understanding all java access modifiers

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

XI. STRING HANDLING in JAVA

A string is a sequence of character in Java, widely used as an object.

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

1. char[] ch={'j','a','v','a','t','p','o','i','n','t'};
2. String s=new String(ch);

is same as:

1. String s="javatpoint";

Java String class provides a lot of methods to perform operations on string such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

What is String in java

Generally, string is a sequence of characters. But in java, string is an object that represents a sequence of characters. The java.lang.String class is used to create string object.

How to create String object?

There are two ways to create String object:

1. By string literal
2. By new keyword

1) String Literal

Java String literal is created by using double quotes. For Example:

1. String s="welcome";

Each time you create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, a reference to the pooled instance is returned. If string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

1. String s1="Welcome";
2. String s2="Welcome";//will not create new instance

2) By new keyword

1. String s=new String("Welcome");//creates two objects and one reference variable

1. public class StringExample{
2. public static void main(String args[]){
3. String s1="java";//creating string by java string literal
4. char ch[]={'s','t','r','i','n','g','s'};
5. String s2=new String(ch);//converting char array to string
6. String s3=new String("example");//creating java string by new keyword
7. System.out.println(s1);
8. System.out.println(s2);
9. System.out.println(s3);
10.   }}

Java String class methods

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 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 object
11	boolean isEmpty()	checks if string is empty
12	String concat(String str)	concatinates specified string
13	String replace(char old, char new)	replaces all occurrences of specified char value
14	String replace(CharSequence old, CharSequence new)	replaces all occurrences of specified CharSequence
15	static String equalsIgnoreCase(String another)	compares another string. It doesn't check case.
16	String[] split(String regex)	returns splitted string matching regex
17	String[] split(String regex, int limit)	returns splitted string matching regex and limit
18	String intern()	returns interned string

19	int indexOf(int ch)	returns specified char value index
20	int indexOf(int ch, int fromIndex)	returns specified char value index starting with given index
21	int indexOf(String substring)	returns specified substring index
22	int indexOf(String substring, int fromIndex)	returns specified substring index starting with given index
23	String toLowerCase()	returns string in lowercase.
24	String toLowerCase(Locale l)	returns string in lowercase using specified locale.
25	String toUpperCase()	returns string in uppercase.
26	String toUpperCase(Locale l)	returns 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 overloaded.

```
public class Sample {

   public static void main(String args[]) {
      char[] nameArray = {'A', 'l', 'e', 'x'};
      String name = new String(nameArray);
      System.out.println(name);
   } }
```

concat() method can be used to attach strings.

```
public class Sample {

   public static void main(String args[]) {
      String str1 = "Hello ", str2 = "World!";
      System.out.println(str1.concat(str2));
   }
```

}

+ operator is more commonly used to attach strings.

"Hello," + " world" + "!"

Java **toUpperCase()** and **toLowerCase()** method is used to change string case.

```java
public class Sample {

    public static void main(String args[]) {
        String str1 = "Hello";
        System.out.println(str1.toUpperCase());
        System.out.println(str1.toLowerCase());
    }
}
```

Java **trim()** method is used to eliminates white spaces before and after a string.

```java
public class Sample {

    public static void main(String args[]) {
        String str = " Hello ";
        System.out.println(str.trim());
    }
}
```

Java **length()** method is used to get the length of the string.

```java
public class Sample {

    public static void main(String args[]) {
        String str = "Cloud";
        System.out.println(str.length());
    }
}
```

Java String compare

We can compare string in java on the basis of content and reference.

It is used in **authentication** (by equals() method), **sorting** (by compareTo() method), **reference matching** (by == operator) etc.

There are three ways to compare string in java:

1. By equals() method
2. By = = operator
3. By compareTo() method

1) String compare by equals() method

The String equals() method compares the original content of the string. It compares values of string for equality. String class provides two methods:

- **public boolean equals(Object another)** compares this string to the specified object.
- **public boolean equalsIgnoreCase(String another)** compares this String to another string, ignoring case.

```
1. class Teststringcomparison1{
2.  public static void main(String args[]){
3.    String s1="Sachin";
4.    String s2="Sachin";
5.    String s3=new String("Sachin");
6.    String s4="Saurav";
7.    System.out.println(s1.equals(s2));//true
8.    System.out.println(s1.equals(s3));//true
9.    System.out.println(s1.equals(s4));//false
10.       }
11.       }
```

2) String compare by == operator

The = = operator compares references not values.

```
1. class Teststringcomparison3{
2.  public static void main(String args[]){
3.    String s1="Sachin";
4.    String s2="Sachin";
5.    String s3=new String("Sachin");
6.    System.out.println(s1==s2);//true (because both refer to same instance)
7.    System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)
8. }
9. }
```

3) String compare by compareTo() method

The String compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two string variables. If:

- **s1 == s2** :0
- **s1 > s2** :positive value
- **s1 < s2** :negative value

1. class Teststringcomparison4{
2. public static void main(String args[]){
3. String s1="Sachin";
4. String s2="Sachin";
5. String s3="Ratan";
6. System.out.println(s1.compareTo(s2));//0
7. System.out.println(s1.compareTo(s3));//1(because s1>s3)
8. System.out.println(s3.compareTo(s1));//-1(because s3 < s1)
9. }
10.   }

String Concatenation in Java

In java, string concatenation forms a new string *that is* the combination of multiple strings. There are two ways to concat string in java:

1. By + (string concatenation) operator
2. By concat() method

1) String Concatenation by + (string concatenation) operator

Java string concatenation operator (+) is used to add strings. For Example:

1. class TestStringConcatenation1{
2. public static void main(String args[]){
3. String s="Sachin"+" Tendulkar";
4. System.out.println(s);//Sachin Tendulkar
5. }
6. }

2) String Concatenation by concat() method

The String concat() method concatenates the specified string to the end of current string. Syntax:

1. public String concat(String another)

Let's see the example of String concat() method.

1. class TestStringConcatenation3{
2. public static void main(String args[]){
3.   String s1="Sachin ";
4.   String s2="Tendulkar";
5.   String s3=s1.concat(s2);
6.   System.out.println(s3);//Sachin Tendulkar
7. } }

Substring in Java

A part of string is called **substring**. In other words, substring is a subset of another string. In case of substring startIndex is inclusive and endIndex is exclusive.   Note: Index starts from 0.

You can get substring from the given string object by one of the two methods:

1. **public String substring(int startIndex):** This method returns new String object containing the substring of the given string from specified startIndex (inclusive).
2. **public String substring(int startIndex, int endIndex):** This method returns new String object containing the substring of the given string from specified startIndex to endIndex.

In case of string:

- **startIndex:** inclusive   **endIndex:** exclusive

Let's understand the startIndex and endIndex by the code given below.

1. String s="hello";
2. System.out.println(s.substring(0,2));//he

In the above substring, 0 points to h but 2 points to e (because end index is exclusive).

Example of java substring

1. public class TestSubstring{
2. public static void main(String args[]){

```
3.    String s="SachinTendulkar";
4.    System.out.println(s.substring(6));//Tendulkar
5.    System.out.println(s.substring(0,6));//Sachin
6.  }
7.  }
```

StringTokenizer in Java

The **java.util.StringTokenizer** class allows you to break a string into tokens. It is simple way to break string.

It doesn't provide the facility to differentiate numbers, quoted strings, identifiers etc. like StreamTokenizer class. We will discuss about the StreamTokenizer class in I/O chapter.

Constructors of StringTokenizer class

There are 3 constructors defined in the StringTokenizer class

There are 3 constructors defined in the StringTokenizer class.

Constructor	Description
StringTokenizer(String str)	creates StringTokenizer with specified string.
StringTokenizer(String str, String delim)	creates StringTokenizer with specified string and delimeter.
StringTokenizer(String str, String delim, boolean returnValue)	creates StringTokenizer with specified string, delimeter and returnValue. If return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens.

