MCA 102 - OBJECT ORIENTED PROGRAMMING WITH JAVA

UNIT – I

Object Oriented Programming Fundamentals & Java: Java Features, Object Oriented Programming Concepts –Abstraction, Encapsulation, Inheritance and Polymorphism. Java Fundamentals: Data Types, variables, arrays, Inheritance to classes: class fundamentals, Objects, References, Constructors, Overloading of methods, Access control, Nested and Inner classes. Inheritance: Inheritance basics, Using Super, multilevel hierarchy, method overriding, dynamic method dispatch, abstract classes, final with inheritance.

UNIT-II

Packages, Exceptions and Threads: Packages and Interfaces: Packages, Access protection, Importing packages, interfaces, Exception Handling: fundamentals, exception types, uncaught exceptions, using try, nested try statements, throw, throws, Java built-in exceptions, user defined exceptions. Multithreading: Thread model, main thread, creating a thread, multiple threads, thread priorities, synchronization, Inter thread communication, String handling.

UNIT-III

Java Utilities: Type wrappers: Number, Double, Float, Byte, Short, Integer and Long, Character, Boolean, Math class. Collections: Collection interfaces, collection classes, legacy classes and interfaces: Enumeration interface, Vector, Stack, Dictionary, Hash table. More utility classes: String Tokenizer, Bit set, Date, And Calendar Input/output: File, Stream classes, Byte Streams, Character Streams.

UNIT-IV

GUI Programming Features Applets: Applet basics, Applet architecture, an applet skeleton, Applet display method, Repainting, Using Status window, HTML APPLET tag, passing parameters to applet, Audio Clip interface. Even Handling; two event handling mechanisms, Event model, Event classes, sources of events, Event Listener interfaces, Adapter classes. Introduction to SWING: Window Fundamentals, working with frame windows, creating window programs, working with color, fonts, SWING Controls, Layout Managers and Menus: Control fundamentals, Labels, Using buttons, check boxes, checkbox group, choice controls, lists, scroll bars, text field, layout managers, menu bars, and menus.

UNIT - V

Networking in Java Network Programming with Java, Networking classes and Interfaces, Inet Address, Factory method, Instance Methods, Sockets, Knowing IP address, URL-URL Connection class. Creating a server that sends data, creating a client that receives data, two way communication between server and client, Stages in a JDBC program, registering the driver, connecting to a database, Preparing SQL statements, improving the performance of a JDBC program.

Text Book

1. Herbert Schildt : “The Complete Reference Java 2”(Fifth Edition),TMH.

Reference Books

Dietel & Dietel : “Java2 How to Program”, Prentice Hall.

Thamus Wu: “An Introduction to Object Oriented Programming With Java.” TMH

Balagurusamy:” Programming With Java”: TMH.

Lecture Notes

UNIT - I

➢Object-Oriented Programming Concepts

What is an Object?

An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life. This lesson explains how state and behavior are represented within an object, introduces the concept of data encapsulation, and explains the benefits of designing your software in this manner.

IN Briefly, Objects are key to understanding *object-oriented* technology. Look around right now and you'll find many examples of real-world objects:
your dog, your desk, your television set, your bicycle.

Real-world objects share two characteristics: They all have *state* and *behavior*. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes)

Bundling code into individual software objects provides a number of benefits, including:

1. Modularity: The source code for an object can be written and maintained independently of the source code for other

objects. Once created, an object can be easily passed around inside the system.
2. Information-hiding: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
3. Code re-use: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.
4. Pluggability and debugging ease: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement.

WhatIsaClass?

A class is a blueprint or prototype from which objects are created. This section defines a class that models the state and behavior of a real-world object.

```
class Bicycle { int cadence = 0; int speed = 0; int gear = 1; void
   changeCadence(int newValue) {cadence = newValue} void
   changeGear(int newValue) { gear = newValue} void speedUp(int
   increment) { speed = speed + increment; } void
   applyBrakes(int decrement) {speed = speed - decrement;
          }
   void printStates(){
           System.out.println("cadence:" + cadence + " speed:"
           + speed + " gear:" + gear);        } }
```

WhatIsInheritance?

Inheritance provides a powerful and natural mechanism for organizing and structuring your software. This section explains how classes inherit state and behavior from their superclasses, and explains how to derive one class from another using the simple syntax provided by the Java programming language.

Different kinds of objects often have a certain amount in common with each other. Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics of bicycles (current speed, current pedal cadence, current gear).

Object-oriented programming allows classes to *inherit* commonly used state and behavior from other classes. In this example, Bicycle now becomes the *superclass* of MountainBike, RoadBike, and TandemBike. In the Java programming language, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of *subclasses*:

The syntax for creating a subclass is simple. At the beginning of your class declaration, use the extends keyword, followed by the name of the class to inherit from:

```
class MountainBike extends Bicycle {
   // new fields and methods defining
    // a mountain bike would go here
}
```

This gives MountainBike all the same fields and methods as Bicycle, yet allows its code to focus exclusively on the features that make it unique. This makes code for your subclasses easy to read. However, you must take care to properly document the state and behavior that each superclass defines, since that code will not appear in the source file of each subclass.

WhatIsanInterface?

An interface is a contract between a class and the outside world. When a class implements an interface, it promises to provide the behavior published by that interface. This section defines a simple interface and explains the necessary changes for any class that implements it.

In its most common form, an interface is a group of related methods with empty bodies. A bicycle's behavior, if specified as an interface, might appear as follows:

```
interface Bicycle {
   //   wheel   revolutions   per
     minute                    void
     changeCadence(int
     newValue);                void
     changeGear(int  newValue);
     void               speedUp(int
     increment);
      void applyBrakes(int decrement); }
```

To implement this interface, the name of your class would change (to a particular brand of bicycle, for example, such as ACMEBicycle), and you'd use the implements keyword in the class declaration:

```
class ACMEBicycle **implements** Bicycle {

    int cadence = 0; int speed = 0;
    int gear = 1; void
    changeCadence(int newValue) {
         cadence = newValue}
         void changeGear(int
         newValue) { gear =
         newValue;} void
         speedUp(int increment) {
         speed = speed +
         increment;    } void
         applyBrakes(int
         decrement) { speed =
         speed - decrement;}
         void printStates()
             {System.out.println("cadence:" +
             cadence + " speed:" + speed + " gear:"
             + gear);}}
```

Implementing an interface allows a class to become more formal about the behavior it promises to provide. Interfaces form a contract between the class and the outside world, and this contract is enforced at build time by the compiler.

WhatIsaPackage?

A package is a namespace for organizing classes and interfaces in a logical manner. Placing your code into packages makes large software projects easier to manage. This section explains why this is useful, and introduces you to the Application Programming Interface (API) provided by the Java platform.

The Java Platform API Specificationcontains the complete listing for all packages, interfaces, classes, fields, and methods supplied by the Java SE platform. Load the page in your browser and bookmark it. As a programmer, it will become your single most important piece of reference documentation.

➢Features of Java

The prime reason behind creation of Java was to bring portability and security feature into a computer language.Those features are :

1)Simple

Java is easy to learn and its syntax is quite simple, clean and easy to understand.The confusing and ambiguous concepts of C++ are either left out in Java or they have been re-implemented in a cleaner way.

Eg : Pointers and Operator Overloading are not there in java but were an important part of C++.

2)Object Oriented

In java, everything is an object which has some data and behaviour. Java can be easily extended as it is based on Object Model. Following are some basic concept of OOP's.

i. Object ii class iii inheritance iv polymorphism v abstraction vi encapsulation

3)Robust

Java makes an effort to eliminate error prone codes by emphasizing mainly on compile time error checking and runtime checking. But the main areas which Java improved were Memory Management and mishandled Exceptions by introducing automatic **Garbage Collector** and **Exception Handling**.

4)Platform Independent

Unlike other programming languages such as C, C++ etc which are compiled into platform specific machines. Java is guaranteed to be write- once, run-anywhere language.On compilation Java program is compiled into bytecode. This bytecode is platform independent and can be run on any machine, plus this bytecode format also provide security. Any machine with

Java RuntimeEnvironment can run Java Programs

5)Secure

When it comes to security, Java is always the first choice. With java secure features it enable us to develop virus free, temper free system. Java program always runs in Java runtime environment with almost null interaction with system OS, hence it is more secure.

6)Multi Threading

Java multithreading feature makes it possible to write program that can do many tasks simultaneously. Benefit of multithreading is that it utilizes same memory and other resources to execute multiple threads at the same time, like While typing, grammatical errors are checked along.

7)Architectural Neutral

Compiler generates bytecodes, which have nothing to do with a particular computer architecture, hence a Java program is easy to intrepret on any machine.

8)Portable

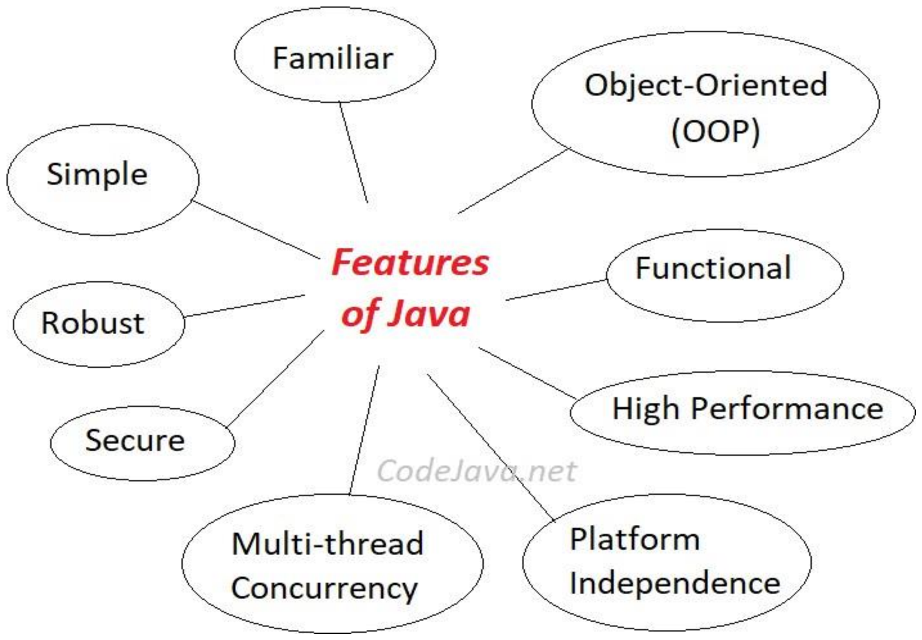
Java Byte code can be carried to any platform. No implementation dependent features. Everything related to storage is predefined, example: size of primitive data types

9)High Performance

Java is an interpreted language, so it will never be as fast as a compiled language like C or C++. But, Java enables high performance with the use of just-in-time compiler.

10)Distributed

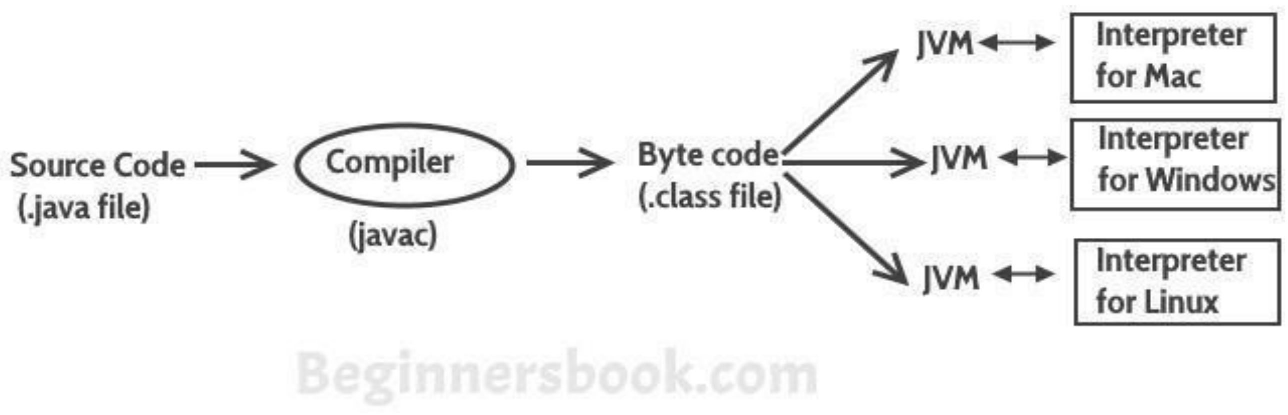
Java is also a distributed language. Programs can be designed to run on computer networks. Java has a special class library for communicating using TCP/IP protocols. Creating network connections is very much easy in Java as compared to C/C++.



➢JAVA Architecure :

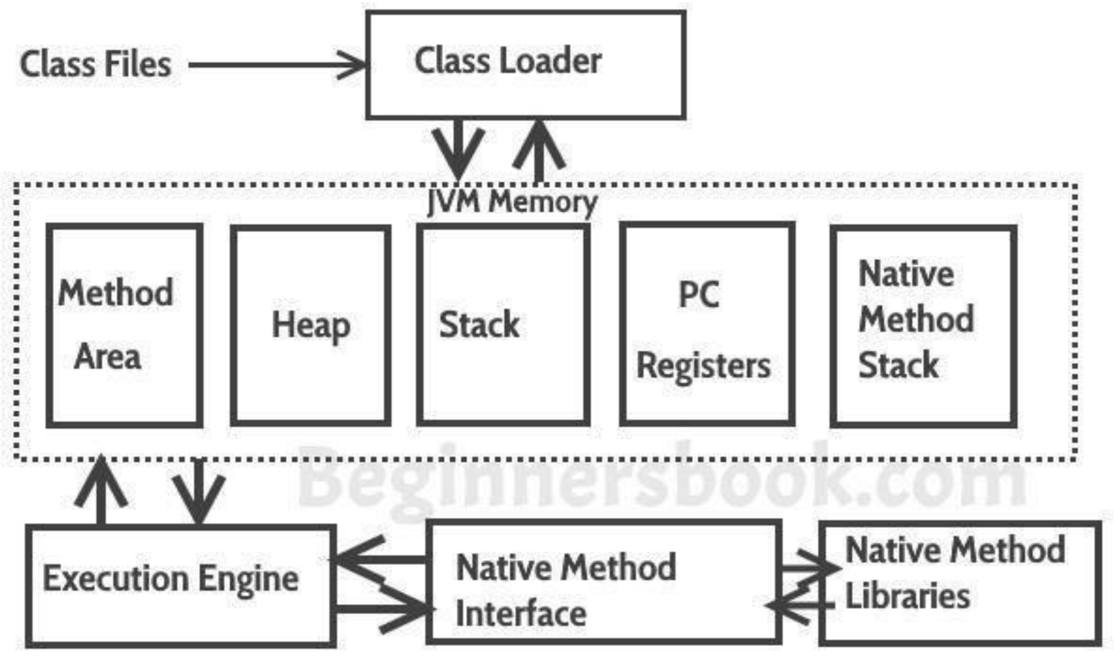
Java Virtual Machine (JVM), Difference JDK, JRE & JVM – Core Java Java is a high level programming language. A program written in high level language cannot be run on any machine directly. First, it needs to be translated into that particular machine language. The **javac compiler** does this thing, it takes java program (.java file containing source code) and translates it into machine code (referred as byte code or .class file).

Java Virtual Machine (JVM) is a virtual machine that resides in the real machine (your computer) and the **machine language for JVM is byte code**. This makes it easier for compiler as it has to generate byte code for JVM rather than different machine code for each type of machine.



So to summarise everything: The Java Virtual machine (JVM) is the virtual machine that runs on actual machine (your computer) and executes Java byte code. The JVM doesn't understand Java source code, that's why we need to have javac compiler that compiles *.java files to obtain *.class files that contain the byte codes understood by the JVM. JVM makes java portable (write once, run anywhere). Each operating system has different JVM, however the output they produce after execution of byte code is same across all operating systems.

JVM Architecture



Lets see how JVM works: **Class Loader:** The class loader reads the .class file and save the byte code in the **method area**.

Method Area: There is only one method area in a JVM which is shared among all the classes. This holds the class level information of each .class file.

Heap: Heap is a part of JVM memory where objects are allocated. JVM creates a Class object for each .class file.

Stack: Stack is a also a part of JVM memory but unlike Heap, it is used for storing temporary variables.

PC Registers: This keeps the track of which instruction has been executed and which one is going to be executed. Since instructions are executed by threads, each thread has a separate PC register.

Native Method stack: A native method can access the runtime data areas of the virtual machine.

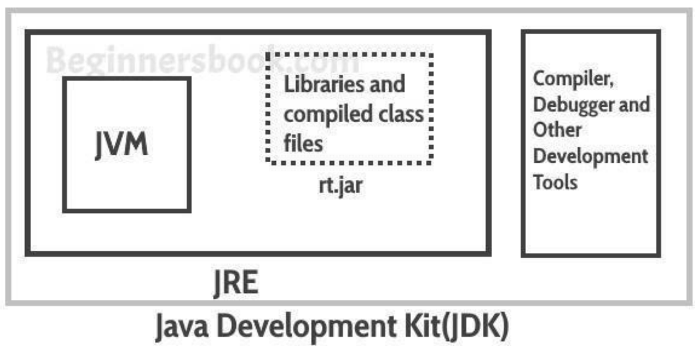
Native Method interface: It enables java code to call or be called by native applications. Native applications are programs that are specific to the hardware and OS of a system.

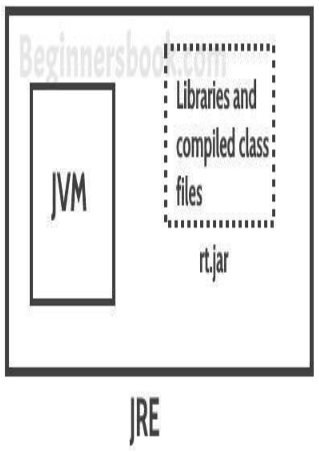
Garbage collection: A class instance is explicitly created by the java code and after use it is automatically destroyed by garbage collection for memory management.