Methods of StringTokenizer class

The 6 useful methods of StringTokenizer class are as follows:

Public method	Description
boolean hasMoreTokens()	checks if there is more tokens available.

String nextToken()	returns the next token from the StringTokenizer object.
String nextToken(String delim)	returns the next token based on the delimeter.
boolean hasMoreElements()	same as hasMoreTokens() method.
Object nextElement()	same as nextToken() but its return type is Object.
int countTokens()	returns the total number of tokens.

1. import java.util.StringTokenizer;
2. public class Simple{
3. public static void main(String args[]){
4.   StringTokenizer st = new StringTokenizer("my name is khan"," ");
5.   while (st.hasMoreTokens()) {
6.   System.out.println(st.nextToken());
7.   }
8.   }
9. }

Example of nextToken(String delim) method of StringTokenizer class

1. import java.util.*;
2.
3. public class Test {
4.   public static void main(String[] args) {
5.   StringTokenizer st = new StringTokenizer("my,name,is,khan");
6.
7.   // printing next token
8.   System.out.println("Next token is : " + st.nextToken(","));
9.   }
10.   }

XII JAVA I/O TUTORIAL

Java I/O (Input and Output) is used *to process the input* and *produce the output.*

Java uses the concept of stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

We can perform **file handling in java** by Java I/O API.

Stream

A stream is a sequence of data.In Java a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

In java, 3 streams are created for us automatically. All these streams are attached with console.

1) System.out: standard output stream

2) System.in: standard input stream

3) System.err: standard error stream

	Byte Based		Character Based	
	Input	Output	Input	Output
Basic	InputStream	OutputStream	Reader InputStreamReader	Writer OutputS
Arrays	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArr
Files	FileInputStream RandomAccessFile	FileOutputStream RandomAccessFile	FileReader	FileWrite
Pipes	PipedInputStream	PipedOutputStream	PipedReader	PipedWr
Buffering	BufferedInputStream	BufferedOutputStream	BufferedReader	Buffered
Filtering	FilterInputStream	FilterOutputStream	FilterReader	FilterWri
Parsing	PushbackInputStream StreamTokenizer		PushbackReader LineNumberReader	
Strings			StringReader	StringWr
Data	DataInputStream	DataOutputStream		
Data – Formatted		PrintStream		PrintWri
Objects	ObjectInputStream	ObjectOutputStream		
Utilities	SequenceInputStream			

Byte Streams Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, FileInputStream and FileOutputStream. Following is an example which makes use of these two classes to copy an input file into an output file:

import java.io.*;

public class CopyFile {

   public static void main(String args[]) throws IOException

   {

```java
        FileInputStream in = null;

        FileOutputStream out = null;

        try {

            in = new FileInputStream("input.txt");

 out = new FileOutputStream("output.txt");

            int c;

            while ((c = in.read()) != -1) {

                out.write(c);

            }

        }finally {

            if (in != null) {

                in.close();

            }

            if (out != null) {

                out.close();

            }

        }   }

}
```

Character Streams Java Byte streams are used to perform input and output of 8-bit bytes, whereas Java Character streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, FileReader and FileWriter. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

```java
import java.io.*;

public class CopyFile {

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

```java
{
  FileReader in = null;

  FileWriter out = null;

  try {

    in = new FileReader("input.txt");

    out = new FileWriter("output.txt");

    int c;

    while ((c = in.read()) != -1) {

      out.write(c);

    }

  }finally {

    if (in != null) {

      in.close();

    }

    if (out != null) {

      out.close();      }    }    }    }


import java.io.*;

public class ReadConsole {

  public static void main(String args[]) throws IOException

  {

    InputStreamReader cin = null;

    try {

      cin = new InputStreamReader(System.in);

      System.out.println("Enter characters, 'q' to quit.");
```

```java
        char c;

        do {

            c = (char) cin.read();

            System.out.print(c);

        } while(c != 'q');

    }finally {

        if (cin != null) {

            cin.close();

        }

    }

}
```

Java BufferedInputStream Class

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

```java
1.  import java.io.*;
2.  public class BufferedInputStreamExample{
3.   public static void main(String args[]){
4.    try{
5.     FileInputStream fin=new FileInputStream("D:\\testout.txt");
6.     BufferedInputStream bin=new BufferedInputStream(fin);
7.     int i;
8.     while((i=bin.read())!=-1){
9.      System.out.print((char)i);
10.          }
11.         bin.close();
12.         fin.close();
13.       }catch(Exception e){System.out.println(e);}
14.      }
15.     }
```

UNIT III

XIII EXCEPTION HANDLING IN JAVA

The **exception handling in java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained. Exception is an abnormal condition. Exception Handling is a mechanism to handle runtime errors

Hierarchy of Java Exception classes
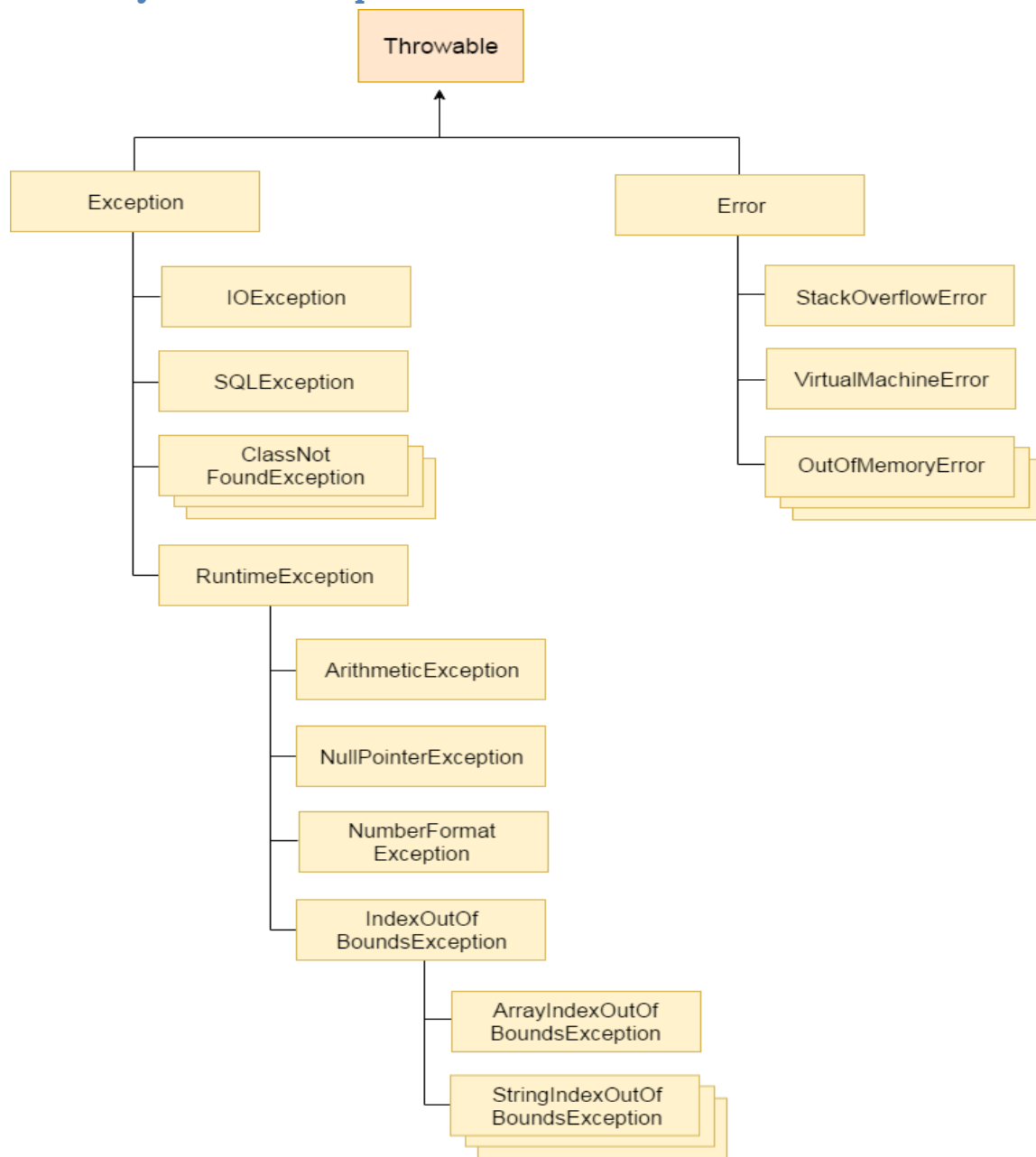
Types of Exception

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

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between checked and unchecked exceptions

1) Checked Exception

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

3) Error

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

Java Exception Handling Keywords

There are 5 keywords used in java exception handling.

1. Try   catch   finally   throw throws

Java try-catch

Java 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

1. try{
2. //code that may throw exception
3. }catch(Exception_class_Name ref){}

1. public class Testtrycatch2{
2.   public static void main(String args[]){
3.   try{
4.   int data=50/0;
5.   }catch(ArithmeticException e){System.out.println(e);}
6.   System.out.println("rest of the code...");
7. }
8. }

Java Multi catch block

If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block.

Let's see a simple example of java multi-catch block.

```
1.  public class TestMultipleCatchBlock{
2.   public static void main(String args[]){
3.    try{
4.     int a[]=new int[5];
5.     a[5]=30/0;
6.    }
7.    catch(ArithmeticException e){System.out.println("task1 is completed
    ");}
8.    catch(ArrayIndexOutOfBoundsException e){System.out.println("task
    2 completed");}
9.    catch(Exception e){System.out.println("common task completed");}
10.
11.         System.out.println("rest of the code...");
12.      }
13.      }
```

Java finally block

Java finally block is a block that is used *to execute important code* such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block.

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**.

```
1.  class TestFinallyBlock{
2.   public static void main(String args[]){
3.    try{
```

```
4.   int data=25/5;
5.    System.out.println(data);
6.   }
7.   catch(NullPointerException e){System.out.println(e);}
8.   finally{System.out.println("finally block is always executed");}
9.   System.out.println("rest of the code...");
10.          }
11.          }
```

Case 3

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

```
1.  public class TestFinallyBlock2{
2.    public static void main(String args[]){
3.    try{
4.     int data=25/0;
5.     System.out.println(data);
6.    }
7.    catch(ArithmeticException e){System.out.println(e);}
8.    finally{System.out.println("finally block is always executed");}
9.    System.out.println("rest of the code...");
10.          }
11.          }
```

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 given below.

```
1.  throw exception;
```

```
1.  public class TestThrow1{
2.    static void validate(int age){
3.      if(age<18)
4.       throw new ArithmeticException("not valid");
5.      else
6.       System.out.println("welcome to vote");
7.    }
8.    public static void main(String args[]){
9.      validate(13);
10.          System.out.println("rest of the code...");
```

```
11.        }
12.    }
```

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.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

Syntax of java throws

```
1. return_type method_name() throws exception_class_name{
2. //method code
3. }
```

```java
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 } }
public class Testthrows2{
   public static void main(String args[]){
    try{
     M m=new M();
     m.method();
    }catch(Exception e){System.out.println("exception handled");}
    System.out.println("normal flow...");
 } }
```

Java Custom Exception

If you are creating your 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.

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

```
1. class InvalidAgeException extends Exception{
2.  InvalidAgeException(String s){
```

```
3.    super(s);
4.  }
5. }

1. class TestCustomException1{
2.
3.    static void validate(int age)throws InvalidAgeException{
4.      if(age<18)
5.       throw new InvalidAgeException("not valid");
6.      else
7.       System.out.println("welcome to vote");
8.    }
9.
10.          public static void main(String args[]){
11.            try{
12.            validate(13);
13.            }catch(Exception m){System.out.println("Exception occured:
    "+m);}
14.
15.            System.out.println("rest of the code...");  }}
```

XIV MULTI THREADING IN JAVA

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.

Advantages of Java Multithreading

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

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

3) Threads are independent so it doesn't affect other threads if 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.

Life cycle of a Thread (Thread States)

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated

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:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated
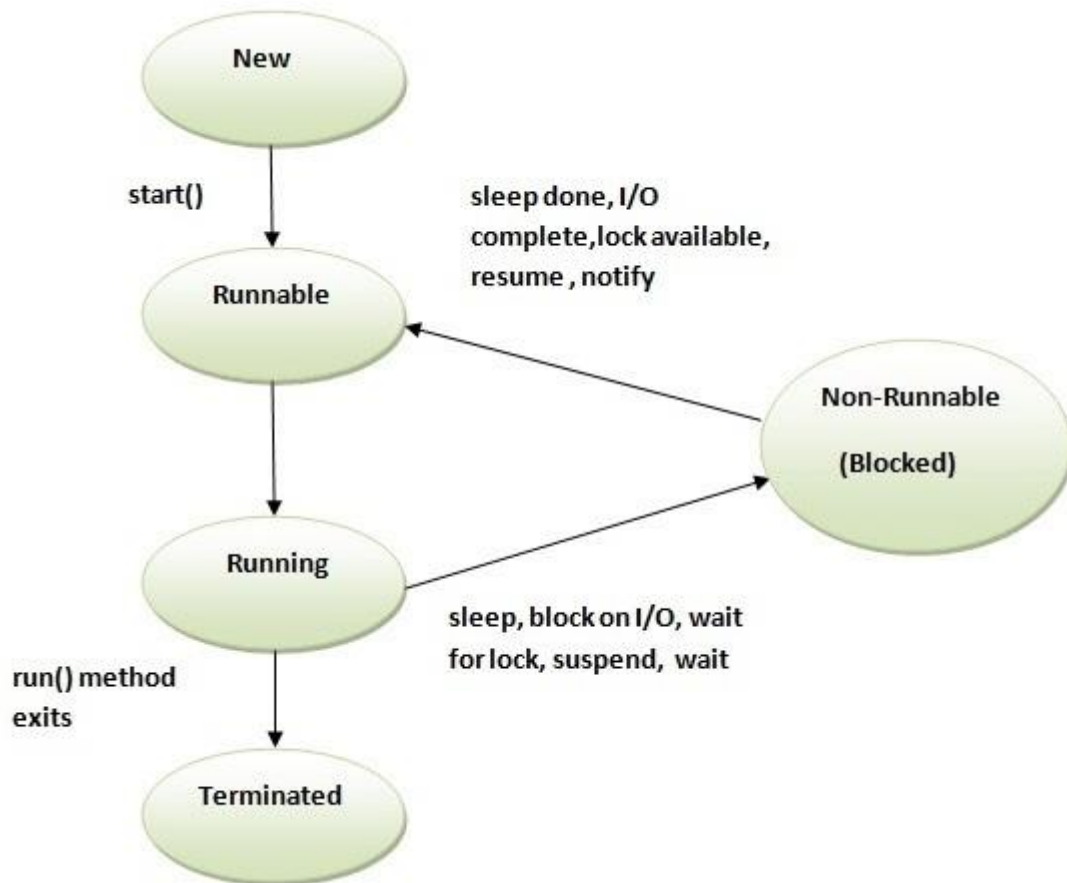
1) New

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

2) Runnable

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

3) Running

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

4) Non-Runnable (Blocked)

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

5) Terminated

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

How to create thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

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:

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

public void start(): starts the execution of the thread.JVM calls the run() method on the thread.

public void sleep(long miliseconds): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

public void join(): waits for a thread to die.

public void join(long miliseconds): waits for a thread to die for the specified miliseconds.

public int getPriority(): returns the priority of the thread.

public int setPriority(int priority): changes the priority of the thread.

public String getName(): returns the name of the thread.

public void setName(String name): changes the name of the thread.

public Thread currentThread(): returns the current thread.

public int getId(): returns the id of the thread.

public Thread.State getState(): returns the state of the thread.

public boolean isAlive(): tests if the thread is alive.

public void yield(): causes the currently executing thread object to temporarily pause and allow other threads to execute.

public void suspend(): is used to suspend the thread(depricated).

public void resume(): is used to resume the suspended thread(depricated).

public void stop(): is used to stop the thread(depricated).

public boolean isDaemon(): tests if the thread is a daemon thread.

public void setDaemon(boolean b): marks the thread as daemon or user thread.

public void interrupt(): interrupts the thread.

public boolean isInterrupted(): tests if the thread has been interrupted.

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

1. class Multi extends Thread{
2. public void run(){
3. System.out.println("thread is running...");