JVM Vs JRE Vs JDK

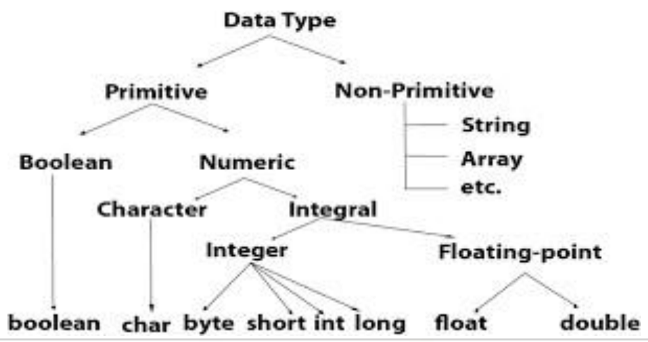
JRE: JRE is the environment within which the java virtual machine runs. JRE contains Java virtual Machine(JVM), class libraries, and other files excluding development tools such as compiler and debugger. Which means you can run the code in JRE but you can't develop and compile the code in JRE.

JVM: As we discussed above, JVM runs the program by using class, libraries and files provided by JRE.


Beginnersbook.com
JVM
Libraries and compiled class files
rt.jar
Compiler, Debugger and Other Development Tools
JRE
Java Development Kit(JDK)



JDK: JDK is a superset of JRE, it contains everything that JRE has along with development tools such as compiler, debugger etc. **Data types in java:**



Data Type	Size	Description
byte	1 byte	Stores whole numbers from -128 to 127
short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
boolean	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter or ASCII values

➢Java Variables:

Variables are containers for storing data values. In Java, there are different **types** of variables, for example:

- String - stores text, such as "Hello". String values are surrounded by double quotes
- int - stores integers (whole numbers), without decimals, such as 123 or -123
- float - stores floating point numbers, with decimals, such as 19.99 or -
  19.99

- char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
- boolean - stores values with two states: true or false

Declaring (Creating) Variables

To create a variable, you must specify the type and assign it a value:

Syntax ***type variable* = *value*;**

Where *type* is one of Java's types (such as int or String), and *variable* is the name of the variable (such as **x** or **name**). The **equal sign** is used to assign values to the variable.

To create a variable that should store text, look at the following example:

Example **Create a variable called** name **of type String and assign it the value "**John**":**

```
String name = "John"; System.out.println(name);
```

To create a variable that should store a number, look at the following example:

Example **Create a variable called** myNum **of type int and assign it the value** 15**:**

```
int myNum = 15; System.out.println(myNum);
```

You can also declare a variable without assigning the value, and assign the value later:

Example int myNum; myNum = 15; System.out.println(myNum);

Example Change the value of myNum from 15 to 20:

```
int myNum = 15; myNum = 20; // myNum is now 20

System.out.println(myNum);
```

Final Variables

However, you can add the final keyword if you don't want others (or yourself) to overwrite existing values (this will declare the variable as "final" or "constant", which means unchangeable and read-only):

Example

```
final int myNum = 15; myNum = 20;        // will generate an error: cannot
assign a value to a final variable
```

Other Types A demonstration of how to declare variables of other types:

Example

```
int myNum = 5; float myFloatNum = 5.99f; char myLetter =
'D';boolean myBool = true; String myText = "Hello";
```

Display Variables

The println() method is often used to display variables.To combine both text and a variable, use the + character**:**

Example

```
String name = "John"; System.out.println("Hello " + name);
```

You can also use the + character to add a variable to another variable:

Example

```
String firstName = "John ";String lastName = "Doe";

String fullName = firstName + lastName; System.out.println(fullName);
```

For numeric values, the + character works as a mathematical operator (notice that we use int (integer) variables here):

Example

```
int x = 5;int y = 6;System.out.println(x + y); // Print the value of x + y
```

From the example above, you can expect:

- x stores the value 5 y stores the value 6
- Then we use the println() method to display the value of x + y, which is **11**

Declare Many Variables **To declare more than one variable of the** same type**, use a comma-separated list:**

Example **int x = 5, y = 6, z = 50;System.out.println(x + y + z);**

JAVA Identifiers All Java **variables** must be **identified** with **unique names**.

These unique names are called **identifiers**.Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

Example int m = 60;

The general rules for constructing names for variables (unique identifiers) are:

- Names can contain letters, digits, underscores, and dollar signs
- Names must begin with a letter
- Names should start with a lowercase letter and it cannot contain whitespace
- Names can also begin with $ and _ (but we will not use it in this tutorial)
- Names are case sensitive ("myVar" and "myvar" are different variables)
- Reserved words (like Java keywords, such as int or boolean) cannot be used as names

➢Java Operators

Operators are used to perform operations on variables and values.In the example below, we use the + **operator** to add together two values: Example int x = 100 + 50; Although the + operator is often used to add together two values, like in the example above, it can also be used to add together a variable and a value, or a variable and another variable:

Example **int sum1 = 100 + 50; int sum2 = sum1 + 250; int sum3 = sum2 + sum2;**

Java divides the operators into the following groups:

Arithmetic operators * Assignment operators *Comparison operators *Logical operators * Bitwise operators

Arithmetic Operators Arithmetic operators are used to perform common mathematical operations.

Operator	**Name**	**Description**	**Example**	
+	Addition	Adds together two values	x + y	

-	Subtraction	Subtracts one value from another	x - y	
*	Multiplication	Multiplies two values	x * y	
/	Division	Divides one value by another	x / y	
%	Modulus	Returns the division remainder	x % y	
++	Increment	Increases the value of a variable by 1	++x	
--	Decrement	Decreases the value of a variable by 1	--x	

Java Assignment Operators Assignment operators are used to assign values to variables.In the example below, we use the **assignment** operator (=) to assign the value **10** to a variable called **x**:

Example **int x = 10**;

The **addition assignment** operator (+=) adds a value to a variable:

Example **int x = 10**;**x += 5**;

A list of all assignment operators:

Operator	Example	Same As

=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Java Comparison Operators **Comparison operators are used to compare two values:**

Operator	**Name**	**Example**

==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Java Logical Operators **Logical operators are used to determine the logic between variables or values:**

Operator	**Name**	**Description**	**Example**
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

➢Control Structures

In the most basic sense, a program is a list of instructions. Control structures are programming blocks that can change the path we take through those instructions.

There are three kinds of control structures:

- Conditional Branches, which we use **for choosing between two or more paths.** There are three types in Java: *if/else/else if*, *ternary operator* and *switch*.
- Loops that are used to **iterate through multiple values/objects and repeatedly run specific code blocks.** The basic loop types in Java are *for*, *while* and *do while*.
- Branching Statements, which are used to **alter the flow of control in loops.** There are two types in Java: *break* and *continue*.

If/Else/Else If

The *if/else* statement isthemostbasicofcontrolstructures, but can also be considered the very basis of decision making in programming.

While *if* can be used by itself, the most common use-scenario is choosing between two paths with *if/else*:

```
if (count >2) {
  System.out.println("Count is higher than 2");
} else {
  System.out.println("Count is lower or equal than 2");
}
```

Theoretically, we can infinitely chain or nest ***if/else*** blocks but this will hurt code readability, and that's why it's not advised.

We'll explore alternative statements in the rest of this article.
Ternary Operator

We can**useaternaryoperator**as a shorthand expression that works like an ***if/else*** statement**.**

Let's see our *if/else* example again:

```
if (count >2) { System.out.println("Count is higher than 2");
} else { System.out.println("Count is lower or equal than 2");}
```

We can refactor this with a ternary as follows:

```
System.out.println(count >2 ? "Count is higher than 2" : "Count is lower or
equal than 2");
```

While ternary can be a great way to make our code more readable, it isn't always a good substitute for *if/else.*

Switch If we have multiple cases to choose from, we can use *switch* **statement.**Let's again see a simple example:

```
int count = 3; switch (count) { case 0:
System.out.println("Count is equal to 0"); break;
case 1: System.out.println("Count is equal to 1")
break;
default: System.out.println("Count is either negative, or higher than
1");break;}
```

Three or more *if/else* statements can be hard to read. As one of the possible workarounds, we can use *switch,* as seen above.

And also keep in mind that *switch* has scopeand input limitationsthat we need to remember before using it.

Loops

We use**loops**when we need to repeat the same code multiple times in succession.

Let's see a quick example of comparable *for* and *while* type of loops:

```
for (int i = 1; i <= 50; i++) { methodToRepeat();}

int whileCounter = 1;
while (whileCounter <= 50) { methodToRepeat() whileCounter++;}
```

Both code blocks above will call *methodToRepeat* 50 times.

Break

We need to use ***break*** to exit early from a loop**.** Let's see a quick example:

```
List<String>          names          =
getNameList();
String name = "John Doe";
int index = 0;
for ( ; index < names.length; index++)
{
  if (names[index].equals(name)) { break; }}
```

Here, we are looking for a name in a list of names, and we want to stop looking once we've found it.

Continue

Simply put,***continue***means to skip the rest of the loop we're in:

```
List<String> names = getNameList();
String name = "John Doe"; String
list = ""; for (int i = 0; i <
names.length; i++) {   if
(names[i].equals(name)) {
continue;
  }list += names[i];}
```

Here, we skip appending the duplicate *names* into the list.

➢Java Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type with **square brackets**:

```
String[] cars;
```

We have now declared a variable that holds an array of strings. To insert values to it, we can use an array literal - place the values in a comma- separated list, inside curly braces:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

To create an array of integers, you could write:

```
int[] myNum = {10, 20, 30, 40};
```

Access the Elements of an Array

You access an array element by referring to the index number.

This statement accesses the value of the first element in cars:

Example **String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};**
System.out.println(cars[0]); // Outputs Volvo

Note: Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

Change an Array Element

To change the value of a specific element, refer to the index number:

Example**cars[0] = "Opel";**

Example **String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};**
cars[0]= "Opel";

```
System.out.println(cars[0]); // Now outputs Opel instead of Volvo
```

Array Length To find out how many elements an array has, use the length property:

Example **String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};**
System.out.println(cars.length); // Outputs

Loop Through an Array

You can loop through the array elements with the for loop, and use the length property to specify how many times the loop should run.
The following example outputs all elements in the **cars** array:

Example **String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};**

```
for (int i = 0; i <cars.length; i++) { System.out.println(cars[i]);
```

Loop Through an Array with For-Each

There is also a "**for-each**" loop, which is used exclusively to loop through elements in arrays:

Syntax

```
for (type variable : arrayname) {}
```

The following example outputs all elements in the **cars** array, using a "**foreach**" loop:

Example **String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};**

```
for (String i : cars) { System.out.println(i);}
```

The example above can be read like this: **for each** String element (called **i** - as in **i**ndex) in **cars**, print out the value of **i**.

If you compare the for loop and **for-each** loop, you will see that the **for- each** method is easier to write, it does not require a counter (using the length property), and it is more readable.

Multidimensional Arrays

A multidimensional array is an array containing one or more arrays.

To create a two-dimensional array, add each array within its own set of **curly braces**:Example `int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };`

myNumbers **is now an array with two arrays as its elements.**

To access the elements of the **myNumbers** array, specify two indexes: one fo**r the array, and one for the element inside that array. This example accesses the third element (2) in the second array (1) of myNumbers:**

```
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} }; int x = myNumbers[1][2];
System.out.println(x); // Outputs 7
```

We can also use a for loop inside another for loop to get the elements of a two-dimensional array (we still have to point to the two indexes): Example is

```
public class Main { public static void main(String[] args) {
    int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };for
    (int i = 0; i <myNumbers.length; ++i) {for(int j = 0; j
    <myNumbers[i].length; ++j) {
    System.out.println(myNumbers[i][j]);}}}
```

➢Control Structures :

Control Statements in Javais one of the fundamentals required for Java Programming. It allows the smooth flow of a program. Following pointers will be covered in this article:Decision making statements simple if statement ,if- else statement ,Nested if statement, Switch statement, Looping statements, while,Do-while,For ,For-each ,Branching statements ,Break, Continue

Control Statements can be divided into three categories, namely

Selection statements ,Iteration statements ,Jump statements

Decision-MakingStatements Statements that determine which statement to execute and when are known as decision-making statements. The flow of the execution of the program is controlled by the control flow statement. There are four decision-making statements available in java.

Simple if statement

The if statement determines whether a code should be executed based on the specified condition.
Syntax:

```
if (condition) {

Statement 1; //executed if condition is true}

Statement 2; //executed irrespective of the condition
```

Output:

```
If                                                        statement!
Hello World!
```

Moving on with this article on Control Statements in Java

If..else statement

In this statement, if the condition specified is true, the if block is executed. Otherwise, the else block is executed.

Example:

```
public class Main{public static void

main(String args[]){ int a = 15;if (a

> 20)

System.out.println("a is greater than 10");

else

System.out.println("a is less than 10");

System.out.println("Hello World!");}}}
```

Output:

```
a              is              less              than              10
Hello World!
```

Nested if statement

An if present inside an if block is known as a nested if block. It is similar to an if..else statement, except they are defined inside another if..else statement.

Syntax:

```
if (condition1) {

Statement 1; //executed if first condition is true if

(condition2) {

Statement 2; //executed if second condition is true} else {

Statement 3; //executed if second condition is false}}
```

Example:

```
public  class  Main{public  static  void

main(String args[]){ int s = 18; if (s

> 10) if (s%2==0)

System.out.println("s is an even number and greater than 10!");

else

System.out.println("s is a odd number and greater than 10!");}

Else {System.out.println("s is less than 10")}

System.out.println("Hello World!");}}
```

Output:
```
s is an even number and greater than 10!
Hello World!
```

Switch statement

A switch statement in java is used to execute a single statement from multiple conditions. The switch statement can be used with short, byte, int, long, enum types, etc. Certain points must be noted while using the switch statementɑ One or N number of case values can be specified for a switch expression. ɑ Case values that are duplicate are not permissible. A compile-time error is generated by the compiler if unique values are not used. ɑ The case value must be literal or constant. Variables are not permissible. ɑ Usage of break statement is made to terminate the statement sequence. It is optional to use this statement. If this statement is not specified, the next case is executed.

Example:

```
public   class   Music   {public
static    void    main(String[]
args){int instrument = 4;
```

```
String musicInstrument;
// switch statement with int data type switch (instrument)
{case 1:musicInstrument = "Guitar";break; case
2:musicInstrument = "Piano";break; case
3:musicInstrument = "Drums";break; case
4:musicInstrument = "Flute";break; case
5:musicInstrument = "Ukelele";break; case
6:musicInstrument = "Violin";break;case
7:musicInstrument = "Trumpet";break;default:
musicInstrument = "Invalid";break;}
System.out.println(musicInstrument);}}
```

Output:

```
Flute
```

Looping Statements

Statements that execute a block of code repeatedly until a specified condition is met are known as looping statements. Java provides the user with three types of loops:

While

Known as the most common loop, the while loop evaluates a certain condition. If the condition is true, the code is executed. This process is continued until the specified condition turns out to be false. The condition to be specified in the while loop must be a Boolean expression. An error will be generated if the type used is int or a string.

Syntax:

```
while (condition){statementOne;}
```

Example:

```
public class whileTest

{public static void main(String args[])

{int i = 5;while (i <= 15)

{System.out.println(i);i = i+2;}}}
```

Output:

```
5 7 9 11 13 15
```

Do..while

The do-while loop is similar to the while loop, the only difference being that the condition in the do-while loop is evaluated after the execution of the loop body. This guarantees that the loop is executed at least once.

Syntax:

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

Example:

```
public class Main
{ public static void main(String args[])
{ int i = 20; do
{ System.out.println(i); i = i+1; } while (i <= 20); }
```

Output:

```
20
```

Moving on with this article on Control Statements in Java The for loop in java is used to iterate and evaluate a code multiple times. When the number of iterations is known by the user, it is recommended to use the for loop.

Syntax:

```
for (initialization; condition; increment/decrement)
{statement;}
```

Example:

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

Output:

```
5
6
7
8
9
10
```

For-Each

The traversal of elements in an array can be done by the for-each loop. The elements present in the array are returned one by one. It must be noted that the user does not have to increment the value in the for-each loop.

Example:

```
public class foreachLoop{public

static void main(String args[]){

int s[] = {18,25,28,29,30};for

(int i : s)

{System.out.println(i);}}}
```

Output:

```
18
25
28
29
30
```

Branching Statements

Branching statements in java are used to jump from a statement to another statement, thereby the transferring the flow of execution.

Break

The break statement in java is used to terminate a loop and break the current flow of the program.

Example:

```
public class Test

{ public static void main(String args[]) {for (int i = 5; i <

10; i++){if (i == 8)break;System.out.println(i);

}}}
```

Output:

```
5
```

6
7

Continue

To jump to the next iteration of the loop, we make use of the continue statement. This statement continues the current flow of the program and skips a part of the code at the specified condition.

Example:

```
public class Main

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

for (int k = 5; k < 15; k++)

{if (k%2 != 0)continue; System.out.print(k + " ");}}}
```

Output:

```
6 8 10 12 14
```

➢Arrays

An array is a grouping of the same typed values in computer memory. In Java, the data stored in an array can be a primitive or an object. As a quick refresher, primitive data types in Java are byte, short, int, long, float, double, or character. When storing primitives in an array Java stores these in adjacent blocks of memory. An element is an individual value in an array. The index is the location of an element in the array.

Creating an array

In Java, there are two ways to create an array. The first way is using the new keyword as shown below.int[] array = new int[10];The other way to create an array is using a shortcut syntax. The shortcut syntax allows us to instantiate the array with exactly the values we need. Using this syntax, we skip having to do the costly operation of looping through the array to insert an element at each index.

double[] array = {10.2, 15.4, 20.6, 25.8};

If you don't know the elements that you want to put in your array when you instantiate it go with the new keyword. It's important to note that no matter how we create an array we need to know the size we want to allocate to the array. The indexes of the array are zero-indexed. If you have an array length of 10 the first element of this array will actually be at the index of 0 and the last index will be 9.

```
float[] array = new float[10];
```

Finding array length

Finding the length of an array is essential for using arrays. Without knowing the length of an array a programmer won't know when to stop looping through an array. Trying to access an index outside of the array, index 11 in an array of length 10 for example, results in an out of bounds exception in Java. Lucky for us arrays have a length property. The length property makes finding the array as easy as calling .length on your array. There are no parentheses as this is a property, not a function, on the array. The length is one based meaning in the example below our length would return 3 but the indexes in the array would be 0, 1, and 2.

```
int[] array = {22, 33, 44};
```

array.length When first learning arrays in Java it can be easy to get tripped up on the length property being one based. Remember when trying to find
where an array ends you will always want the length -

Java array utility class

Java includes a utility class.

```
public class ArrayTest{

  public static void main(String args[]){

  int[] array = {1,2,3}; Arrays.sort(array); } }
```

➢Java Classes and Objects

Java is an object-oriented programming language.Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.