```
4.  }
5.  public static void main(String args[]){
6.  Multi t1=new Multi();
7.  t1.start();
8.   }
9.  }
```

2) Java Thread Example by implementing Runnable interface

```
1.  class Sample  implements Runnable{
2.  public void run(){
3.  System.out.println("Thread is running...");
4.  }
5.
6.  public static void main(String args[]){
7.  Sample  s=new Sample();
8.  Thread t1 =new Thread(s);
9.  t1.start();
10.        }
11.        }
```

Sleep method in java

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

Syntax of sleep() method in java

The Thread class provides two methods for sleeping a thread:

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

Example of sleep method in java

```
1.  class Ex  extends Thread{
2.   public void run(){
3.    for(int j=1;j<5;j++){
4.      try{Thread.sleep(500);}catch(InterruptedException e){System.out.pri
   ntln(e);}
5.      System.out.println(j);
```

6.   }
7.   }
8.   public static void main(String args[]){
9.   Ex t1=new Ex();
10.   Ex t2=new Ex();
11.
12.   t1.start();
13.   t2.start();
14.   }
15.   }

The join() method

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

Syntax:
public void join()throws InterruptedException

public void join(long milliseconds)throws InterruptedException

Example of join() method

1. class Sample extends Thread{
2.   public void run(){
3.   for(int i=1;i<=5;i++){
4.   try{
5.   Thread.sleep(500);
6.   }catch(Exception e){System.out.println(e);}
7.   System.out.println(i);
8.   }   }
9. public static void main(String args[]){
10.   Sample t1=new Sample ();
11.   Sample t2=new Sample ();
12.   Sample t3=new Sample ();
13.   t1.start();
14.   try{
15.   t1.join();
16.   }catch(Exception e){System.out.println(e);}
17.   t2.start();
18.   t3.start();
19.   } }

getName(),setName(String) and getId() method:

public String getName()

public void setName(String name)

public long getId()

```java
1.  class Sample1  extends Thread{
2.   public void run(){
3.    System.out.println("running...");
4.   }
5.   public static void main(String args[]){
6.   Sample1t1=new Sample1 ();
7.    Sample1 t2=new Sample1 ();
8.    System.out.println("Name of t1:"+t1.getName());
9.    System.out.println("Name of t2:"+t2.getName());
10.        System.out.println("id of t1:"+t1.getId());
11.        t1.start();
12.        t2.start();
13.        t1.setName("Sonoo Jaiswal");
14.        System.out.println("After changing name of t1:"+t1.getName());

15.        }}
```

Priority of a Thread (Thread Priority):

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

3 constants defined in Thread class:
1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. 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:

1. class TestMultiPriority1 extends Thread{
2.   public void run(){
3.   System.out.println("running thread name is:"+Thread.currentThread ().getName());
4.   System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
5.
6.   }
7.   public static void main(String args[]){
8.   TestMultiPriority1 m1=new TestMultiPriority1();
9.   TestMultiPriority1 m2=new TestMultiPriority1();
10.   m1.setPriority(Thread.MIN_PRIORITY);
11.   m2.setPriority(Thread.MAX_PRIORITY);
12.   m1.start();
13.   m2.start();
14.
15.   }
16.   }

ThreadGroup in Java

Java provides a convenient way to group multiple threads in a single object. In such way, we can suspend, resume or interrupt group of threads by a single method call.

Constructors of ThreadGroup class

There are only two constructors of ThreadGroup class.

No.	Constructor	Description
1)	ThreadGroup(String name)	creates a thread group with given name.
2)	ThreadGroup(ThreadGroup parent, String name)	creates a thread group with given parent group and name.

ThreadGroup Example

File: ThreadGroupDemo.java

```
1.  public class ThreadGroupDemo implements Runnable{
2.     public void run()
3.     {
4.         System.out.println(Thread.currentThread().getName());
5.     }
6.    public static void main(String[] args)
7.     {
8.       ThreadGroupDemo runnable = new ThreadGroupDemo();
9.         ThreadGroup tg1 = new ThreadGroup("Parent ThreadGroup");
10.
11.            Thread t1 = new Thread(tg1, runnable,"one");
12.            t1.start();
13.            Thread t2 = new Thread(tg1, runnable,"two");
14.            t2.start();
15.            Thread t3 = new Thread(tg1, runnable,"three");
16.            t3.start();
17.
18.            System.out.println("Thread Group Name: "+tg1.getName()
    );
19.            tg1.list();
20.
21.        }
22.        }
```

```
1.  class TestMultitasking1 extends Thread{
2.  public void run(){
3.    System.out.println("task one");
4.  }
5.  public static void main(String args[]){
6.  TestMultitasking1 t1=new TestMultitasking1();
7.  TestMultitasking1 t2=new TestMultitasking1();
8.  TestMultitasking1 t3=new TestMultitasking1();
9.
10.        t1.start();
11.        t2.start();
12.        t3.start();
13.      }
14.      }
```

Synchronization in Java

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

The synchronization is mainly used to

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

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

1. by synchronized method
2. by synchronized block
3. by static synchronization

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has an lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package java.util.concurrent.locks contains several lock implementations.

```
1.  //example of java synchronized method
2.  class Table{
3.   synchronized void printTable(int n){//synchronized method
4.     for(int i=1;i<=5;i++){
5.       System.out.println(n*i);
6.       try{
7.        Thread.sleep(400);
8.       }catch(Exception e){System.out.println(e);}
9.     }
10.
11.       }
12.       }
13.
14.      class MyThread1 extends Thread{
15.      Table t;
16.      MyThread1(Table t){
17.      this.t=t;
18.      }
19.      public void run(){
20.      t.printTable(5);
21.      }
22.
23.      }
24.      class MyThread2 extends Thread{
25.      Table t;
26.      MyThread2(Table t){
27.      this.t=t;
28.      }
29.      public void run(){
30.      t.printTable(100);
31.      }
32.      }
33.
34.      public class TestSynchronization2{
35.      public static void main(String args[]){
36.      Table obj = new Table();//only one object
37.      MyThread1 t1=new MyThread1(obj);
38.      MyThread2 t2=new MyThread2(obj);
39.      t1.start();
40.      t2.start();
41.      }
42.      }
```

Inter-thread communication in Java

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

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

- wait()
- notify()
- notifyAll()

1) wait() method

Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
public final void wait()throws InterruptedException	waits until object is notified.
public final void wait(long timeout)throws InterruptedException	waits for the specified amount of time.

2) notify() method

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. Syntax:

public final void notify()

3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor. Syntax:

public final void notifyAll()

UNIT IV

Java Applet

Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

Advantage of Applet

There are many advantages of applet. They are as follows:

- It works at client side so less response time.
- Secured
- It can be executed by browsers running under many plateforms, including Linux, Windows, Mac Os etc.
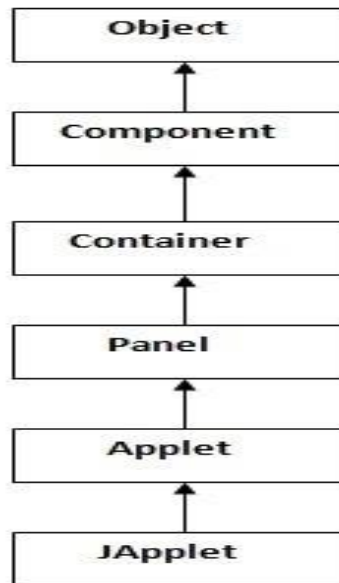
Drawback of Applet

- Plugin is required at client browser to execute applet.

Do You Know

- Who is responsible to manage the life cycle of an applet ?
- How to perform animation in applet ?
- How to paint like paint brush in applet ?
- How to display digital clock in applet ?
- How to display analog clock in applet ?
- How to communicate two applets ?

Hierarchy of Applet

Displaying Graphics in Applet

java.awt.Graphics class provides many methods for graphics programming.

Commonly used methods of Graphics class:

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:

1. import java.applet.Applet;
2. import java.awt.*;
3.
4. public class GraphicsDemo extends Applet{
5.
6. public void paint(Graphics g){
7. g.setColor(Color.red);
8. g.drawString("Welcome",50, 50);
9. g.drawLine(20,30,20,300);
10.   g.drawRect(70,100,30,30);
11.   g.fillRect(170,100,30,30);
12.   g.drawOval(70,200,30,30);
13.
14.   g.setColor(Color.pink);
15.   g.fillOval(170,200,30,30);
16.   g.drawArc(90,150,30,30,30,270);
17.   g.fillArc(270,150,30,30,0,180);
18.
19.   }
20.   }

EventHandling in Applet

As we perform event handling in AWT or Swing, we can perform it in applet also. Let's see the simple example of event handling in applet that prints a message by click on the button.

Example of EventHandling in applet:

1. import java.applet.*;
2. import java.awt.*;
3. import java.awt.event.*;
4. public class EventApplet extends Applet implements ActionListener{
5. Button b;
6. TextField tf;
7.

```
8.  public void init(){
9.  tf=new TextField();
10.        tf.setBounds(30,40,150,20);
11.
12.        b=new Button("Click");
13.        b.setBounds(80,150,60,50);
14.
15.        add(b);add(tf);
16.        b.addActionListener(this);
17.
18.        setLayout(null);
19.        }
20.
21.         public void actionPerformed(ActionEvent e){
22.          tf.setText("Welcome");
23.         }
24.         }
```

In the above example, we have created all the controls in init() method because it is invoked only once.

myapplet.html

```
1.  <html>
2.  <body>
3.  <applet code="EventApplet.class" width="300" height="300">
4.  </applet>
5.  </body>
6.  </html>
```

Java AWT Tutorial

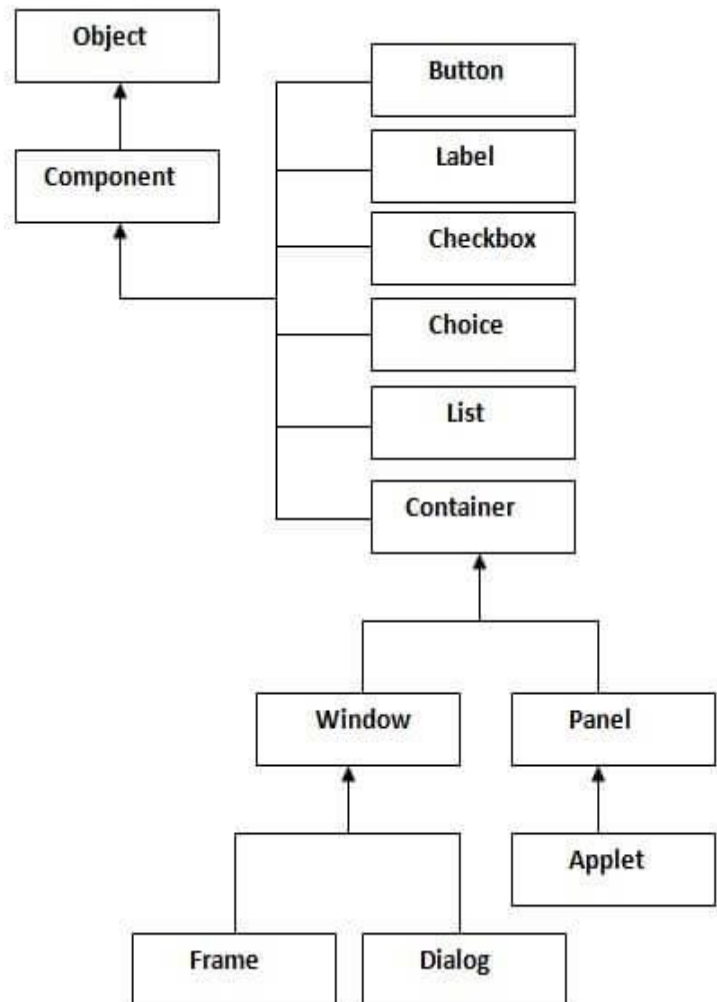**Java AWT** (Abstract Window Toolkit) is *an API to develop GUI or window-based applications* in java.

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.