A Class is like an object constructor, or a "blueprint" for creating objects. **Create a Class** To create a class, use the keyword class:

Main.java **Create a class named "Main" with a variable x:**

```
public class Main {int x = 5;}
```

Create an Object In Java, an object is created from a class. We have already created the class named MyClass, so now we can use this to create objects. To create an object of MyClass, specify the class name, followed by the object name, and use the keyword new:

Example **Create an object called "myObj" and print the value of x:**

```
public class Main { int x = 5;public static void main(String[] args)
{Main myObj = new Main();
 System.out.println(myObj.x);}}
```

Multiple Objects: You can create multiple objects of one class:

Example **Create two objects of Main:**

```
public class Main {int x = 5;public static void main(String[] args) {
    Main myObj1 = new Main(); // Object 1
    Main myObj2 = new Main(); // Object 2
    System.out.println(myObj1.x);System.out.println(myObj2.x);}}
```

Using Multiple Classes

You can also create an object of a class and access it in another class. This is often used for better organization of classes (one class has all the attributes and methods, while the other class holds the main() method (code to be executed)).

Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory/folder:

- Main.java
- Second.java

```
Main.java public class Main {int x = 5;}

Second.java class Second {public static void main(String[] args) {

   Main myObj = new Main(); System.out.println(myObj.x);} }
```

Wrapper classes in Java

The **wrapper class in Java** provides the mechanism *to* convert primitive into object and object into primitive*.* Since J2SE 5.0, **autoboxing** and **unboxing** feature *convert* primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Use of Wrapper classes in Java

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

○ **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.

- **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- **Synchronization:** Java synchronization works with objects in Multithreading.
- **java.util package:** The java.util package provides the utility classes to deal with objects.
- **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

The eight classes of the *java.lang* package are known as wrapper classes in Java. The list of eight wrapper classes are given below:

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

Auto boxing

Auto boxing the auto matic conversion of primitive data type into its corresponding wrapper class is known as autoboxing,for example,byte to byte, char to character ,int to integer ,long to long,float to Float,Boolean to Boolean,double to Double, and short to Short

Wrapper class Example :Primitive Wrapper:

```
Public class WrapperExample1{

Public static void main(String args[]){

Int a=20; Integer i=Integer.valueOf(a);Integer j=a;

System.out.println(a+""+i+++);}} output:20 20 20
```
Unboxing :

The automatic conversion of wrapper type into its correspondingprimitivetype is known as unboxing.It is the reverse process of autoboxing.

➢Constructors

1.Typesofconstructors

1. DefaultConstructor
2. ParameterizedConstructor

2. ConstructorOverloading
3. Doesconstructorreturnanyvalue?
4. Copyingthevaluesofoneobjectintoanother
5. Doesconstructorperform othertasksinsteadoftheinitialization

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the classis created. At the time of calling constructor, memory for the object is allocated in the memory.

It is a special type of method which is used to initialize the object.Every time an object
is created using the new() keyword, at least one constructor is called.It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

Rules for creating Java constructor

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

Types of Java constructors

There are two types of constructors in Java:

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

Java Default constructor: “A constructor does not having any parameter”.Main purpose of these one is to provide the default values to the object like0,null,etc,depending on the type

Syntax: class name(){}

Class Bike{Bike()S.o.p(“Bike is created”);} public static void main(String args[]){

Bike1 b=new Bike();}} Output:Bike is created.

Java Parameterized Constructor

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

Why use the parameterized constructor?

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also

Constructor Overloading in Java

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

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

Java Copy Constructor

There is no copy constructor in Java. However, we can copy the values from one object to another like copy constructor in C++.
There are many ways to copy the values of one object into another in Java. They are:

- By constructor ○By assigning the values of one object into another
- By clone() method of Object class

In this example, we are going to copy the values of one object into another using Java constructor.

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

➢Overloading of methods

1. Differentwaystooverloadthe method
2. Bychangingtheno.ofarguments
3. Bychangingthedatatype
4. Whymethodoverloadingisnotpossiblebychangingthereturntype

5. Canweoverloadthemainmethod
6. methodoverloadingwithType Promotion

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

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

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

So, we perform method overloading to figure out the program quickly.

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 methodsso that we don't need to create instance for calling methods.

```
class
 Add
 er{
```

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

2)Method Overloading: changing data type of arguments

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

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

Can we overload java main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. ButJVMcalls main() method which receives string array as arguments only.

Why Method Overloading is not possible by changing the return type of method only?

In java, method overloading is not possible by changing the return type of the method only because of ambiguity.

➢Access Control

Access control specifies the accessibility of code. By using these you can specify the scope of data, method, class etc.There are 3 types of access control in :

1. Public or Default
2. Private
3. Internal

Public Access Control Example

Public access control also known as default. It is accessible for all. If you don't specify any access control explicitly in your code, by default it follows public access control.

```
type AccessControl() =
member public x.a = 10
member public x.display() = printfn "This is public method"
let ac = new AccessControl()        ac.display()
printfn "a = %d" ac.a Output:This is public methoda = 10
```

Private Access Control Example

Private access control is most restricted and has limited scope. F# provides **private** keyword to declare private members in the class or type. The scope of private is limited to the local block in which it is declared.

1. **type** AccessControl() = **member private** x.a = 10
2. **member private** x.display() = printf "This is private method"
3. **let** ac = **new** AccessControl() ac.display()   printf "a= %d" ac.a
   .

Output: error FS0491: The member or object constructor 'display' is not accessible.

Private members may only be accessed from within the declaring type.

Internal Access Control

Internal access control is only accessible from the same assembly. An assembly is a file which is automatically generated by your compiler after compilation of F# code. It can be either Dynamic Link Library (DLL) or Executable File (exe).

1. **type** AccessControlIN()**member internal** x.a = 10 **member internal** x.display() =
2. printfn "This is internal method"
3. **let** aci = **new** AccessControlIN() aci.display() printfn "a= %d" aci.a

Output:This is internal method A

➢Nested class and Inner class:

Nested class

Nested class is such class which is created inside another class. In Kotlin, nested class is by default **static**, so its data member and member function can be accessed without creating an object of class. Nested class cannot be able to access the data member of outer class.

```
class outerClass{ //outer class
   codeclass nestedClass{
   //nested class code } }
```

Inner class

Inner class is a class which is created inside another class with keyword **inner**. In other words, we can say that a nested class which is marked as **"inner"** is called inner class.

Inner class cannot be declared inside interfaces or non-inner nested classes.

```
class outerClass{ //outer class code inner class
   innerClass{ //nested class code  } }
```

The advantage of inner class over nested class is that, it is able to access members of outer class even it is private. Inner class keeps a reference to an object of outer class

Abstract class in Java

A class which is declared with the abstract keyword is known as an abstract class inJava. It can have abstract and non-abstract methods

➢Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user. Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstract class in Java: Aclass which is declared as abstract is known as an abstract class itcan have abstract and non-abstract methods.It needs to be extended and its method implemented.

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

Ex:Abstract Method in java :Amethod which is declared as abstract and does not have implementation is known as an abstract method.

Abstract void printStatus();no method body and abstract
Ex:abstract class Bike{

Abstract void run();}

Class Honda4 extends Bike{

Void run(){System.out.println("running safely");

Public static void main(String args[]{Bike obj=new Honda4();

Obj.run();}}

Another example of Abstract class in java

File: TestBank.java

```
1. abstract class Bank{    abstract int getRateOfInterest();   }
2. class SBI extends Bank{       int  getRateOfInterest(){return  7;}
   }
3. class PNB extends Bank{       int getRateOfInterest(){return 8;} }
4. class TestBank{    public static void main(String args[]){
5. Bank b; b=new SBI();
6. System.out.println("Rate  of  Interest  is:  "+b.getRateOfInterest()+"
   %");
7. b=new PNB();
8. System.out.println("Rate  of  Interest  is:  "+b.getRateOfInterest()+"
   %");}
   }Rate of Interest is: 7 % Rate of interest is :8%
```

➢Inheritance in Java

1. Inheritance
2. TypesofInheritance
3. Whymultipleinheritanceisnot possibleinJavaincaseofclass?

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. The idea behind inheritance in Java is that you can create new classesthat are built upon existing classes. . Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.
Why use inheritance in

java○ForMethodOverriding(soruntimepolymorphismcan be

achieved). ○For Code Reusability. ○Terms used in Inheritance

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

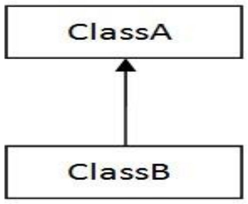
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

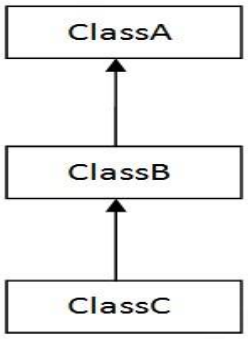
```
1.class Subclass-name extends Superclass-name
  { //methods and fields }
```

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

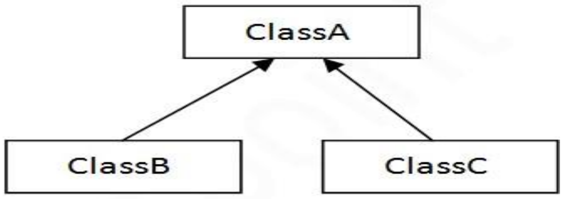
Types of inheritance in java : those are 3 types (basis of class)ie single,multilevel and hierarchical .In java programming,multiple and hybrid inheritance is supported through interface only



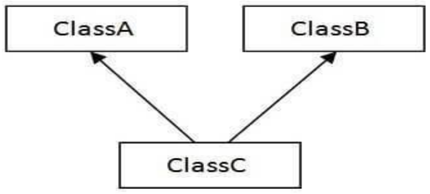
1) Single



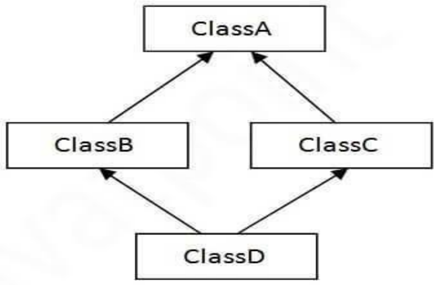
2) Multilevel



3) Hierarchical



4) Multiple



5) Hybrid

.

➢Super Keyword in Java:

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

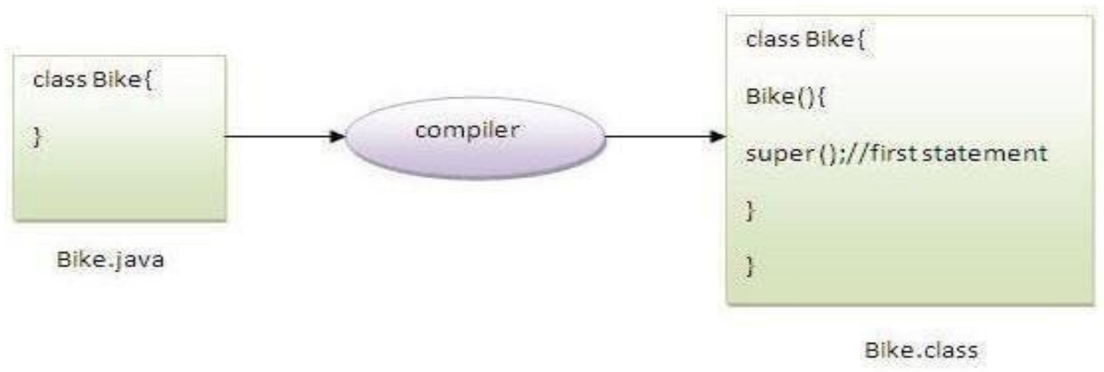
Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

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.



Example of super keyword

1. **class** Person{ **int** id; String name; Person(**int** id,String name){
2. **this**.id=id; **this**.name=name; } }
3. **class** Emp **extends** Person{ **float** salary;
4. Emp(**int** id,String name,**float** salary){
5. **super**(id,name);//reusing parent constructor **this**.salary=salary; }
6. **void** display(){System.out.println(id+" "+name+" "+salary);} }
7. **class** TestSuper5{
8. **public static void** main(String[] args){
9. Emp e1=**new** Emp(1,"ankit",45000f);
10. e1.display(); }} Output: 1 ankit 45000

Multilevel Inheritance Example (or) multilevel hierarchy

When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

File: TestInheritance2.java

1. **class** Animal{ **void** eat(){System.out.println("eating...");}

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

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

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

➢Method Overriding:

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

Rules for Java Method Overriding

1.The method must have the same name as in the parent class 2.The method must have the same parameter as in the parent class.

3.There must be an IS-A relationship (inheritance).

Example of method overriding

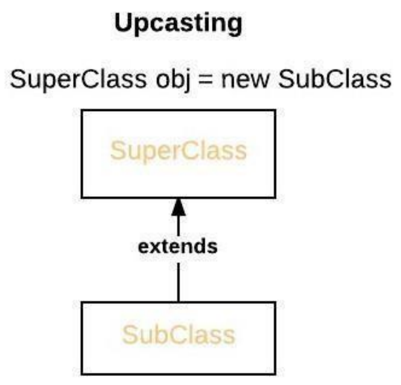
In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

```
class Vehicle{
void run(){System.out.println("Vehicle is running");}
class Bike2 extends Vehicle{
void run(){System.out.println("Bike is running safely");}
public static void main(String args[]){
Bike2 obj = new Bike2();//creating object
obj.run();//calling method  } } Output:Bike is running safely
```

➢Dynamic Method Dispatch or Runtime Polymorphism in Java

Method overriding is one of the ways in which Java supports Runtime Polymorphism. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

- When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs.
- At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed
- A superclass reference variable can refer to a subclass object. This is also known as upcasting.



Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

Advantages of Dynamic Method Dispatch

1. Dynamic method dispatch allow Java to supportoverridingofmethodswhich is central for run-time polymorphism.
2. It allows a class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
3. It also allow subclasses to add its specific methods subclasses to define the specific implementation of some.

StaticvsDynamicbinding

- Static binding is done during compile-time while dynamic binding is done during run-time.
- private, final and static methods and variables uses static binding and bonded by compiler while overridden methods are bonded during runtime based upon type of runtime object

➢Using final with Inheritance in Java

finalis a keyword in java used for restricting some functionalities. We can declare variables, methods and classes with final keyword.**Using final with inheritance:**
During inheritance, we must declare methods with final keyword for which we required to follow the same implementation throughout all the derived classes. Note that it is not necessary to declare final methods in the initial stage of inheritance.

final keyword with inheritance in java

- The **final keyword** is final that is we cannot change.
- We can use **final keywords** for variables, methods, and class.
- If we use the **final keyword for the inheritance** that is if we declare any method with the **final keyword** in the base class so the implementation of the **final method** will be the same as in derived class.
- We can declare the **final method** in any subclass for which we want that if any other class extends this subclass.

Short Questions

1.Explain the concepts of object oriented programming .
2.what is Access control in java?
3.what is dynamic method dispatch?
4.what is method oveoverriding?

Long Questions

1.what is inheritance? Explain various types of inheritance.
2.Explain the differences between method overloading and method overriding
3.What is a constructor? Explain different types of constructors
4.what is final with inheritance? Explain breifly
5.what is nested and inner classes. Explain

UNIT-II

➢Java Math class

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

Unlike some of the StrictMath class numeric methods, all implementations of the equivalent function of Math class can't define to return the bit-for-bit same results. If the size is int or long and the results overflow the range of value, the methods addExact(), subtractExact(), multiplyExact(), and toIntExact() throw an ArithmeticException.

```
publicclass JavaMathExample1
 { publicstaticvoid main(String[] args) { double x = 28;  double y = 4;


   System.out.println("Maximum number of x and y is: "
+Math.max(x, y));
System.out.println("Square root of y is: " + Math.sqrt(y));
    System.out.println("Power of x and y is: " + Math.pow(x, y));
    System.out.println("Logarithm of x is: " + Math.log(x));
     System.out.println("Logarithm of y is: " + Math.log(y));
     System.out.println("log10 of x is: " + Math.log10(x));
    System.out.println("log10 of y is: " + Math.log10(y));
    System.out.println("log1p of x is: " +Math.log1p(x));
    System.out.println("exp of a is: " +Math.exp(x));
    System.out.println("expm1 of a is: " +Math.expm1(x)); }    }
```

Output:

```
Maximum number of x and y is: 28.0 Square root of y is: 2.0
Power of x and y is: 614656.0 Logarithm of x is: 3.332204510175204
Logarithm of y is: 1.3862943611198906 log10 of x is: 1.4471580313422192
log10 of y is: 0.6020599913279624 log1p of x is: 3.367295829986474 exp
of a is: 1.446257064291475E12 expm1 of a is: 1.446257064290475E1
```

Java Math Methods

The java.lang.Math class contains various methods for performing basic numeric operations such as the logarithm, cube root, and trigonometric functions etc. The various java math methods are as follows: **BasicMath methods :** Math.abs(),Math.max(),Math.min(),Math.round(),Math.sqrt(),Math.cbrt(),Math.pow(),Math.floor(),Math.nextAfter(),Math.ceil(),Math.copySign(),Math.random(),Math.toIntExact()

LogarithmicMathMethods: **Math.log(),Math.log10(),Math.log1p(),Math.exp(),Math.expm1()**

Trigonometric Math Methods: **Math.sin(),Math.cos(),Math.tan(),Math.asin(),,Math.acos(),Math.atan()**

Hyperbolic Math Methods: **Math.sinh(),Math.cosh(),Math.tanh()**

Angular Math Methods:**Math.toDegrees,Math.toRadians**

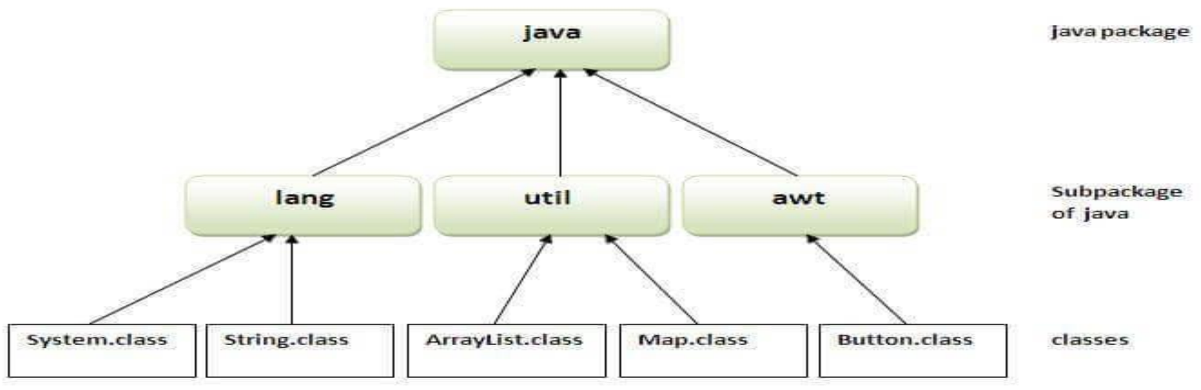
➢Java Package

A **java package** is a group of similar types of classes, interfaces and subpackages.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.