```
                        ┌──────────┐
                        │  Object  │        ┌──────────┐
                        └──────────┘        │  Button  │
                             ▲              └──────────┘
                             │              ┌──────────┐
                       ┌───────────┐        │  Label   │
                       │ Component │        └──────────┘
                       └───────────┘        ┌──────────┐
                             ▲              │ Checkbox │
                             │              └──────────┘
                                            ┌──────────┐
                                            │  Choice  │
                                            └──────────┘
                                            ┌──────────┐
                                            │   List   │
                                            └──────────┘
                                            ┌──────────┐
                                            │Container │
                                            └──────────┘
                                                 ▲
                          ┌──────────────────────┴────────────┐
                    ┌──────────┐                        ┌──────────┐
                    │  Window  │                        │  Panel   │
                    └──────────┘                        └──────────┘
                         ▲                                   ▲
              ┌──────────┴──────────┐                  ┌──────────┐
         ┌──────────┐        ┌──────────┐              │  Applet  │
         │  Frame   │        │  Dialog  │              └──────────┘
         └──────────┘        └──────────┘
```

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

1. public class Button extends Component implements Accessible

Java AWT Button Example

1. import java.awt.*;
2. public class ButtonExample {
3. public static void main(String[] args) {
4.   Frame f=new Frame("Button Example");
5.   Button b=new Button("Click Here");
6.   b.setBounds(50,100,80,30);
7.   f.add(b);
8.   f.setSize(400,400);
9.   f.setLayout(null);
10.   f.setVisible(true);
11.   }
12.   }

Output:
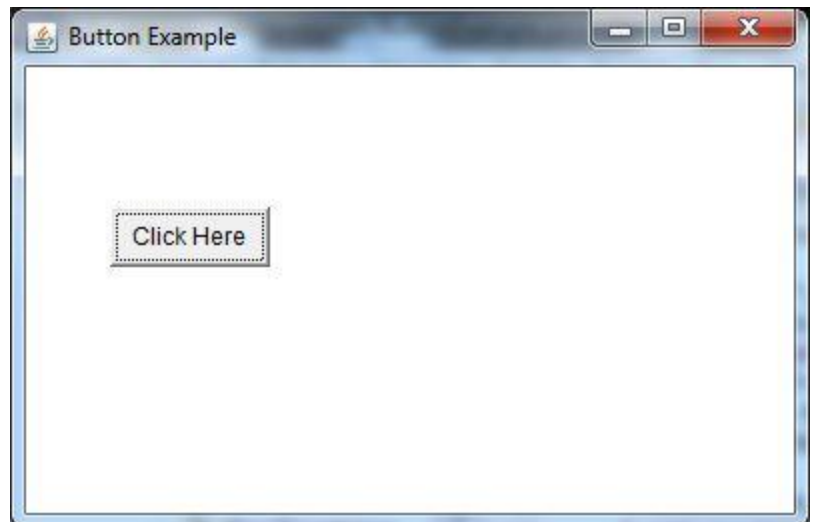
Java ActionListener Interface

The Java ActionListener is notified whenever you click on the button or menu item. It is notified against ActionEvent. The ActionListener interface is found in java.awt.event package. It has only one method: actionPerformed().

actionPerformed() method

The actionPerformed() method is invoked automatically whenever you click on the registered component.

1. public abstract void actionPerformed(ActionEvent e);

Java ActionListener Example: On Button click

1. import java.awt.*;
2. import java.awt.event.*;
3. public class ActionListenerExample {
4. public static void main(String[] args) {
5.   Frame f=new Frame("ActionListener Example");
6.   final TextField tf=new TextField();
7.   tf.setBounds(50,50, 150,20);
8.   Button b=new Button("Click Here");
9.   b.setBounds(50,100,60,30);
10.
11.   b.addActionListener(new ActionListener(){
12.   public void actionPerformed(ActionEvent e){
13.   tf.setText("Welcome to Javatpoint.");
14.   }
15.   });
16.   f.add(b);f.add(tf);
17.   f.setSize(400,400);
18.   f.setLayout(null);
19.   f.setVisible(true);
20.   }
21.   }

Output:
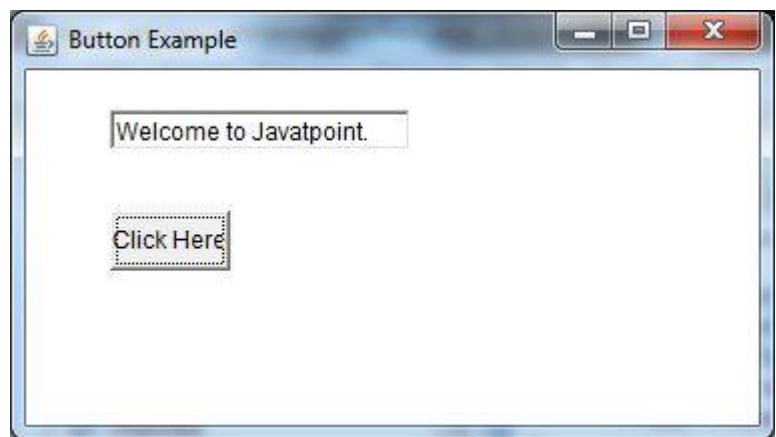
UNIT V

Java LayoutManagers

The LayoutManagers are used to arrange components in a particular manner. LayoutManager is an interface that is implemented by all the classes of layout managers. There are following classes that represents the layout managers:

1. java.awt.BorderLayout
2. java.awt.FlowLayout
3. java.awt.GridLayout
4. java.awt.CardLayout
5. java.awt.GridBagLayout
6. javax.swing.BoxLayout
7. javax.swing.GroupLayout
8. javax.swing.ScrollPaneLayout
9. javax.swing.SpringLayout etc.

Java BorderLayout

The BorderLayout is used to arrange the components in five regions: north, south, east, west and center. Each region (area) may contain one component only. It is the default layout of frame or window. The BorderLayout provides five constants for each region:

1. **public static final int NORTH**
2. **public static final int SOUTH**
3. **public static final int EAST**
4. **public static final int WEST**
5. **public static final int CENTER**

Constructors of BorderLayout class:

- **BorderLayout():** creates a border layout but with no gaps between the components.
- **JBorderLayout(int hgap, int vgap):** creates a border layout with the given horizontal and vertical gaps between the components.

Java GridBagLayout

The Java GridBagLayout class is used to align components vertically, horizontally or along their baseline.

The components may not be of same size. Each GridBagLayout object maintains a dynamic, rectangular grid of cells. Each component occupies one or more cells known as its display area. Each component associates an instance of GridBagConstraints. With the help of constraints object we arrange component's display area on the grid. The GridBagLayout manages each component's minimum and preferred sizes in order to determine component's size.

Fields

Modifier and Type	Field	Description
double[]	columnWeights	It is used to hold the overrides to the column weights.

int[]	columnWidths	It is used to hold the overrides to the column minimum width.
protected Hashtable<Component,GridBagConstraints >	comptable	It is used to maintains the association between a component and its gridbag constraints.
protected GridBagConstraints	defaultConstraints	It is used to hold a gridbag constraints instance containing the default values.
protected GridBagLayoutInfo	layoutInfo	It is used to hold the layout information for the gridbag.
protected static int	MAXGRIDSIZE	No longer in use just for backward compatibility
protected static int	MINSIZE	It is smallest grid that can be laid out by the grid

		bag layout.
protected static int	PREFERREDSIZE	It is preferred grid size that can be laid out by the grid bag layout.
int[]	rowHeights	It is used to hold the overrides to the row minimum heights.
double[]	rowWeights	It is used to hold the overrides to the row weights.

Useful Methods

Modifier and Type	Method	Description
void	addLayoutComponent(Component comp, Object constraints)	It adds specified component to the layout, using the specified constraints object.
void	addLayoutComponent(String name, Component comp)	It has no effect, since this layout manager does not use a per-component string.
protected void	adjustForGravity(GridBagConstraints constraints, Rectangle r)	It adjusts the x, y, width, and height fields to

		the correct values depending on the constraint geometry and pads.
protected void	AdjustForGravity(GridBagConstraints constraints, Rectangle r)	This method is for backwards compatibility only
protected void	arrangeGrid(Container parent)	Lays out the grid.
protected void	ArrangeGrid(Container parent)	This method is obsolete and supplied for backwards compatibility
GridBagConstraints	getConstraints(Component comp)	It is for getting the constraints for the specified component.
float	getLayoutAlignmentX(Container parent)	It returns the alignment along the x axis.
float	getLayoutAlignmentY(Container parent)	It returns the alignment along the y axis.
int[][]	getLayoutDimensions()	It determines column widths and row heights for the layout grid.
protected GridBagLayoutInfo	getLayoutInfo(Container parent, int sizeflag)	This method is obsolete and supplied for

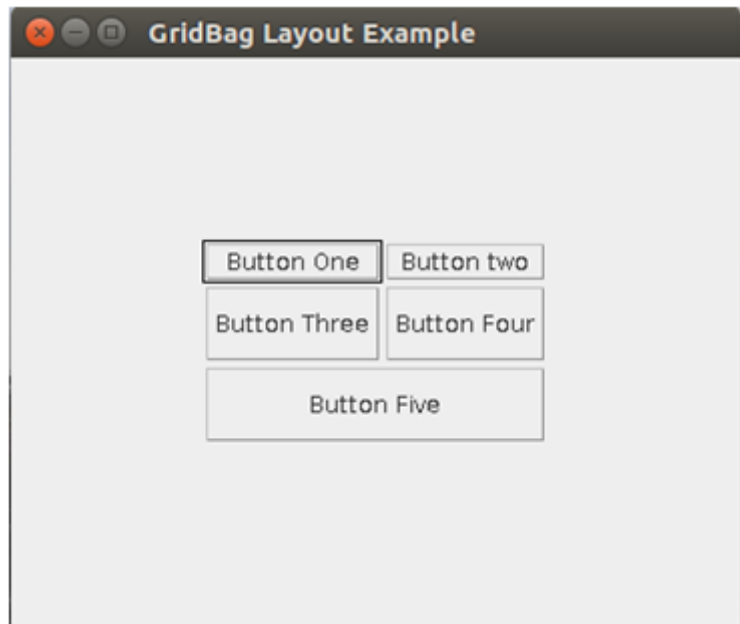|		
protected GridBagLayoutInfo	GetLayoutInfo(Container parent, int sizeflag)	This method is obsolete and supplied for backwards compatibility.
Point	getLayoutOrigin()	It determines the origin of the layout area, in the graphics coordinate space of the target container.
double[][]	getLayoutWeights()	It determines the weights of the layout grid's columns and rows.
protected Dimension	getMinSize(Container parent, GridBagLayoutInfo info)	It figures out the minimum size of the master based on the information from getLayoutInfo.
protected Dimension	GetMinSize(Container parent, GridBagLayoutInfo info)	This method is obsolete and supplied for backwards compatibility only

Example

1. import java.awt.Button;
2. import java.awt.GridBagConstraints;
3. import java.awt.GridBagLayout;
4.
5. import javax.swing.*;

```java
6.  public class GridBagLayoutExample extends JFrame{
7.     public static void main(String[] args) {
8.         GridBagLayoutExample a = new GridBagLayoutExample();
9.      }
10.          public GridBagLayoutExample() {
11.        GridBagLayoutgrid = new GridBagLayout();
12.            GridBagConstraints gbc = new GridBagConstraints();
13.            setLayout(grid);
14.            setTitle("GridBag Layout Example");
15.            GridBagLayout layout = new GridBagLayout();
16.        this.setLayout(layout);
17.        gbc.fill = GridBagConstraints.HORIZONTAL;
18.        gbc.gridx = 0;
19.        gbc.gridy = 0;
20.        this.add(new Button("Button One"), gbc);
21.        gbc.gridx = 1;
22.        gbc.gridy = 0;
23.        this.add(new Button("Button two"), gbc);
24.        gbc.fill = GridBagConstraints.HORIZONTAL;
25.        gbc.ipady = 20;
26.        gbc.gridx = 0;
27.        gbc.gridy = 1;
28.        this.add(new Button("Button Three"), gbc);
29.        gbc.gridx = 1;
30.        gbc.gridy = 1;
31.        this.add(new Button("Button Four"), gbc);
32.        gbc.gridx = 0;
33.        gbc.gridy = 2;
34.        gbc.fill = GridBagConstraints.HORIZONTAL;
35.        gbc.gridwidth = 2;
36.        this.add(new Button("Button Five"), gbc);
37.            setSize(300, 300);
38.            setPreferredSize(getSize());
39.            setVisible(true);
40.            setDefaultCloseOperation(EXIT_ON_CLOSE);
41.
42.          }
43.
44.      }
```

Output:

Java Swing Tutorial

Java Swing tutorial is a part of Java Foundation Classes (JFC) that is *used to create window-based applications*. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.

Unlike AWT, Java Swing provides platform-independent and lightweight components.

The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

Difference between AWT and Swing

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

No.	Java AWT	Java Swing
1)	AWT components are **platform-**	Java swing components are

	dependent.	**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.	

Java JButton

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

JButton class declaration

Let's see the declaration for javax.swing.JButton class.

1. public class JButton extends AbstractButton implements Accessible

Commonly used Constructors:

Constructor	Description
JButton()	It creates a button with no text and icon.
JButton(String s)	It creates a button with the specified text.
JButton(Icon i)	It creates a button with the specified icon object.

Commonly used Methods of AbstractButton class:

Methods	Description
void setText(String s)	It is used to set specified text on button
String getText()	It is used to return the text of the button.
void setEnabled(boolean b)	It is used to enable or disable the button.
void setIcon(Icon b)	It is used to set the specified Icon on the button.
Icon getIcon()	It is used to get the Icon of the button.
void setMnemonic(int a)	It is used to set the mnemonic on the button.
void addActionListener(ActionListener a)	It is used to add the action listener to this object.

Java JRadioButton

The JRadioButton class is used to create a radio button. It is used to choose one option from multiple options. It is widely used in exam systems or quiz.

It should be added in ButtonGroup to select one radio button only.

JRadioButton class declaration

Let's see the declaration for javax.swing.JRadioButton class.

1. public class JRadioButton extends JToggleButton implements Accessible

Commonly used Constructors:

Constructor	Description
JRadioButton()	Creates an unselected radio button with no text.

JRadioButton(String s)	Creates an unselected radio button with specified text.
JRadioButton(String s, boolean selected)	Creates a radio button with the specified text and selected status.

Commonly used Methods:

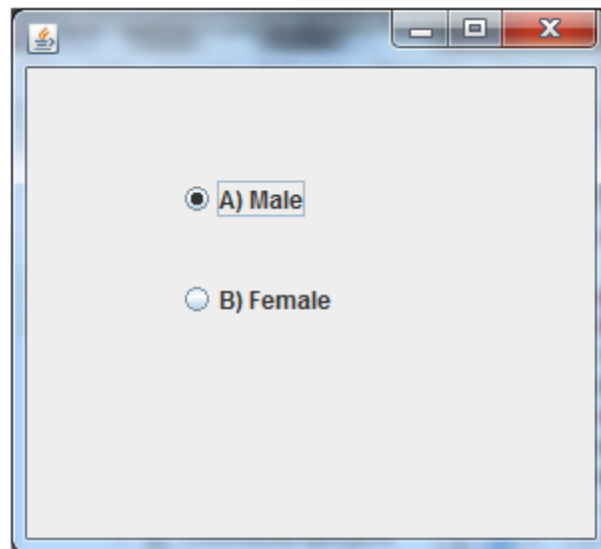| Methods	Description
void setText(String s)	It is used to set specified text on button.
String getText()	It is used to return the text of the button.
void setEnabled(boolean b)	It is used to enable or disable the button.
void setIcon(Icon b)	It is used to set the specified Icon on the button.
Icon getIcon()	It is used to get the Icon of the button.
void setMnemonic(int a)	It is used to set the mnemonic on the button.
void addActionListener(ActionListener a)	It is used to add the action listener to this object.

Java JRadioButton Example

1. import javax.swing.*;
2. public class RadioButtonExample {
3. JFrame f;
4. RadioButtonExample(){
5. f=new JFrame();
6. JRadioButton r1=new JRadioButton("A) Male");
7. JRadioButton r2=new JRadioButton("B) Female");
8. r1.setBounds(75,50,100,30);
9. r2.setBounds(75,100,100,30);
10.   ButtonGroup bg=new ButtonGroup();
11.   bg.add(r1);bg.add(r2);
12.   f.add(r1);f.add(r2);