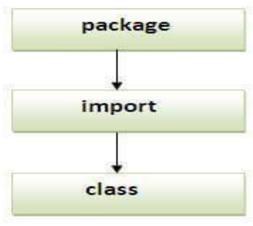
Simple example of java package The **package keyword** is used to create a package in java.

package mypack; **publicclass** Simple{ **publicstaticvoid** main(String a rgs[]){System.out.println("Welcome to package"); }}

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.2) Using packagename.classname If you import package.classname then only declared class of this package will be accessible. 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.

Subpackage in java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on

➢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 multipleinheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body. Java Interface also **represents the IS-A relationship**.It cannot be instantiated just like the abstract class.

Why use Java interface?

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

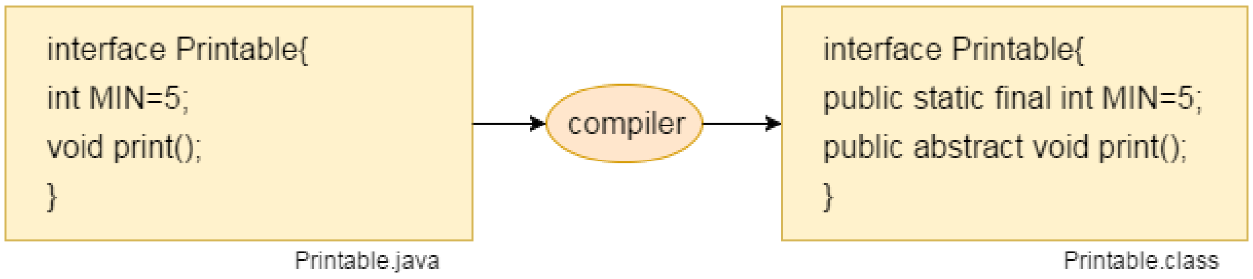
- It is used to achieve abstraction. By interface, we can support the functionality of multiple inheritance. It can be used to achieve loose coupling.

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

Syntax:

1.**interface**<interface_name>{

In other words, Interface fields are public, static and final by default, and the methods are public and abstract.

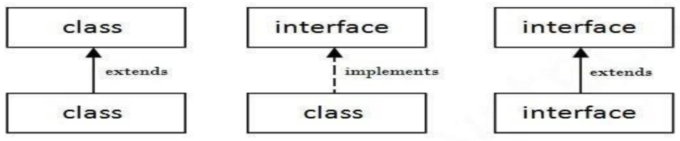


```
interface Printable{
int MIN=5;
void print();
}
```

```
interface Printable{
public static final int MIN=5;
public abstract void print();
}
```

The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



Java Interface Example In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

```
1.   interface printable{  void print();  }   class A6 implements printable{
2.   publicvoid print(){System.out.println("Hello");}
3.   publicstaticvoid main(String args[]){   A6 obj = new A6();  obj.print();
      }  }
```

Multiple inheritance in Java by interface

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

1. **interface** Printable{ **void** print(); } **interface** Showable{ **void** show(); }
2. **class** A7 **implements** Printable,Showable{
3. **publicvoid** print(){System.out.println("Hello");}
4. **publicvoid** show(){System.out.println("Welcome");} **publicstaticvoid** main(String args[]){
5. A7 obj = **new** A7(); obj.print(); obj.show(); }}

What is marker or tagged interface?An interface which has no member is known as a marker or tagged interface, for example,Serializable, Cloneable, Remote, etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

➢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.

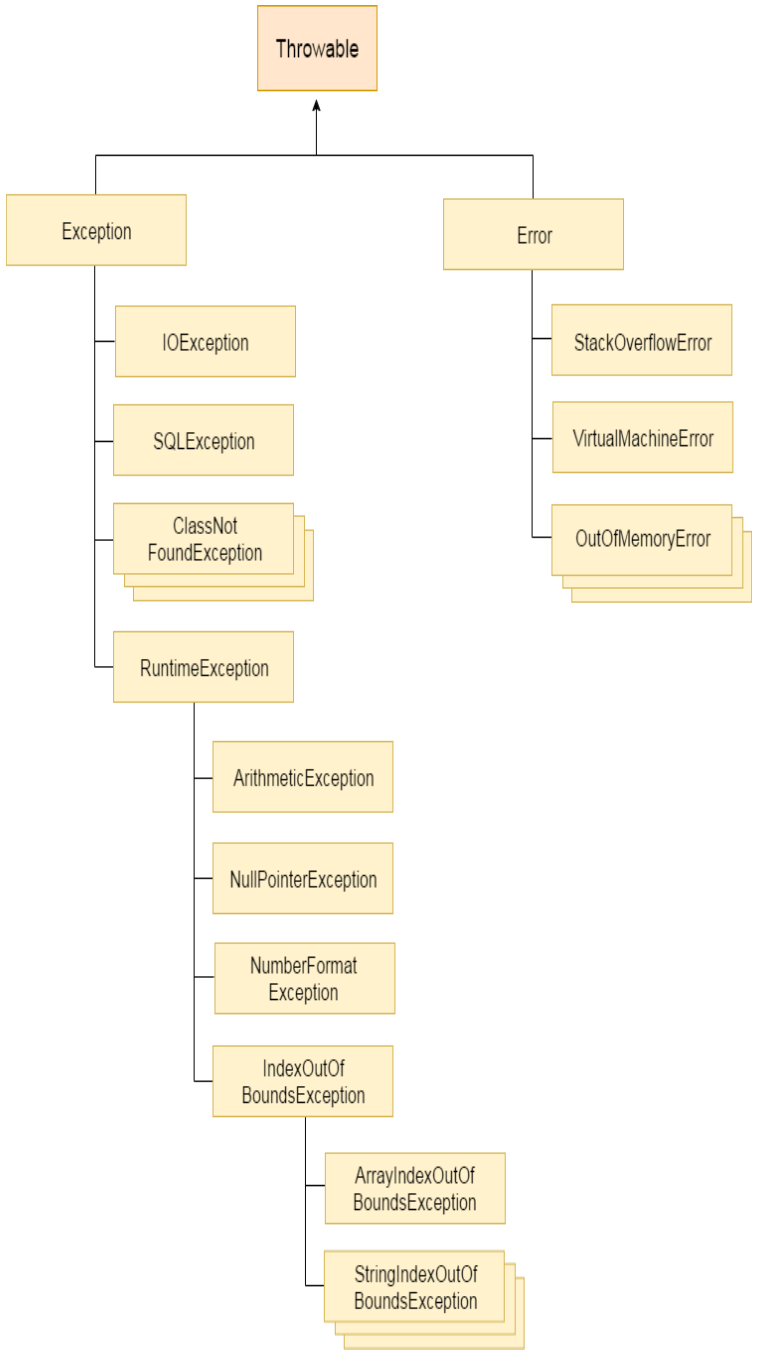
What is Exception Handling Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

1.statement 1; 2 statement 2; 3 statement 3;//exception ocuurs 4 statement 4 ;

5 statement 5; 6 statement 6;

Suppose there are 6 statements in your program and there occurs an exception at statement 3.the rest of the code will not be executed i.e. statement 4 to 6 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling inJava.

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



Unchecked Exception
Checked Exception
Error
Exception Handling

➢Types of Java Exceptions

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

1.Checked Exception 2 Unchecked Exception 3 Error

Difference between Checked and Unchecked Exceptions

1) Checked Exception The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compiletime.

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

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

Java Exception Keywords

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

Try :It specify a block where we should place exception code .it must be followed by either catch or finally ie canot use try block alone.

Catch : The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later

Finally :The “finally”block is used to execute the important code of the program .it executes whether an exception is handled or not.

Throw : The "throw" keyword is used to throw an exception

Throws: The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Java Exception Handling Example

Let's see an example of Java Exception Handling where we using a try-catch statement to handle the exception.

```
publicclass JavaExceptionExample{
publicstaticvoid main(String args[]){
try{        int     data=100/0;         }catch(ArithmeticException
e){System.out.println (e);}
System.out.println("rest of the code...");  }  }
```

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

```
rest of the code...
```

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1)A scenario where ArithmeticException occurs **If we divide any number by zero, there occurs an ArithmeticException.**

2) A scenario where NullPointerException occurs

If we have a null value in anyvariable, performing any operation on the variable throws a NullPointerException.

3) A scenario where NumberFormatException occurs

The wrong formatting of any value may occur NumberFormatException. Suppose I have astringvariable that has characters, converting this variable into digit will occur NumberFormatException.

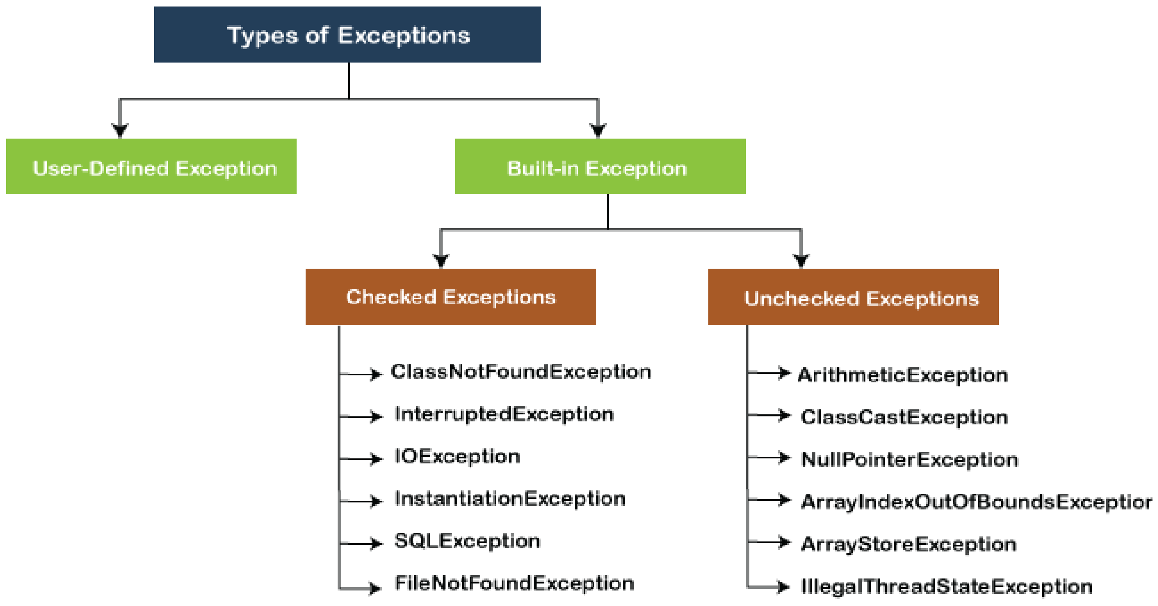
4) A scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result in ArrayIndexOutOfBoundsException

➢Types of Exception in Java

In Java, **exception** is an event that occurs during the execution of a program and disrupts the normal flow of the program's instructions. Bugs or errors that we don't want and restrict our program's normal execution of code are referred to as **exceptions**.

Exceptions can be categorized into two ways: 1 Built-in Exceptions
Checked
Exception Unchecked Exception 2.User-Defined Exceptions



Built-in Exception**. It can be categorized into two broad categories, i.e.,** checked exceptions **and .** unchecked exception

Checked Exception Checked **exceptions are called** compiletime **exceptions because these exceptions are checked at compiletime by the compiler. The compiler ensures whether the programmer handles the exception or not.**

1. The **FileInputStream(File filename)** constructor throws the **FileNotFoundException** that is checked exception.
2. The **read()** method of the **FileInputStream** class throws the **IOException**.
3. The **close()** method also throws the IOException.

   Unchecked Exceptions The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error. Usually, it occurs when the user provides bad data during the interaction with the program.

S.No	Checked Exception	Unchecked Exception

1.	These exceptions are checked at compile time. These exceptions are handled at compile time too.	These exceptions are jus opposite to the checked exceptions. These exceptions are not checked and handled a compile time.
2.	These exceptions are direct subclasses of exception but not extended from RuntimeException class.	They are the direct subclasses of the RuntimeException class.
3.	The code gives a compilation error in the case when a method throws a checked exception. The compiler is not able to handle the exception on its own.	The code compiles without any error because the exceptions escape the notice of the compiler. These exceptions are the results of usercreated errors in programming logic.
4.	These exceptions mostly occur when the probability of failure is too high.	These exceptions occur mostly due to programming mistakes.
5.	Common checked exceptions include IOException, DataAccessException, InterruptedException, etc.	Common unchecked exceptions include ArithmeticException, InvalidClassException, NullPointerException, etc.
6.	These exceptions are propagated using the throws keyword.	These are automatically propagated.
7.	It is required to provide the trycatch and try-finally block to handle the checked exception.	In the case of unchecked exception it is not mandatory.

User-defined ExceptionInJava, we already have some built-in exception classes like ArrayIndexOutOfBoundsException, NullPointerException**, and** ArithmeticException**. These exceptions are restricted to trigger on some predefined conditions. In Java, we can write our own exception class by extends the Exception class. We**

can throw our own exception on a particular condition using the throw keyword. For creating a user-defined exception, we should have basic knowledge of thetry-catch**block and**throw**keyword.**

ArithmeticException, ArrayIndexOutOfBoundExceptions, ClassNotFoundExceptions etc. are come in the category of **Built-in Exception**. Sometimes, the built-in exceptions are not sufficient to explain or describe certain situations. For describing these situations, we have to create our own exceptions by creating an exception class as a subclass of the **Exception** class. These types of exceptions come in the category of **User-Defined Exception**.

Un caught exceptions :

The uncaught exceptions are the exceptions that are not caught by the compiler but automatically caught and handled by the Java built-in exception handler.

Java programming language has a very strong exception handling mechanism. It allow us to handle the exception use the keywords like try, catch, finally, throw, and throws. When an uncaught exception occurs, the JVM calls a special private method known **dispatchUncaughtException( )**, on the Thread class in which the exception occurs and terminates the thread.

The Division by zero exception is one of the example for uncaught exceptions. Look at the following code import java.util.Scanner

public class uncaughtExceptionExample{ publicstaticvoidmain(String[] args){ Scanner read =new Scanner(system.in);System.out.println(“enter the a and b values:”);

int a=readnexInt(); int b=readnextInt();int c=a/b;System.out.println(a+”/”+b+”=”+c);}}

When we execute the above code, it produce the following output for the value a = 10 and b = 0

The keyword try is used to define a block of code that will be tests the occurence of an exception. The keyword catch is used to define a block of code that handles the exception occured in the respective try block.

The uncaught exceptions are the exceptions that are not caught by the compiler but automatically caught and handled by the Java built-in exception handler.

Both try and catch are used as a pair. Every try block must have one or more catch blocks. We can not use try without atleast one catch, and catch alone can be used (catch without try is not allowed).

```
try{…….. code to be tested………}
```

catch(ExceptionType object)

{………. Code for handling the eccption….}

➢Nested try statements

The java allows to write a try statement inside another try statement. A try block within another try block is known as nested try block.

When there are nested try blocks, each try block must have one or more seperate catch blocks.

```
Public class TryCatchExample{ public static void main(String[] args{
 Try{ int list[]=new int[5];list[2]=10;list[4]=2;list[0]=list[2]/list[4];
    Try { list[10]=100;}
     Catch      (ArrayIndexOutOfBoundsException      aie){
publicclassTryCatchExample{
System.out.println("Problem info: ArrayIndexOutOfBoundsException has
occurred");}}
 Catch(Exception      e){System.out.println("Probleminfo:Unknownexceptioon has
occurs");
}
Catch(ArithmeticException ae){System.out.println(""Problem info: Value of
divisor can not be ZERO.")}}}
```

In case of nested try blocks,if an exception occurred in the inner try block and its catch blocks are unable to handle it then transfers the control the outer try's catch block to handle it

➢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;
2. **publicclass** TestThrow1{
3. **staticvoid** validate(**int** age){

```
if(age<18)
thrownew ArithmeticException("not valid");
else
System.out.println("welcome to vote");   }
publicstaticvoid main(String args[]){
validate(13);
System.out.println("rest of the code...");  }}
```

Difference between throw and throws in Java

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Java throw example

```
void m(){
throw new ArithmeticE xception("sorry"); }
```

Java throws example

```
void m() throws Arithmeti cException{  }
```

Java throw and throws example

1.**void** m()**throws** ArithmeticException{ **thrownew** ArithmeticException("sorry"); }

➢Multithreading in Java

Multithreading :Javais a process of executing multiple threads simultaneously.

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

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process. Java Multithreading is mostly used in games, animation, etc.

Advantages of Java Multithreading

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

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

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

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

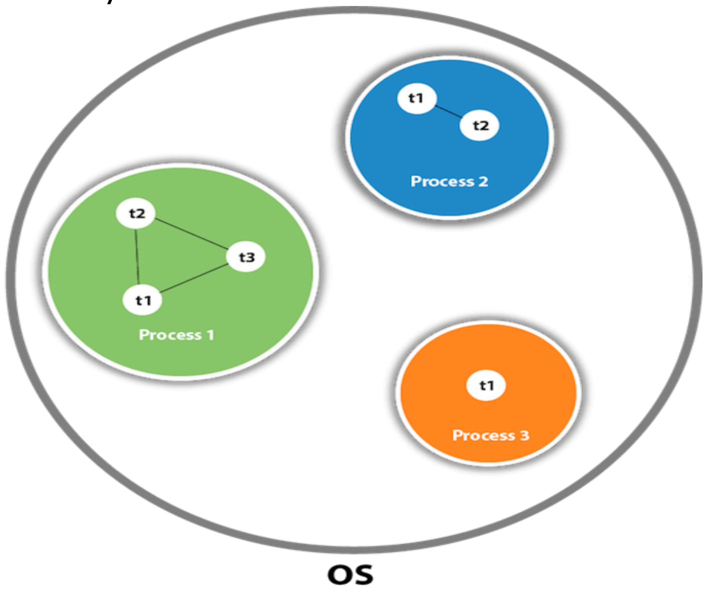
- Process-based Multitasking (Multiprocessing)
- Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing) ○Each process has an address in memory. In other words, each process allocates a separate memory area.
   - A process is heavyweight. ○Cost of communication between the process is high.
   - Switching from one process to another requires some time for saving and loadingregisters, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading) ○Threads share the same address space,A thread is lightweight. ,Cost of communication between the thread is low.

➢Thread in java

A thread is a lightweight subprocess, the 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 uses a shared memory area.



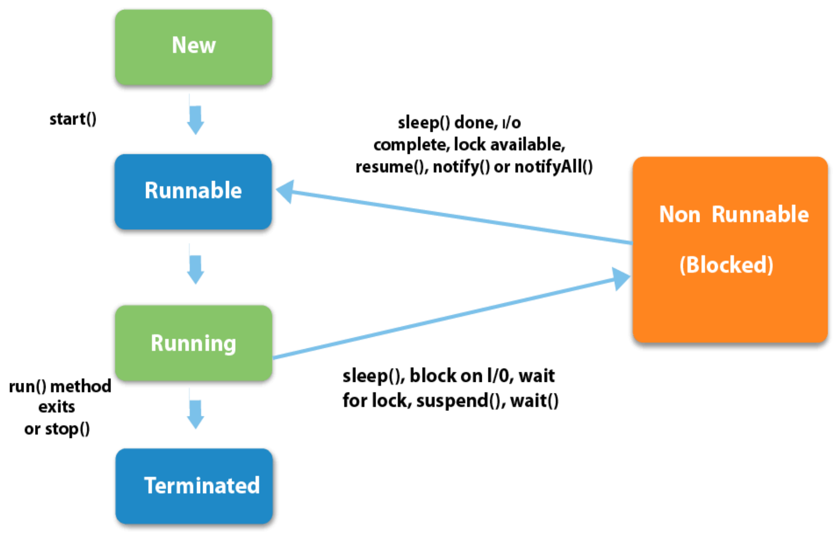
As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside theOS, and one process can have multiple threads.

Life cycle of a Thread (Thread States)

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.

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


New
start()
Runnable
sleep() done, I/o complete, lock available, resume(), notify() or notifyAll()
Non Runnable (Blocked)
Running
sleep(), block on I/0, wait for lock, suspend(), wait()
run() method exits or stop()
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.

➢Multithreading in Java

MULTITHREADING in Java is a process of executing two or more threads simultaneously to maximum utilization of CPU. Multithreaded applications execute two or more threads run concurrently. Hence, it is also known as Concurrency in Java. Each thread runs parallel to each other. Mulitple threads don't allocate separate memory area, hence they save memory. Also, context switching between threads takes less time.

Advantages of multithread:

- The users are not blocked because threads are independent, and we can perform multiple operations at times

- As such the threads are independent, the other threads won't get affected if one thread meets an exception

➢**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.

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.

➢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

Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

```
class Table{
void printTable(int n){//method not synchronized
for(int i=1;i<=5;i++){
System.out.println(n*i);        try{
Thread.sleep(400);
    }catch(Exception e){System.out.println(e);}   }   }  }
class MyThread1 extends Thread{    Table
t;   MyThread1(Table t){
this.t=t;  }
publicvoid run(){
t.printTable(5);  }  }
class MyThread2 extends Thread{    Table t;
MyThread2(Table    t){        this.t=t;        }
publicvoid run(){
t.printTable(100);  }  }    class
TestSynchronization1{
publicstaticvoid main(String args[]){    Table
obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
.       t1.start();   t2.start(); } }
Output: 5 100 10 200 15 300  20 400 25  500
```

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

```
class Table{
synchronizedvoid printTable(int n){//synchronized
methodfor(int i=1;i<=5;i++){
System.out.println(n*i);       try{   Thread.sleep(400);
    }catch(Exception e){System.out.println(e);}   }  }  }   class
MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;  }  publicvoid run(){   t.printTable(5);  }
class MyThread2 extends Thread{
Table t;
MyThread2(Table    t){       this.t=t;       }
publicvoid run(){  t.printTable(100);  }  }
publicclass              TestSynchronization2{
publicstaticvoid main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();  t2.start();  }  }
```

Synchronized Block in Java

Synchronized block can be used to perform synchronization on any specific resource of the method. Points to remember for Synchronized block

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.

Syntax to use synchronized block

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

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. 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. 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()

wait()	sleep()
wait() method releases the lock	sleep() method doesn't release the lock.
is the method of Object class	is the method of Thread class
is the non-static method	is the static method
is the non-static method	is the static method
should be notified by after the specified amount of notify() or notifyAll() time, sleep is completed. methods	

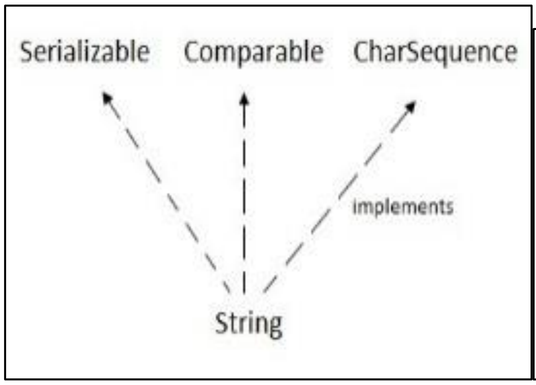
➢String Handling

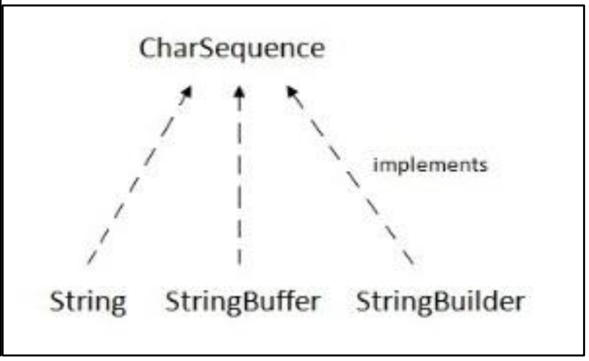
InJava, string is basically an object that represents sequence of char values. Anarrayof characters works same as Java string. For example:

char[] ch={'j','a','v','a','t','p','o','i','n','t'}; String s=new String(ch); is same as:
String s="javatpoint";

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

The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces





CharSequence Interface

The CharSequence interface is used to represent the sequence of characters. String,StringBufferandStringBuilderclasses implement it. It means, we can create strings in java by using these three classes.

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 a string object.

How to create a string object? There are two ways to create String object: By string literal ,By new keyword

1) String Literal Java String literal is created by using double quotes. For Example: 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 the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

String s1="Welcome"; String s2="Welcome";//It doesn't create a new inst ance

2) By new keyword

String s=new String("Welcome");//creates two objects and one reference v ariable

In such case,JVMwill create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool.

Java String class methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values. Char charAt(int,index),intlength(),static String format(String format, Object... args),static String format(String format, Object... args),String substring(int beginIndex),String substring(int beginIndex, int endIndex),static String valueOf(int value),boolean contains(CharSequence s),static String join(CharSequence delimiter, CharSequence... elements),String concat(String str),boolean isEmpty(),String[] split(String regex),String toUpperCase()String toLowerCase()

Immutable String in Java

In java, string objects are immutable. Immutable simply means unmodifiable or unchangeable. Once string object is created its data or state can't be changed but a new string object is created.

Why string objects are immutable in java?

Because java uses the concept of string literal.Suppose there are 5 reference variables,all referes to one object "sachin".If one reference variable changes the value of the object, it will be affected to all the reference variables.

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

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.

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.

2) String compare by == operator

The = = operator compares references not values.

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

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:

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.

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

Java String class methods The java.lang.String class provides a lot of methods to work on string. By the help of these methods, we can perform operations on string such as trimming, concatenating, converting, comparing, replacing strings etc.

Java String is a powerful concept because everything is treated as a string if you submit any form in window based, web based or mobile application.

Let's see the important methods of String class.

Java String toUpperCase() and toLowerCase() method

The java string toUpperCase() method converts this string into uppercase letter and string toLowerCase() method into lowercase letter.

```
String s="Sachin";
System.out.println(s.toUpperCase());//SACHIN
System.out.println(s.toLowerCase());//sachin  System.out.println(s);
```

Java String startsWith() and endsWith() method

Java String charAt() method The string charAt() method returns a character at specified index.

Java String charAt() method The string charAt() method returns a character at specified index.

Java String intern() method

A pool of strings, initially empty, is maintained privately by the class String.

When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned.
Java String valueOf() method The string valueOf() method coverts given type such as int, long, float, double, boolean, char and char array into string.

Java String replace() method The string replace() method replaces all occurrence of first sequence of character with second sequence of character.

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

Important Constructors of StringBuffer class:

StringBuffer(), StringBuffer(String str),StringBuffer(int capacity)

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

1) StringBuffer append() method The append() method concatenates the given argument with this string.

2) StringBuffer insert() method The insert() method inserts the given string with this string at the given position.

3)StringBuffer replace() method The replace() method replaces the given string from the specified beginIndex and endIndex.

4)StringBuffer delete() method The delete() method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

5) StringBuffer reverse() method The reverse() method of StringBuilder class reverses the current string

6) StringBuffer capacity() method

The capacity() method of StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

7) StringBuffer ensureCapacity() method

The ensureCapacity() method of StringBuffer class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

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

Constructors of StringTokenizerclass *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 set.

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.

Short Questions

1.what is access protection?
2.what are the importing packages?

3.What are the exception types?
4.what is thread model?

Long Questions

5.explain about packages and interfaces?
6.Briefly explain about exception handling?
7.Explain about multithreading ?
8.Inter thread communication, string handling ?

UNIT-III

➢Wrapper classes in Java

The **wrapper class in Java** provides the mechanism to convert primitive into object and objectinto primitive.

Since J2SE 5.0, **autoboxing** and **unboxing** feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Use of Wrapper classes in Java Let us see the different scenarios, where we need to use the wrapper classes.

- **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- **Synchronization:** Java synchronization works with objects in Multithreading. ○**java.util package:** The java.util package provides the utility classes to deal with objects.
- **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

The list of eight wrapper classes are given below:

Primitive	**Wrapper**

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

Autoboxing The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Unboxing The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing.

Java Number Class JavaNumberclassis an **abstract** class which is placed in **java.lang** package. It has **four** abstract methods and two **concrete** methods. The abstract class Number is the superclass of classes BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, and Short. This class contains a single consructor **number()**.

➢Number Class

Java Number class provides methods to convert the represented numeric value to byte, double, float, int, long, and short type. The various Java Number methods are as follows-

SN	Modifier & Type	Method	Description
1)	Byte	byteValue()	It converts the given number into a byte type and returns the value of the specified number as a byte.
2)	abstract double	doubleValue()	It returns the value of the specified number as a double equivalent.

3)	abstract float	floatValue()	It returns the float equivalent value of the specified Number object.
4)	abstract int	intValue()	It returns the value of the specified number as an int.
5)	abstract long	longValue()	It returns the value of the specified number object as long equivalent.
6)	short	shortValue()	It returns the value of the specified number as a short type after a primitive conversion.

➢Java Character class

The Character class generally wraps the value of all the primitive type char into an object. Any object of the type Character may contain a single field whose type is char. All the fields, methods, and constructors of the class Character are specified by the Unicode Data file which is particularly a part of Unicode Character Database and is maintained by the Unicode Consortium.

Methods: charCount(int codePoint),charValue(),codePointAt(char[]a, int index),codePointAt(char[]a, int index, int limit),codePointAt(CharSequence seq, int index),codePointBefore(char[]a, int index), codePointBefore(char[]a, int index, int start)

➢Java Boolean class

The Boolean class wraps a value of the primitive type boolean in an object. Its object contains only a single field whose type is boolean.

Methods:booleanValue(),compare(),compareTo(),equals(),getBoolean(),hashCode(),logicalAnd(),logicalOr(),logicalXor(),parseBoolean(),toString(), valueOf()

➢More Utility Classes:

Java.utilpackage contains the collection's framework, legacy collection classes, event model, date and time facilities,internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array). Here is the list of most commonly used top ten Java utility classes:

1. Java Arrays Class Java.util package provides an**Arrays class**that contains a static factory that allows arrays to be viewed as lists. The class contains various methods for manipulating arrays like sorting and searching. The methods in this class throw a NullPointerException, if the specified array reference is null.

2. Java Vector Class Java.util. A vector is a sequence container implementing array that can change in size. Unlike an array, the size of a vector changes automatically when elements are appended or deleted. It is similar to ArrayList, but with two differences:1 Vector package provides a**vector class**that models and implements vector data structure is synchronized. Vector contains many legacy methods that are not part of the collections framework.

3. Java LinkedList Class Java.util package provides a**LinkedList class**that has Doubly-linked list implementation of the List and Deque interfaces. The class has all optional list operations, and permits all elements (including null). Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

4. Java Calendar Class Java.util package provides a**Calendar class**that represents a specific instant in time, with millisecond precision.

5. Java Collections Class

Java.util package provides a**Collections class**which consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection. The methods of this class all throw a NullPointerException if the collections or class objects provided to them are null.

6. Java HashMap Class

A Hashtable models and implements the Map interface. This implementation provides all the optional map operations and permits null values and the null key. A**HashMap class**is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls. This class makes no guarantees as to the order of the map. It does not guarantee that the order will remain constant over time.

7. Java Random Class

Java.util package provides a**Random class**. An instance of this class is used to generate a stream of pseudorandom numbers. The class uses a 48-bit seed, which is modified using a linear congruential formula. The algorithms implemented by class Random use a protected utility method that on each invocation can supply up to 32 pseudorandomly generated bits.

8. Java UUID Class Java.util package provides a**UUID class**that represents animmutable universally unique identifier (UUID). A UUID represents a 128-bit value. It is used for creating random file names, session id in web application, transaction id etc. There are four different basic types of UUIDs: time-based, DCE security, name-based, and randomly generated UUIDs.

9. Java Scanner Class

Java.util package provides a**Scanner class**. A simple text scanner parse primitive types and strings using regular expressions. A Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various next methods.

10. Java ArrayList Class

Java.util package provides an**ArrayList class**that provides resizable-array implementation of the list interface. It implements all optional list operations and permits all elements including null. ➢**Java.util.Vector Class**

The **java.util.Vector** class implements a growable array of objects. Similar to an Array, it contains components that can be accessed using an integer index. Following are the important points about Vector −

- The size of a Vector can grow or shrink as needed to accommodate adding and removing items.
- Each vector tries to optimize storage management by maintaining a capacity and a capacity Increment.
- As of the Java 2 platform v1.2, this class was retrofitted to implement the List interface.
- Unlike the new collection implementations, Vector is synchronized.
- This class is a member of the Java Collections Framework.

Class declaration

Following is the declaration for **java.util.Vector** class − public class

Vector<E> extends AbstractList<E> implements List<E>,

RandomAccess, Cloneable, Serializable
Here <E> represents an Element, which could be any class. For example, if you're building an array list of Integers then you'd initialize it as follows −
ArrayList<Integer> list =newArrayList<Integer>();

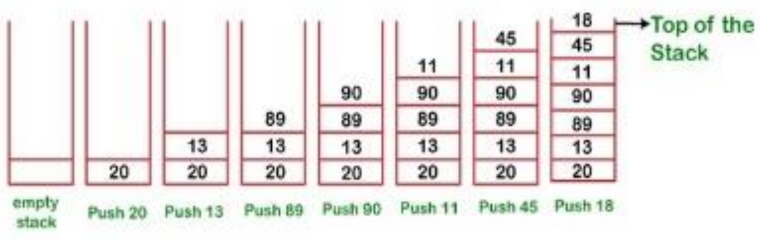
Class constructors

Sr.No.	**Constructor & Description**
1	**Vector()** This constructor is used to create an empty vector so that its internal data array has size 10 and its standard capacity increment is zero.
2	**Vector(Collection<? extends E> c)** This constructor is used to create a vector containing the elements of the specified collection, in the order they are returned by the collection's iterator.
3	**Vector(int initialCapacity)** This constructor is used to create an empty vector with the specified initial capacity and with its capacity increment equal to zero.
4	**Vector(int initialCapacity, int capacityIncrement)** This constructor is used to create an empty vector

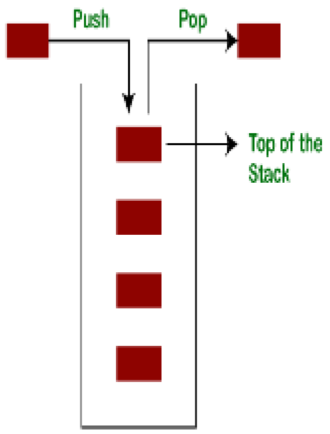
	with the specified initial capacity and capacity increment.

➢Stack

The **stack** is a linear data structure that is used to store the collection of objects. It is based on **Last-In-First-Out** (LIFO).Java collectionframework provides many interfaces and classes to store the collection of objects. One of them is the **Stack class** that provides different operations such as push, pop, search, etc.

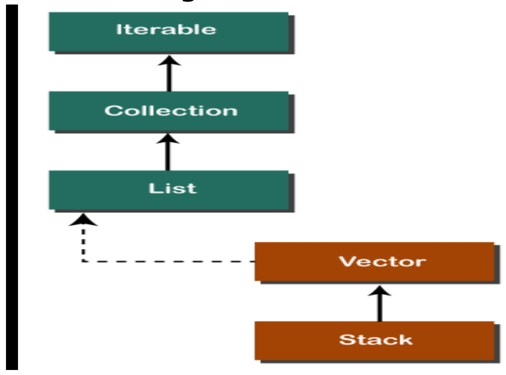


Push operation



Java Stack Class

In Java, **Stack** is a class that falls under the Collection framework that extends the **Vector** class. It also implements interfaces **List, Collection, Iterable, Cloneable, Serializable.** It represents the LIFO stack of objects. Before using the Stack class, we must import the java.util package. The stack class arranged in the Collections framework hierarchy, as shown below.



Methods of the Stack Class **We can perform push, pop, peek and search operation on the stack. The Java Stack class provides mainly five methods to. Along with this, it also provides all the methods of theJava Vector class.**

Method	**Modifier and Type**	perform these operations **Description**
empty()	boolean	The method checks the stack is empty or not.
push(E item)	E	The method pushes (insert) an element onto
		the top of the stack.
pop()	E	The method removes an element from the top of the stack and returns the same element as the value of that function.
peek()	E	The method looks at the top element of the stack without removing it.
search(Object o)	int	The method searches the specified object and returns the position of the object.

➢Java Hashtable class

Java Hashtable class implements a hashtable, which maps keys to values. It inherits Dictionary class and implements the Map interface.

Points to remember ○A Hashtable is an array of a list. Each list is known as a bucket. The position of the bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key. ○Java Hashtable class contains unique elements. ○Java Hashtable class doesn't allow null key or value.

- ○ Java Hashtable class is synchronized.
- ○ The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.

Hashtable class declaration

```
1.publicclass Hashtable<K,V>extends Dictionary<K,V>implements Map
```

<K,V>, Cloneable, Serializable

Hashtable class Parameters **Let's see the Parameters for java.util.Hashtable class.**

- **K**: It is the type of keys maintained by this map,**V**: It is the type of mapped values.

Constructors of Java Hashtable class

Constructor	**Description**
Hashtable()	It creates an empty hashtable having the initial default capacity and load factor.
Hashtable(int capacity)	It accepts an integer parameter and creates a hash table that contains a specified initial capacity.
Hashtable(int capacity, float loadFactor)	It is used to create a hash table having the specified initial capacity and loadFactor.

Methods of Java Hashtable class

Method	**Description**
void clear()	It is used to reset the hash table.
Object clone()	It returns a shallow copy of the Hashtable.
V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a mapping for the specified key and its current mapped value
V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)	It is used to compute its value using the given mapping function, if the specified key is not already associated with a value and enters it into this map unless null.

V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null.

➢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.***

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: boolean hasMoreTokens(),String nextToken(),String nextToken(String delim), boolean hasMoreElements(),Object nextElement(),int countTokens()

➢Calendar Class in Java with examples

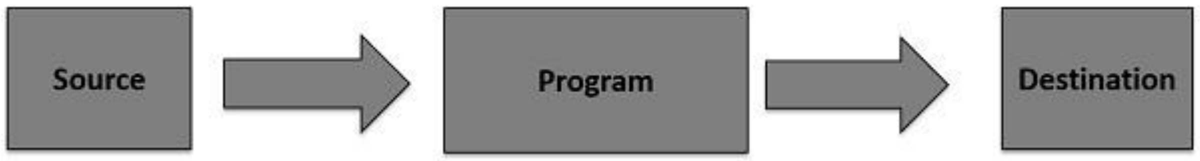
Calendar class in Java is an abstract class that provides methods for converting date between a specific instant in time and a set of calendar fields such as MONTH, YEAR, HOUR, etc. It inherits Object class and implements the Comparable, Serializable, Cloneable interfaces.

As it is an Abstract class, so we cannot use a constructor to create an instance. Instead, we will have to use the static method Calendar.getInstance() to instantiate and implement a sub-class.

Calendar.getInstance(): return a Calendar instance based on the current time in the default time zone with the default locale.
Calendar.getInstance(TimeZone zone) Calendar.getInstance(Locale aLocale) Calendar.getInstance(TimeZone zone, Locale aLocale)

- **Input/output**
- **Stream**

A stream can be defined as a sequence of data. There are two kinds of Streams –



InPutStream – The InputStream is used to read data from a source.

- **OutPutStream** – The OutputStream is used for writing data to a destination.
- Java provides strong but flexible support for I/O related to files and networks but this tutorial covers very basic functionality related to streams and I/O. We will see the most commonly used examples one by one – ➢**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 –
Example

```
import java.io.*;
publicclassCopyFile{

publicstaticvoid main(String args[])throwsIOException{
FileInputStreamin=null;
FileOutputStreamout=null;try{in=newFileInputStream("input.txt");
out=newFileOutputStream("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.

➢Standard INPUT/OUTPUT Streams

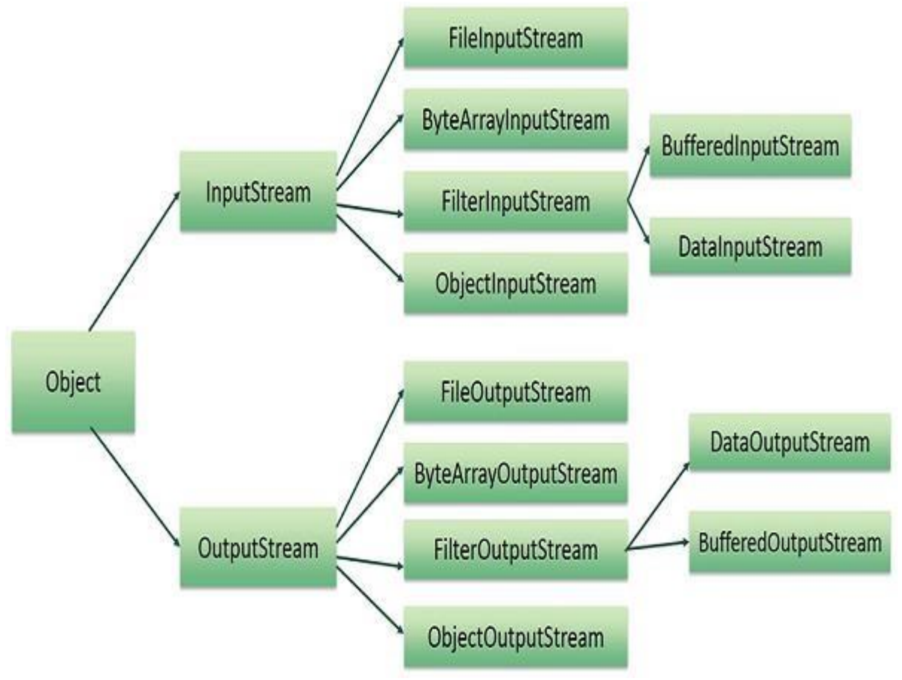
All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen. If you are aware of C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR. Similarly, Java provides the following three standard streams −

- **Standard Input** − This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.
- **Standard Output** − This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as **System.out**.
- **Standard Error** − This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as **System.err**

Reading and Writing Files

As described earlier, a stream can be defined as a sequence of data. The **InputStream** is used to read data from a source and the **OutputStream** is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.



FileInputStream

This stream is used for reading data from the files. Objects can be created using the keyword **new** and there are several types of constructors available.

FileOutputStream

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output. Here are two constructors which can be used to create a FileOutputStream object.

➢ Java - File Class

Java File class represents the files and directory pathnames in an abstract manner. This class is used for creation of files and directories, file searching, file deletion, etc.
The File object represents the actual file/directory on the disk. Following is the list of constructors to create a File object.

Sr.No.	**Method & Description**

1	**File(File parent, String child)** This constructor creates a new File instance from a parent abstract pathname and a child pathname string.
2	**File(String pathname)** This constructor creates a new File instance by converting the given pathname string into an abstract pathname.
3	**File(String parent, String child)** This constructor creates a new File instance from a parent pathname string and a child pathname string.
4	**File(URI uri)** This constructor creates a new File instance by converting the given file: URI into an abstract pathnam

Sr.No.	**Method & Description**
1	**public String getName()** Returns the name of the file or directory denoted by this abstract pathname.
2	**public String getParent()** Returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
3	**public File getParentFile()** Returns the abstract pathname of this abstract pathname's parent, or null if this pathname does not name a parent directory.
4	**public String getPath()** Converts this abstract pathname into a pathname string.

5	**public boolean isAbsolute()** Tests whether this abstract pathname is absolute. Returns true if this abstract pathname is absolute, false otherwise.
6	**public String getAbsolutePath()** Returns the absolute pathname string of this abstract pathname.
7	**public boolean canRead()** Tests whether the application can read the file denoted by this abstract pathname. Returns true if and only if the file specified by this abstract pathname exists and can be read by the application; false otherwise.

- **Programming Graphical User Interface (GUI)**

1. Introduction

There are current three sets of Java APIs for graphics programming: AWT (Abstract Windowing Toolkit), Swing and JavaFX.

1. AWT API was introduced in JDK 1.0. Most of the AWT components have become obsolete and should be replaced by newer Swing components.
2. Swing API, a much more comprehensive set of graphics libraries that enhances the AWT, was introduced as part of Java Foundation Classes (JFC) after the release of JDK 1.1. JFC consists of Swing, Java2D, Accessibility, Internationalization, and Pluggable Look-and-Feel Support APIs. JFC has been integrated into core Java since JDK 1.2.
3. The latest JavaFX, which was integrated into JDK 8, is meant to replace Swing.

Other than AWT/Swing/JavaFX graphics APIs provided in JDK, other organizations/vendors have also provided graphics APIs that work with Java, such as Eclipse's Standard Widget Toolkit (SWT) (used in Eclipse), Google Web Toolkit (GWT) (used in Android), 3D Graphics API such as Java bindings for OpenGL (JOGL) and Java3D.

Programming GUI with AWT

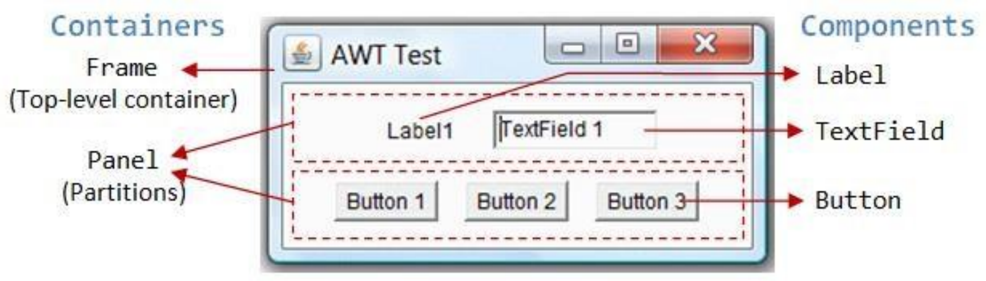
AWT Packages

It consists of 12 packages of 370 classes (Swing is even bigger, with 18 packages of 737 classes as of JDK 8). Fortunately, only 2 packages java.awt and java.awt.event - are commonly-used.

1. The java.awt package contains the *core* AWT graphics classes: ○GUI Component classes, such as Button, TextField, and Label. ○GUI Container classes, such as Frame and Panel.
   - Layout managers, such as FlowLayout, BorderLayout and GridLayout.
   - Custom graphics classes, such as Graphics, Color and Font.
2. The java.awt.event package supports event handling:
   - Event classes, such as ActionEvent, MouseEvent, KeyEvent and WindowEvent,
   - Event Listener Interfaces, such as ActionListener, MouseListener, MouseMotionListener, KeyListener and WindowListener,
   - Event Listener Adapter classes, such as MouseAdapter, KeyAdapter, and WindowAdapter.

AWT provides *a* platform-independent *and* device-independentinterfaceto develop graphic programs that runs on all platforms, including Windows, Mac OS X, and Unixes.

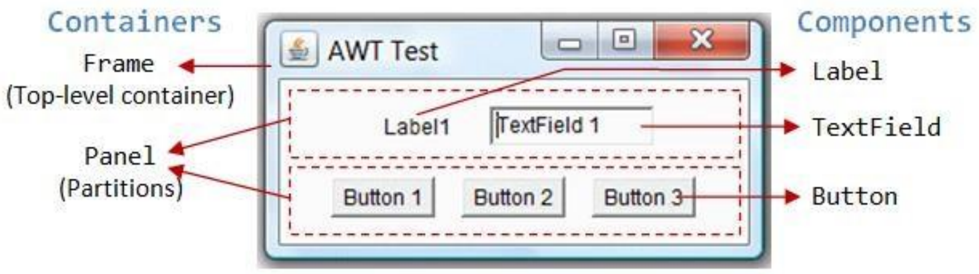
Containers and Components



There are two types of GUI elements:

1. *Component*: Components are elementary GUI entities, such as Button, Label, and TextField.
2. *Container*: Containers, such as Frame and Panel, are used to *hold components in a specific layout* (such as FlowLayout or GridLayout). A container can also hold sub-containers

In the above figure, there are three containers: a Frame and two Panels. A Frame is the *top-level container* of an AWT program. A Frame has a title bar (containing an icon, a title, and the minimize/maximize/close buttons), an optional menu bar and the content display area. A Panel is a *rectangular area* used to group related GUI components in a certain layout. In the above figure, the top-level Frame contains two Panels. There are five components: a Label (providing description), a TextField (for users to enter text), and three Buttons (for user to trigger certain programmed actions).

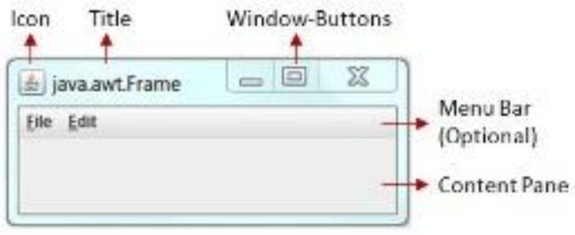


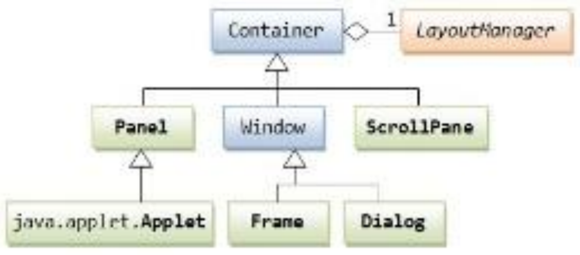
AWT *Container Classes*

Top-Level Containers: Frame, Dialog and Applet

Each GUI program has a *top-level container*. The commonly-used top-level containers in AWT are Frame, Dialog and Applet:

A Frame provides the "main window" for your GUI application. It has a title bar (containing an icon, a title, the minimize, maximize/restore-down and close buttons), an optional menu bar, and the content display area. To write a GUI program, we typically start with a subclass extending from java.awt.Frame to inherit the main window as follows:





- An AWT Dialog is a *"pop-up window*" used for interacting with the users. A Dialog has a title-bar (containing an icon, a title and a close button) and a content display area, as illustrated.

- An AWT Applet (in package java.applet) is the top-level container for an applet, which is a Java program running inside a browser.

Secondary Containers: Panel and ScrollPane

Secondary containers are placed inside a top-level container or another secondary container. AWT provides these secondary containers:

Panel: a rectangular box used to *layout* a set of related GUI components in pattern such as grid or flow. ScrollPane: provides automatic horizontal and/or vertical scrolling for a single child component.

Hierarchy of the AWT Container Classes

The hierarchy of the AWT Container classes is as follows:.

AWT Component Classes

AWT provides many ready-made and reusable GUI components in package java.awt. The frequently-used
are: Button, TextField, Label, Checkbox, CheckboxGroup (radio buttons),
List, and Choice, as illustrated below.

AWT GUI Component: java.awt.Label

A java.awt.Label provides a descriptive text string. Take note that System.out.println() prints to the system console, NOT to the graphics screen. You could use a Label to label another component (such as text field) to provide a text description.
Check the JDK API specification for java.awt.Label.

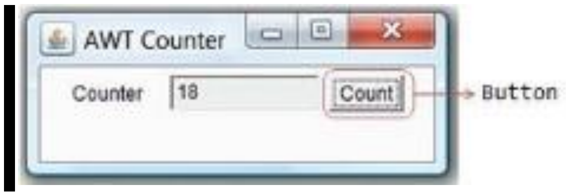
Constructing a Component and Adding the Component into a Container

Three steps are necessary to create and place a GUI component:

1. Declare the component with an *identifier* (*name*);
2. Construct the component by invoking an appropriate constructor via the new operator;
3. Identify the container (such as Frame or Panel) designed to hold this component. The container can then add this component onto itself via *aContainer*.add(*aComponent*) method. Every container has a add(Component) method. Take note that it is the container that actively and explicitly adds a component onto itself, NOT the other way.

AWT GUI Component: java.awt.Button

A java.awt.Button is a GUI component that triggers a certain



prog a upon

4 **Constr**

public *b*);

/// Construct a Button with empty label

The Button class has two constructors. The first constructor creates a Button object with the given label painted over the button. The second constructor creates a Button object with no label. **Public Methods**

```
public String getLabel();
   // Get the label of this Button instance
public void setLabel(String btnLabel);
   // Set the label of this Button instance
public void setEnable(boolean enable);
   // Enable or disable this Button. Disabled Button cannot be clicked.
/ Construct a Button with the given label
public Button();
```

```
import java.awt.*;
import java.awt.event.*;
public class AWTCounter extends Frame implements ActionListener {
   private Label lblCount;
   private TextField tfCount;
   private Button btnCount;
   private int count = 0;
   public AWTCounter () {
      setLayout(new FlowLayout());
      lblCount = new Label("Counter"); add(lblCount);
      tfCount = new TextField(count + "", 10);
      tfCount.setEditable(false);
      add(tfCount);
       btnCount = new Button("Count");
      add(btnCount);
      btnCount.addActionListener(this);
      setTitle("AWT Counter"); setSize(250, 100);
    setVisible(true);
    public static void main(String[] args) {
            AWTCounter app = new AWTCounter();}
  @Override
   public void actionPerformed(ActionEvent evt) {
      ++count;
      tfCount.setText(count + "");  }
}
```

AWT Event-Handling

Java adopts the so-called "Event-Driven" (or "Event-Delegation") programming model for event-handling, similar to most of the visual programming languages like Visual Basic.

In event-driven programming, a piece of event-handling codes is executed (or *called back* by the graphics subsystem) when an event was fired in response to an user input (such as clicking a mouse button or hitting the ENTER key in a text field).

Callback Methods

In some languages, you can directly attach a method (or function) to an event (such as mouse-click). For example, the following JavaScript code (called JSCounter.html) implement a counter similar to the AWTCounter, with a text label, text field and button.

Swing Introduction

Swing is part of the so-called "Java Foundation Classes (JFC)" (have you heard of MFC?), which was introduced in 1997 after the release of JDK 1.1. JFC was subsequently included as an integral part of JDK since JDK 1.2. JFC consists of:

- Swing API: for advanced graphical programming.
- Accessibility API: provides assistive technology for the disabled.
- Java 2D API: for high quality 2D graphics and images.
- Pluggable look and feel supports.
- Drag-and-drop support between Java and native applications.

The goal of Java GUI programming is to allow the programmer to build GUI that looks good on ALL platforms. JDK 1.0's AWT was awkward and nonobject-oriented (using many event.getSource()). JDK 1.1's AWT introduced event-delegation (event-driven) model, much clearer and object-oriented. JDK 1.1 also introduced inner class and JavaBeans – a component programming model for visual programming environment (similar to Visual Basic).

Swing appeared after JDK 1.1. It was introduced into JDK 1.1 as part of an add-on JFC (Java Foundation Classes). Swing is a rich set of easy-to-use, easy-to-understand JavaBean GUI components that can be dragged and dropped as "GUI builders" in visual programming environment. Swing is now an integral part of Java since JDK 1.2.

Swing's Features

Swing is huge (consists of 18 packages of 737 classes as in JDK 1.8) and has great depth. Swing provides a huge and comprehensive collection of reusable GUI components. The main features of Swing are (extracted from the Swing website):

1. Swing is written in pure Java (except a few classes) and therefore is 100% portable.
2. Swing components are lightweight. The AWT components are *heavyweight* (in terms of system resource utilization). Each AWT component has its own opaque native display, and always displays on top of the lightweight components. AWT components rely heavily on the underlying windowing subsystem of the native operating system. For Swing components support *pluggable look-and-feel*. You can choose between *Java look-and-feel* and the *look-and-feel of the underlying OS* Similarly, a Swing button runs on the UNIX looks like a UNIX's button and feels like a UNIX's button.
3. Swing supports *mouse-less operation*, i.e., it can operate entirely using keyboard.
4. Swing components support "tool-tips".
5. Swing components are *JavaBeans* – a Component-based Model used in Visual Programming (like Visual Basic). You can drag-and-drop a Swing component into a "design form" using a "GUI builder" and double-click to attach an event handler.
6. Swing application uses AWT event-handling classes (in package java.awt.event). Swing added some new classes in package javax.swing.event, but they are not frequently used.
7. Swing application uses AWT's layout manager (such as FlowLayout and BorderLayout in package java.awt). It added new layout managers, such as Springs, Struts, and BoxLayout (in package javax.swing).
8. Swing implements *double-buffering* and automatic repaint batching for smoother screen repaint.
9. Swing introduces JLayeredPane and JInternalFrame for creating Multiple Document Interface (MDI) applications.
10. Swing supports floating toolbars (in JToolBar), splitter control, "undo".

Using Swing API

Swing's Components

Compared with the AWT component classes (in package java.awt), Swing component classes (in package javax.swing) begin with a prefix "J", e.g., JButton, JTextField, JLabel, JPanel, JFrame, or JApplet.

Writing Swing Applications

In summary, to write a Swing application, you have:

1. Use the Swing components with prefix "J" in package javax.swing, e. g., JFrame, JButton, JTextField, JLabel, etc.
2. A top-level container (typically JFrame) is needed. The JComponents should not be added directly onto the top-level container. They shall be added onto the *content-pane* of the top-level container. You can retrieve a reference to the content-pane by invoking method getContentPane() from the top-level container.
3. Swing applications uses AWT event-handling classes, e. g., ActionEvent/ActionListener, MouseEvent/MouseListener, etc.

4. Run the constructor in the Event Dispatcher Thread (instead of Main thread) for thread safety, as shown in the following program template.

Short Questions

1.what are the float and byte?

2.what are the two differences between byte and byte stream?

3.what are the collection classes ?

4.what are the character streams?

Long Questions

5. Explain the Java utilities?

6. Briefly explain the type wrappers?

7. Explain the collections in Java?

8.what are the more **utility** classes? Explain?

UNIT-4

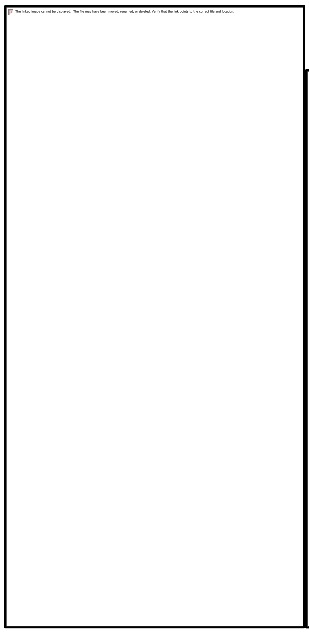
➢Java Applet

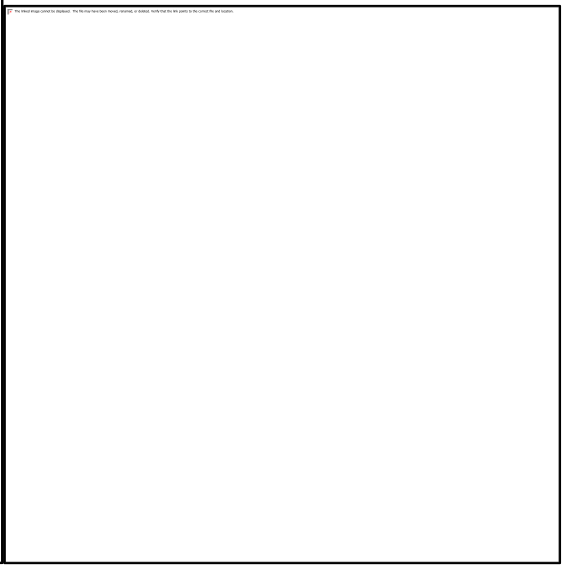
Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

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.





Lifecycle of Java Applet

1.Applet is initialized. 2 Applet is started. 3 Applet is painted. 4Applet is stopped.5 Applet is destroyed.

Lifecycle methods for Applet:

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

java.applet.Applet class

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

1. **public void init():** is used to initialized the Applet. It is invoked only once.
2. **public void start():** is invoked after the init() method or browser is maximized. It is used to start the Applet.
3. **public void stop():** is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.
4. **public void destroy():** is used to destroy the Applet. It is invoked only once.

java.applet Component class

The Component class provides 1 life cycle method of applet.

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

Who is responsible to manage the life cycle of an applet?

Java Plug-in software.

How to run an Applet? **There are two ways to run an applet**

1.By html file.2. By appletViewer tool(for testing pppurpose).

Simple example of Applet by html file:

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

```
//First.java
import java.applet.Applet;   import java.awt.Graphics;
publicclass First extends Applet{
publicvoid paint(Graphics g){
g.drawString("welcome",150,150);  } myapplet.html
```

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

Simple example of Applet by appletviewer tool:

To execute the applet by appletviewer tool, create an applet that contains applet tag in comment and compile it. After that run it by: appletviewer First.java. Now Html file is not required but it is for testing purpose only.

```
//First.java
import java.applet.Applet;
import java.awt.Graphics;
publicclass First extends Applet{
publicvoid paint(Graphics g){
g.drawString("welcome to applet",150,150); }  }
/*<applet code="First.class" width="300" height="300"></applet>*/
```

To execute the applet by appletviewer tool, write in command prompt:
c:\>javac First.java **c:\>**appletviewer First.java

➢Displaying Image in Applet

Applet is mostly used in games and animation. For this purpose image is required to be displayed. The java.awt.Graphics class provide a method drawImage() to display the image.

Syntax of drawImage() method:

1.**public abstract boolean drawImage(Image img, int x, int y**
ImageObserver observer): is used draw the specified image.

How to get the object of Image:

The java.applet.Applet class provides getImage() method that returns the object o
Image. Syntax:

```
public Image getImage(URL u, String image){}
```

Other required methods of Applet class to display image:

1. **public URL getDocumentBase():** is used to return the URL of the document in which applet is embedded.
2. **public URL getCodeBase():** is used to return the base URL.

Example of displaying image in applet:

```
import java.awt.*;  import java.applet.*;
publicclass DisplayImage extends Applet {
Image picture;    publicvoid init() {
picture = getImage(getDocumentBase(),"sonoo.jpg"); }
publicvoid paint(Graphics g) {
g.drawImage(picture, 30,30, this);  }   }
```

myapplet.html

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

➢Parameter in Applet

We can get any information from the HTML file as a parameter. For this purpose, Applet class provides a method named getParameter(). Syntax:

➢**public** String getParameter(String parameterName)

Example of using parameter in Applet:

```
import java.applet.Applet;   import java.awt.Graphics;
publicclass UseParam extends Applet{
publicvoid paint(Graphics g){
String str=getParameter("msg");
g.drawString(str,50, 50);  }  }
```

myapplet.html

```
<html><body>
<applet code="UseParam.class" width="300" height="300">
<param name="msg" value="Welcome to applet">
```

```
</applet></body></html>
```

Animation in Applet

Applet is mostly used in games and animation. For this purpose image is required to be moved.

Example of animation in applet:

```
import java.awt.*;   import java.applet.*;
publicclass AnimationExample extends Applet {
Image picture;    publicvoid init() {
picture =getImage(getDocumentBase(),"bike_1.gif");  }
publicvoid paint(Graphics g) {  for(int i=0;i<500;i++){
g.drawImage(picture, i,30, this);
try{Thread.sleep(100);}catch(Exception e){}  }   }  }
```

myapplet.html

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

➢Event and Listener (Java Event Handling)

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

Java Event classes and Listener interfaces

Steps to perform Event Handling

Following steps are required to perform event handling:

1.Register the component with the Listener

Event Classes	**Listener Interfaces**
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

Registration Methods

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

- **Button**
  - public void addActionListener(ActionListener a){}
- **MenuItem**
  - public void addActionListener(ActionListener a){}
- **TextField**

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

- **TextArea**
  - public void addTextListener(TextListener a){}
- **Checkbox**
  - public void addItemListener(ItemListener a){}
- **Choice**
  - public void addItemListener(ItemListener a){}
- **List**
  - public void addActionListener(ActionListener a){}
  - public void addItemListener(ItemListener a){}

➢Java Event Handling Code

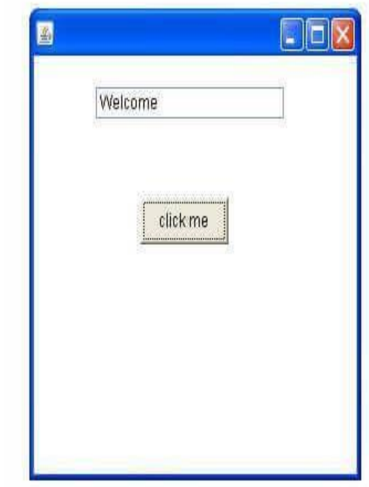
We can put the event handling code into one of the following places:

1.Within class2 Other class 3 Anonymous class

Java event handling by implementing ActionListener

```
import java.awt.*;  import java.awt.event.*;
class AEvent extends Frame implements ActionListener{
TextField tf;   AEvent(){
//create components  tf=new TextField();
tf.setBounds(60,50,170,20);
Button b=new Button("click me");
b.setBounds(100,120,80,30);
b.addActionListener(this);//passing current instance
add(b);add(tf);  setSize(300,300);  setLayout(null);
setVisible(true);  }
publicvoid actionPerformed(ActionEvent e){
tf.setText("Welcome");  }
publicstaticvoid main(String args[]){
new AEvent();  }  }
```

public void setBounds(int xaxis, int yaxis, int width, int height); have been used in the above example that sets the position of the component it may be button, textfield etc.



2) Java event handling by outer class

```
import java.awt.*;
import java.awt.event.*;
class AEvent2 extends Frame{
TextField tf;  AEvent2(){
tf=new TextField();
tf.setBounds(60,50,170,20);
Button b=new Button("click me");
b.setBounds(100,120,80,30);
Outer o=new Outer(this);
b.addActionListener(o);//passing outer class instance
```

```java
add(b);add(tf);
setSize(300,300);  setLayout(null);  setVisible(true);  }
publicstaticvoid main(String args[]){  new AEvent2();  }  }
import java.awt.event.*;
class Outer implements ActionListener{
AEvent2 obj;
Outer(AEvent2 obj){
this.obj=obj;  }
publicvoid actionPerformed(ActionEvent e){
obj.tf.setText("welcome");  }  }
```

3) Java event handling by anonymous class

```java
import java.awt.*;
import java.awt.event.*;
class AEvent3 extends Frame{
TextField tf;
AEvent3(){
tf=new TextField();
tf.setBounds(60,50,170,20);
Button b=new Button("click me");
b.setBounds(50,120,80,30);
b.addActionListener(new ActionListener(){
publicvoid actionPerformed(){
tf.setText("hello");  } });
add(b);add(tf); setSize(300,300);
setLayout(null);  setVisible(true);  }
publicstaticvoid main(String args[]){   new AEvent3();  }  }
```

➢Java Adapter Casses

Java adapter classes *provide the default implementation of listener*interfaces. If you inherit the adapter class, you will not be forced to provide the implementation of all the methods of listener interfaces.

The adapter classes are found in **java.awt.event**, **java.awt.dnd** and **javax.swing.event**packages. The Adapter classes with their corresponding listener interfaces are given below.

Adapter class	**Listenerinterface**
WindowAdapter	WindowListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
FocusAdapter	FocusListener
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
HierarchyBoundsAdapter	HierarchyBoundsListener

Adapter class	**Listener interface**
DragSourceAdapter	DragSourceListener
DragTargetAdapter	DragTargetListener

Adapter class	**Listener interface**
MouseInputAdapter	MouseInputListener
InternalFrameAdapter	InternalFrameListener

java.awt.event Adapter classes

java.awt.dnd Adapter classes

javax.swing.event Adapter classes

➢Java Swing

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

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.

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** where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing **follows MVC**.

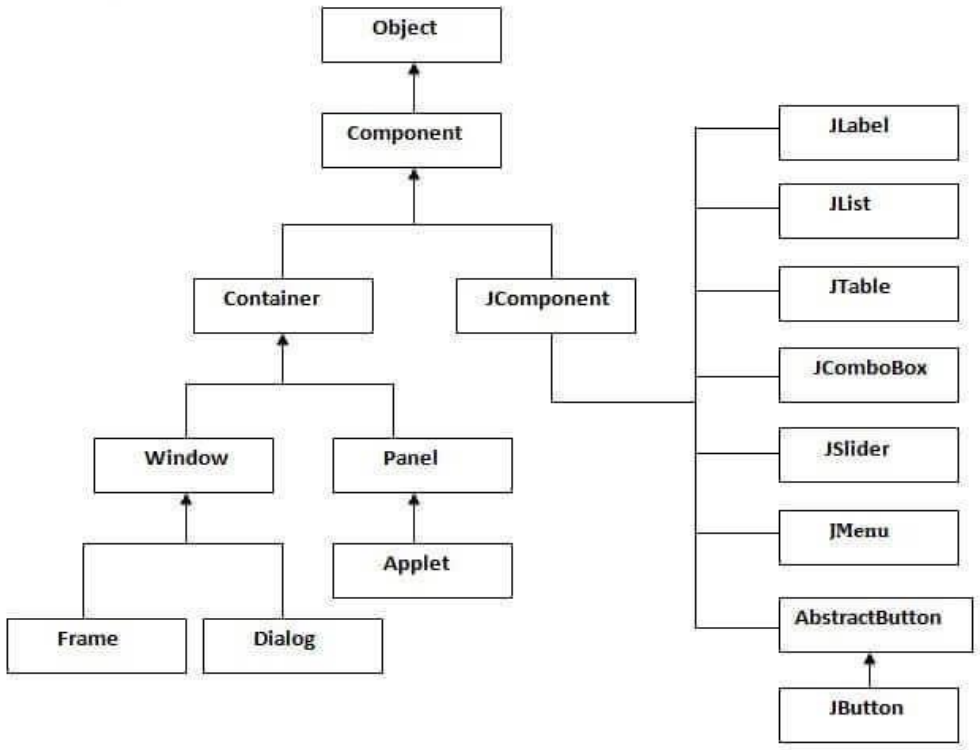
Difference between AWT and Swing There are many differences between java awt and swing that are given below.

What is JFC

The Java Foundation Classes (JFC) are a set of GUI components which simplify the development of desktop applications.

Hierarchy of Java Swing classes

The hierarchy of java swing API is given below.



Commonly used Methods of Component class **The methods of Component class are widely used in java swing that are given below.**

Method	**Description**
public void add(Component c)	add a component on another component.
public void setSize(int width,int height)	sets size of the component.
public void setLayout(LayoutManager m)	sets the layout manager for the component.
public void setVisible(boolean b)	sets the visibility of the component. It is by default false.

Java Swing Examples

There are two ways to create a frame:

○By creating the object of Frame class (association) ○By extending Frame class (inheritance)

We can write the code of swing inside the main(), constructor or any other method.

Simple Java Swing Example

File: FirstSwingExample.java

```
import javax.swing.*;
publicclass FirstSwingExample {
publicstaticvoid main(String[] args) {
JFrame f=new JFrame();//creating instance of JFrame
JButton b=new JButton("click");//creating instance of JButton
b.setBounds(130,100,100, 40);//x axis, y axis, width, height
f.add(b);//adding button in JFrame
f.setSize(400,500);//400 width and 500 height
f.setLayout(null);//using no layout managers
f.setVisible(true);//making the frame visible  }  }
```

Java JFrame

The javax.swing.JFrame class is a type of container which inherits the java.awt.Frame class. JFrame works like the main window where components like labels, buttons, textfields are added to create a GUI.

Unlike Frame, JFrame has the option to hide or close the window with the help of setDefaultCloseOperation(int) method.

Nested Class

Modifier and Type	**Class**	**Description**
protected class	JFrame.AccessibleJFrame	This class implements accessibility support for the JFrame class.

Fields

Modifier and Type	**Field**	**Description**
protected AccessibleContext	accessibleContext	The accessible context property.
static int	EXIT_ON_CLOSE	The exit application
		default window close operation.

protected JRootPane	rootPane	The JRootPane instance that manages the contentPane and optional menuBar for this frame, as well as the glassPane.
protected boolean	rootPaneCheckingEnabled	If true then calls to add and setLayout will be forwarded to the contentPane.

Constructors

Constructor	**Description**
JFrame()	It constructs a new frame that is initially invisible.
JFrame(GraphicsConfiguration gc)	It creates a Frame in the specified GraphicsConfiguration of a screen device and a blank title.
JFrame(String title)	It creates a new, initially invisible Frame with the specified title.
JFrame(String title, GraphicsConfiguration gc)	It creates a JFrame with the specified title and the specified GraphicsConfiguration of a screen device.

Useful Methods

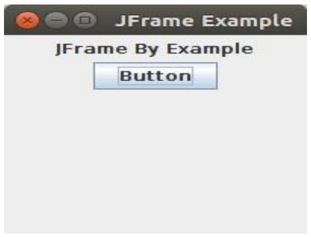
Modifier and Type	**Method**	**Description**

prote cted void	addImpl(Component constraints , int index) c o m p , O b j e ct	Adds the specified child Component.
pr ote cte d	createRo otPane()	Called by the
JRoot Pane		constructor.
protected void	frameInit()	Called by the constructors to init the JFrame properly.
void	setContentPane(Containe contentPane)	It se ts th e content Pane property
static void	setDefaultLookAndFeelDecorated(boole an defaultLookAndFeelDecorated)	Provides a hint as to whether or not newly
void	setIconImage(Image image)	It sets the image to be displayed as the icon for this window.
void	setJMenuBar(JMenuBar menubar)	It sets the menubar for this frame.

JRoot Pane	gtRootPane()	It returns the rootPane object for this frame.
TransferHandler	getTransferHandler()	It gets the transfer Handler property.

JFrame Example

```
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.Jpanel;
publicclass JFrameExample {
publicstaticvoid main(String s[]) {
JFrame frame = new JFrame("JFrame Example");
JPanel panel = new JPanel();
panel.setLayout(new FlowLayout());
JLabel label = new JLabel("JFrame By Example");
JButton button = new JButton();
button.setText("Button");
panel.add(label);  panel.add(button);
frame.add(panel); frame.setSize(200, 300);
frame.setLocationRelativeTo(null);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);  }  }
```



Output

Java JTextArea

The object of a JTextArea class is a multi line region that displays text. It allows the editing of multiple line text. It inherits JTextComponent class

JTextArea class declaration

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

```
publicclass JTextArea extends JTextComponent
```

Commonly used Constructors:

Constructor	Description
JTextArea()	Creates a text area that displays no text initially.
JTextArea(String s)	Creates a text area that displays specified text initially.
JTextArea(int row, int column)	Creates a text area with the specified number of rows and columns that displays no text initially.
JTextArea(String s, int row, int column)	Creates a text area with the specified number of rows and columns that displays specified text.

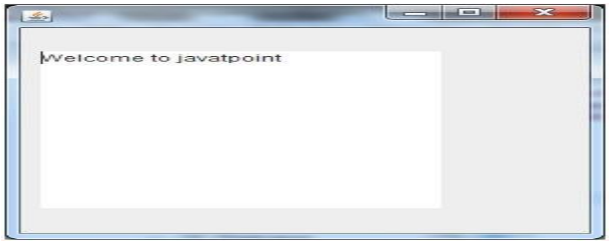
Commonly used Methods:

Methods	Description
void setRows(int rows)	It is used to set specified number of rows.

void setColumns(int cols)	It is used to set specified number of columns.
void setFont(Font f)	It is used to set the specified font.
void insert(String s, int position)	It is used to insert the specified text on the specified position.
void append(String s)	It is used to append the given text to the end of the document.

Java JTextArea Example

```
import javax.swing.*;
publicclass TextAreaExample
{  TextAreaExample(){
JFrame f= new JFrame();
JTextArea area=new JTextArea("Welcome to javatpoint");
area.setBounds(10,30, 200,200);
f.add(area);
f.setSize(300,300);
f.setLayout(null);
f.setVisible(true);  }
publicstaticvoid main(String args[])
{   new TextAreaExample();  }}
```



➢Swing Controls in Java

1. **JLabel** ,**JRadioButton** , **ButtonGroup** ,**JCheckBox** ,**JTextField**,**JTextArea**
2. **JButtonBorder** , **JComboBox** ,**JTabbedPane** , **JPasswordField**

3. **Look and Feel Management in Java Swing**

JLabel

The object of the **JLabel** class may be a component for putting text during a container. It's used to display one line of read-only text. It inherits the **JComponent** class.

Declaration: public class JLabel extends JComponent implements SwingConstants, Accessible

Syntax: JLabel jl = new JLabel();JLabel

Constructors

1. **JLabel():** It is used to create a JLabel instance with no image and with an empty string for the title.
2. **JLabel(String s):** It is used to create a JLabel instance with the specified text.
3. **JLabel(Icon i):** It is used to create a JLabel instance with the specified image.
4. **JLabel(String s, Icon I, int horizontalAlignment):** It is used to create a JLabel instance with the specified text, image, and horizontal alignment.

JRadioButton

This component allows the user to select only one item from a group item. By using the JRadioButton component you can choose one option from multiple options.

Declaration: public class JRadioButton extends JToggleButton implements Accessible

Syntax: JRadioButton jrb = new JRadioButton();

JRadioButton Constructors

1. **JRadioButton():** It is used to create an unselected radio button with no text.
2. **JRadioButton(Label):** It is used to create an unselected radio button with specified text.
3. **JRadioButton(Label, boolean):** It is used to create a radio button with the specified text and selected status.

ButtonGroup

This class is used to place multiple RadioButton into a single group. So the user can select only one value from that group. We can add RadioButtons to the ButtonGroup by using the add method.

Example : add(jrb);

Syntax : ButtonGroup bg = new ButtonGroup();

JCheckBox

This component allows the user to select multiple items from a group of items. It is used to create a CheckBox. It is used to turn an option ON or OFF.

Declaration: **public class JCheckBox extends JToggleButton implements Accessible**JCheckBox Constructors

1. **JCheckBox():** It is used to create an initially unselected checkbox button with no text, no icon.
2. **JCheckBox(Label):** It is used to create an initially unselected checkbox with text.
3. **JCheckBox(Label, boolean):** It is used to create a checkbox with text and specifies whether or not it is initially selected.
4. **JCheckBox(Action a):** It is used to create a checkbox where properties are taken from the Action supplied.

JTextField

The JTextField component allows the user to type some text in a single line. It basically inherits the JTextComponent class.

Declaration: **public class JTextField extends JTextComponent implements SwingConstants**

Syntax: **JTextField jtf = new JTextField();**

JTextField Constructors

1. **JTextField():** It is used to create a new Text Field.
2. **JTextField(String text):** It is used to create a new Text Field initialized with the specified text.
3. **JTextField(String text, int columns):** It is used to create a new Text field initialized with the specified text and columns.
4. **JTextField(int columns):** It is used to create a new empty TextField with the specified number of columns.

JTextArea

The JTextArea component allows the user to type the text in multiple lines.

It also allows the editing of multiple-line text. It basically inherits the JTextComponent class.

Declaration: **public class JTextArea extends JTextComponent**

Syntax: **JTextArea jta = new JTextArea();**JTextArea Constructors

1. **JTextArea():** It is used to create a text area that displays no text initially.
2. **JTextarea(String s):** It is used to create a text area that displays specified text initially.
3. **JTextArea(int row, int column):** It is used to create a text area with the specified number of rows and columns that display no text initially.

4. **JTextarea(String s, int row, int column):** It is used to create a text area with the specified number of rows and columns that display specified text.

JButton

This component can be used to perform some operations when the user clicks on it. When the button is pushed, the application results in some action. It basically inherits the AbstractButton class.

Declaration: public class JButton extends AbstractButton implements Accessible

Syntax: JButton jb = new JButton();

JButton Constructors

1. **JButton():** It is used to create a button with no text and icon.
2. **JButton(String s):** It is used to create a button with the specified text.
3. **JButton(Icon i):** It is used to create a button with the specified icon object.

Border

The border is an interface using which we can apply a border to every component. To create the borders we have to use the methods available in **BorderFactory** class. We can apply the created border to any component by using **SetBorder()** method.**Component.setBorder(Border);**Methods of Border

1. **Border createLineBorder(Color, int), Border createEtchedBorder(int, Color, Color), Border createBevelBorder(int, Color, Color), MatteBorder createMatteBorder(int, int, int, int, Icon) TitledBorder createTitledBorder(Border, String, int, int, Font, Color),. CompoundBorder createCompoundBorder(Border, Border):**

JComboBox

This component will display a group of items as a drop-down menu from which one item can be selected. It basically inherits JComponent class. We can add the items to the ComboBox by using the addItem() method.

Example: jcb.addItem(item);

Declaration: public class JComboBox extends JComponent implements ItemSelectable, ListDataListener, ActionListener, Accessible

Syntax: JComboBox jcb = new JComboBox();

JComboBox Constructors

1. **JComboBox():** It is used to create a JComboBox with a default data model.

2. **JComboBox(Object[] items):** It is used to create a JComboBox that contains the elements in the specified array.
3. **JComboBox(Vector<?> items):** It is used to create a JComboBox that contains the elements in the specified Vector.

JTabbedPane

It is a pane that can contain tabs and each tab can display any component in the same pane. It is used to switch between a group of components by clicking on a tab with a given title or icon. To add the tabs to the JTabbedPane we can use the following methods:

jtp.add(TabName, Components)jtp.addTab(TabName, Components)

Declaration: **public class JTabbedPane extends JComponent implements Serializable, Accessible, SwingConstants**

Syntax: **JTabbedPane jtp = new JTabbedPane();**

JTabbedPane Constructors

1. **JTabbedPane():** It is used to create an empty TabbedPane.
2. **JTabbedPane(int tabPlacement):** It is used to create an empty TabbedPane with a specified tab placement.
3. **JTabbedPane(int tabPlacement, int tabLayoutPolicy):** It is used to create an empty TabbedPane with a specified tab placement and tab layout policy

JPasswordField

It is a text component specialized for password entry. It allows the editing of a single line of text. It basically inherits the JTextField class.

Declaration: public class JPasswordField extends JTextField

Syntax: JPasswordField jpf = new JPasswordField(); JPasswordFiled Constructors

1. **JPasswordField():** It is used to construct a new JPasswordField, with a default document, null starting text string, and column width.
2. **JPasswordField(int columns):** It is used to construct a new empty JPasswordField with the specified number of columns.
3. **JPasswordField(String text):** It is used to construct a new JPasswordField initialized with the specified text.
4. **JPasswordField(String text, int columns):** It is used to construct a new JPasswordField initialized with the specified text and columns.

Look and Feel Management in Java Swing

To manage a look and feel for the UI components swing package provides look and feel managers. In the Swing environment, look and feel are controlled by the UI Manager class.

Look and Feel Managements are shown below:

1. MetalLookAndFeel ,MotifLookAndFeel ,NimbusLooAndFeel

2. WindowsLookAndFeel ,WindowsClassicLookAndFeelThe above looks and feels are the predefined classes.Steps to apply a look and feel:

1. Decide a look and feel type and inform a UI Manager to set a given look and feel and this will be done using setLookAndFeel().
2. Once the look and feel are set we can apply to the UI Component or Component tree

AWT Control Fundamentals

The AWT supports the following types of controls:

Labels ,Push buttons , Check boxes , Choice lists ,Lists ,Scroll bars , Tex
Editing

These controls are subclasses of **Component**. Although this is not a particularly ric set of controls, it is sufficient for simple applications. (Note that both Swing and JavaF provide a substantially larger, more sophisticated set of controls.)

Short Questions

1.what are the applet basics?

2.what is the HTML applet tag ?

3.what are the adapter classes?

4.what is layout manager?

Long Questions

5. Features of applet and explain?

6.Explain about SWING?

7. Explain about layout managers and menus ?

8. Explain the events in GUI programming

UNIT-5

➢**Networking programming with java**

➢Java Networking

Java Networking is a concept of connecting two or more computing devices together so that we can share resources. Java socket programming provides facility to share data between different computing devices.

Advantage of Java Networking

1. sharing resources
2. centralize software management

Java Networking Terminology

The widely used java networking terminologies are given below:

1. IP Address
2. Protocol
3. Port Number
4. MAC Address
5. Connection-oriented and connection-less protocol
6. Socket

1) IP Address

IP address is a unique number assigned to a node of a network e.g. 192.168.0.1 . It is composed of octets that range from 0 to 255.

2) Protocol

A protocol is a set of rules basically that is followed for communication. For example:

- TCP, FTP ,Telnet ,SMTP,POP etc

3) Port Number

The port number is used to uniquely identify different applications. It acts as a communication endpoint between applications.

The port number is associated with the IP address for communication between two applications.

4) MAC Address

MAC (Media Access Control) Address is a unique identifier of NIC (Network Interface Controller). A network node can have multiple NIC but each with unique MAC.

5) Connection-oriented and connection-less protocol

In connection-oriented protocol, acknowledgement is sent by the receiver. So it is reliable but slow. The example of connection-oriented protocol is TCP.

But, in connection-less protocol, acknowledgement is not sent by the receiver. So it is not reliable but fast. The example of connection-less protocol is UDP.

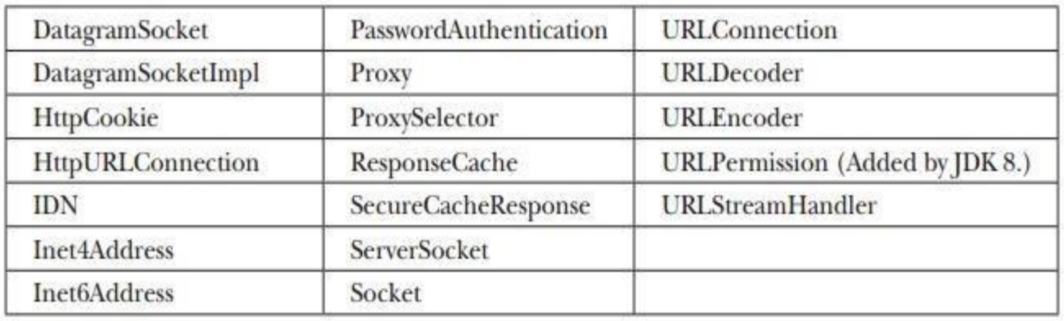
6) Socket

A socket is an endpoint between two way communication.

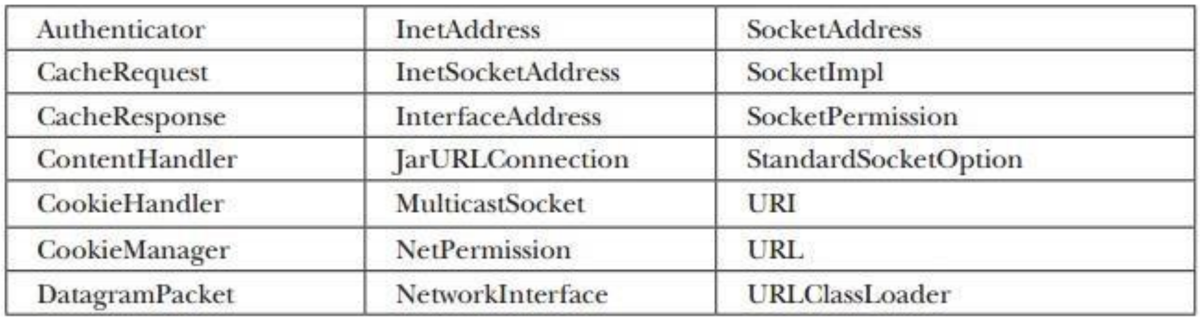
Visit next page for java socket programming. java.net

package

The java.net package provides many classes to deal with networking applications in Java. A list of these classes is given below:



DatagramSocket	PasswordAuthentication	URLConnection
DatagramSocketImpl	Proxy	URLDecoder
HttpCookie	ProxySelector	URLEncoder
HttpURLConnection	ResponseCache	URLPermission (Added by JDK 8.)
IDN	SecureCacheResponse	URLStreamHandler
Inet4Address	ServerSocket	
Inet6Address	Socket	



Authenticator	InetAddress	SocketAddress
CacheRequest	InetSocketAddress	SocketImpl
CacheResponse	InterfaceAddress	SocketPermission
ContentHandler	JarURLConnection	StandardSocketOption
CookieHandler	MulticastSocket	URI
CookieManager	NetPermission	URL
DatagramPacket	NetworkInterface	URLClassLoader

➢What Is a Network Interface?

INTERFACES	
CookiePolicy	CookieStore
FileNameMap	SocketOption
InetAddress	ServerSocket
SocketImplFactory	ProtocolFamily

A network interface is the point of interconnection between a computer and a private or public network. A network interface is generally a network interface card (NIC). Instead, the network interface can be implemented in software. For example, the loopback interface (127.0.0.1 for IPv4 and ::1 for IPv6) is not a physical device but a piece of software simulating a network interface.

➢InetAddress

Inet Address encapsulates both numerical IP address and the domain name for that address. Inet address can handle both IPv4 and Ipv6 addresses. Inet Address class has no visible constructor. To create an inet Address object, you have to use **Factory methods**.

Three commonly used Inet Address factory methods are.

1. static *InetAddress***getLocalHost()** throws **UnknownHostException**
2. static *InetAddress***getByName** (*String hostname*) throws **UnknownHostException**
3. static *InetAddress[ ]***getAllByName** (*String hostname*) throws **UnknownHostException**

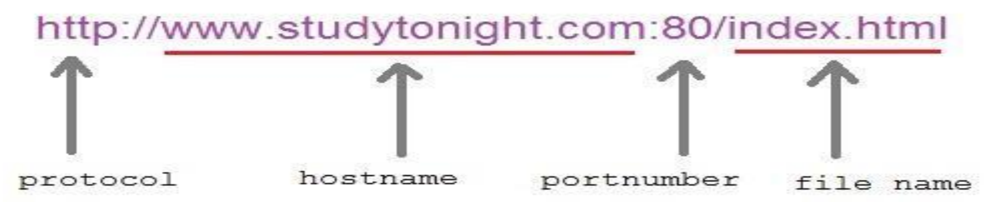
➢Socket and ServerSocket Class

Socket is foundation of modern networking, a socket allows single computer to serve many different clients at once. Socket establishes connection through the use of port, which is a numbered socket on a particular machine. Socket communication takes place via a protocol. Socket provides communication mechanism between two computers using TCP. There are two kind of TCP sockets in Java. One is for server and other is for client.

- **ServerSocket** is for servers.
- **Socket** class is for client.

URL class

Java URL Class present in java.net package, deals with URL (Uniform Resource Locator) which uniquely identify or locate resources on internet.



Important Methods of URL class

- **getProtocol() :** Returns protocol of URL
- **getHost() :** Returns hostname(domain name) of URL
- **getPort() :** Returns port number of URL

- **getFile() :** Returns filename of URL

Inet4Address and Inet6Address

Java includes support for both IPv4 and IPv6 addresses. Because of this, two subclassesof **InetAddress** were created: **Inet4Address** and **Inet6Address**. **Inet4Address** represents atraditional-style IPv4 address. **Inet6Address** encapsulates a newer IPv6 address. Because they are subclasses of **InetAddress,** an **InetAddress** reference can refer to either. This is one way that Java was able to add IPv6 functionality without breaking existing code or adding many more classes. For the most part, you can simply use **InetAddress** when working with IP addresses because it can accommodate both styles

Java Socket Programming

Java Socket programming is used for communication between the applications running on different JRE.