```
13.        f.setSize(300,300);
14.        f.setLayout(null);
15.        f.setVisible(true);
16.        }
17.        public static void main(String[] args) {
18.            new RadioButtonExample();
19.        }
20.        }
```


Java GridBagLayout

The Java GridBagLayout class is used to align components vertically, horizontally or along their baseline.

The components may not be of same size. Each GridBagLayout object maintains a dynamic, rectangular grid of cells. Each component occupies one or more cells known as its display area. Each component associates an instance of GridBagConstraints. With the help of constraints object we arrange component's display area on the grid. The GridBagLayout manages each component's minimum and preferred sizes in order to determine component's size.

Fields

Modifier and Type	Field	Description
double[]	columnWeights	It is used to hold the

		overrides to the column weights.
int[]	columnWidths	It is used to hold the overrides to the column minimum width.
protected Hashtable<Component,GridBagConstraints >	comptable	It is used to maintains the association between a component and its gridbag constraints.
protected GridBagConstraints	defaultConstraints	It is used to hold a gridbag constraints instance containing the default values.
protected GridBagLayoutInfo	layoutInfo	It is used to hold the layout information for the gridbag.
protected static int	MAXGRIDSIZE	No longer in use just for backward compatibility

protected static int		MINSIZE	It is smallest grid that can be laid out by the grid bag layout.
protected static int		PREFERREDSIZE	It is preferred grid size that can be laid out by the grid bag layout.
int[]		rowHeights	It is used to hold the overrides to the row minimum heights.
double[]		rowWeights	It is used to hold the overrides to the row weights.

Useful Methods

Modifier and Type	Method	Description
void	addLayoutComponent(Component comp, Object constraints)	It adds specified component to the layout, using the specified constraints object.
void	addLayoutComponent(String name, Component comp)	It has no effect, since this layout manager does not use a per-component string.

protected void	adjustForGravity(GridBagConstraints constraints, Rectangle r)	It adjusts the x, y, width, and height fields to the correct values depending on the constraint geometry and pads.
protected void	AdjustForGravity(GridBagConstraints constraints, Rectangle r)	This method is for backwards compatibility only
protected void	arrangeGrid(Container parent)	Lays out the grid.
protected void	ArrangeGrid(Container parent)	This method is obsolete and supplied for backwards compatibility
GridBagConstraints	getConstraints(Component comp)	It is for getting the constraints for the specified component.
float	getLayoutAlignmentX(Container parent)	It returns the alignment along the x axis.
float	getLayoutAlignmentY(Container parent)	It returns the alignment along the y axis.
int[][]	getLayoutDimensions()	It determines column widths and row heights for the layout grid.

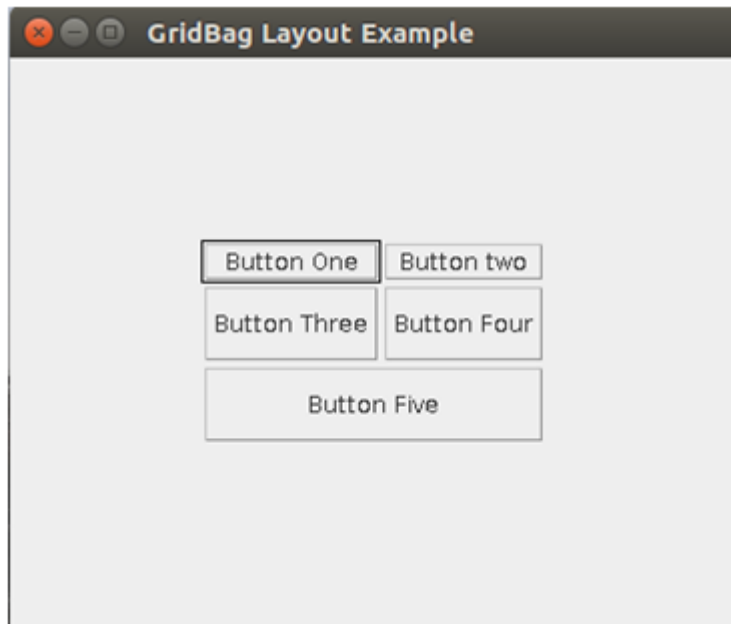|		
protected GridBagLayoutInfo	getLayoutInfo(Container parent, int sizeflag)	This method is obsolete and supplied for backwards compatibility.
protected GridBagLayoutInfo	GetLayoutInfo(Container parent, int sizeflag)	This method is obsolete and supplied for backwards compatibility.
Point	getLayoutOrigin()	It determines the origin of the layout area, in the graphics coordinate space of the target container.
double[][]	getLayoutWeights()	It determines the weights of the layout grid's columns and rows.
protected Dimension	getMinSize(Container parent, GridBagLayoutInfo info)	It figures out the minimum size of the master based on the information from getLayoutInfo.
protected Dimension	GetMinSize(Container parent, GridBagLayoutInfo info)	This method is obsolete and supplied for backwards compatibility only

Example

1. import java.awt.Button;
2. import java.awt.GridBagConstraints;

```java
3. import java.awt.GridBagLayout;
4.
5. import javax.swing.*;
6. public class GridBagLayoutExample extends JFrame{
7.     public static void main(String[] args) {
8.         GridBagLayoutExample a = new GridBagLayoutExample();
9.     }
10.         public GridBagLayoutExample() {
11.         GridBagLayoutgrid = new GridBagLayout();
12.             GridBagConstraints gbc = new GridBagConstraints();
13.             setLayout(grid);
14.             setTitle("GridBag Layout Example");
15.             GridBagLayout layout = new GridBagLayout();
16.         this.setLayout(layout);
17.         gbc.fill = GridBagConstraints.HORIZONTAL;
18.         gbc.gridx = 0;
19.         gbc.gridy = 0;
20.         this.add(new Button("Button One"), gbc);
21.         gbc.gridx = 1;
22.         gbc.gridy = 0;
23.         this.add(new Button("Button two"), gbc);
24.         gbc.fill = GridBagConstraints.HORIZONTAL;
25.         gbc.ipady = 20;
26.         gbc.gridx = 0;
27.         gbc.gridy = 1;
28.         this.add(new Button("Button Three"), gbc);
29.         gbc.gridx = 1;
30.         gbc.gridy = 1;
31.         this.add(new Button("Button Four"), gbc);
32.         gbc.gridx = 0;
33.         gbc.gridy = 2;
34.         gbc.fill = GridBagConstraints.HORIZONTAL;
35.         gbc.gridwidth = 2;
36.         this.add(new Button("Button Five"), gbc);
37.             setSize(300, 300);
38.             setPreferredSize(getSize());
39.             setVisible(true);
40.             setDefaultCloseOperation(EXIT_ON_CLOSE);
41.
42.         }
43.
44.     }
```

Output:

Java JTree

The JTree class is used to display the tree structured data or hierarchical data. JTree is a complex component. It has a 'root node' at the top most which is a parent for all nodes in the tree. It inherits JComponent class.

JTree class declaration

Let's see the declaration for javax.swing.JTree class.

1. public class JTree extends JComponent implements Scrollable, Accessible

Commonly used Constructors:

Constructor	Description
JTree()	Creates a JTree with a sample model.
JTree(Object[] value)	Creates a JTree with every element of the specified array as the child of a new root node.
JTree(TreeNode root)	Creates a JTree with the specified TreeNode as its root, which displays the root node.

Java JTree Example

1. import javax.swing.*;
2. import javax.swing.tree.DefaultMutableTreeNode;
3. public class TreeExample {
4. JFrame f;
5. TreeExample(){
6.   f=new JFrame();
7.   DefaultMutableTreeNode style=new DefaultMutableTreeNode("Style");
8.   DefaultMutableTreeNode color=new DefaultMutableTreeNode("color");
9.   DefaultMutableTreeNode font=new DefaultMutableTreeNode("font");

10.   style.add(color);
11.   style.add(font);
12.   DefaultMutableTreeNode red=new DefaultMutableTreeNode("red");
13.   DefaultMutableTreeNode blue=new DefaultMutableTreeNode("blue");
14.   DefaultMutableTreeNode black=new DefaultMutableTreeNode("black");
15.   DefaultMutableTreeNode green=new DefaultMutableTreeNode("green");
16.   color.add(red); color.add(blue); color.add(black); color.add(green);
17.   JTree jt=new JTree(style);
18.   f.add(jt);
19.   f.setSize(200,200);
20.   f.setVisible(true);
21.   }
22.   public static void main(String[] args) {
23.   new TreeExample();
24.   }}

Output: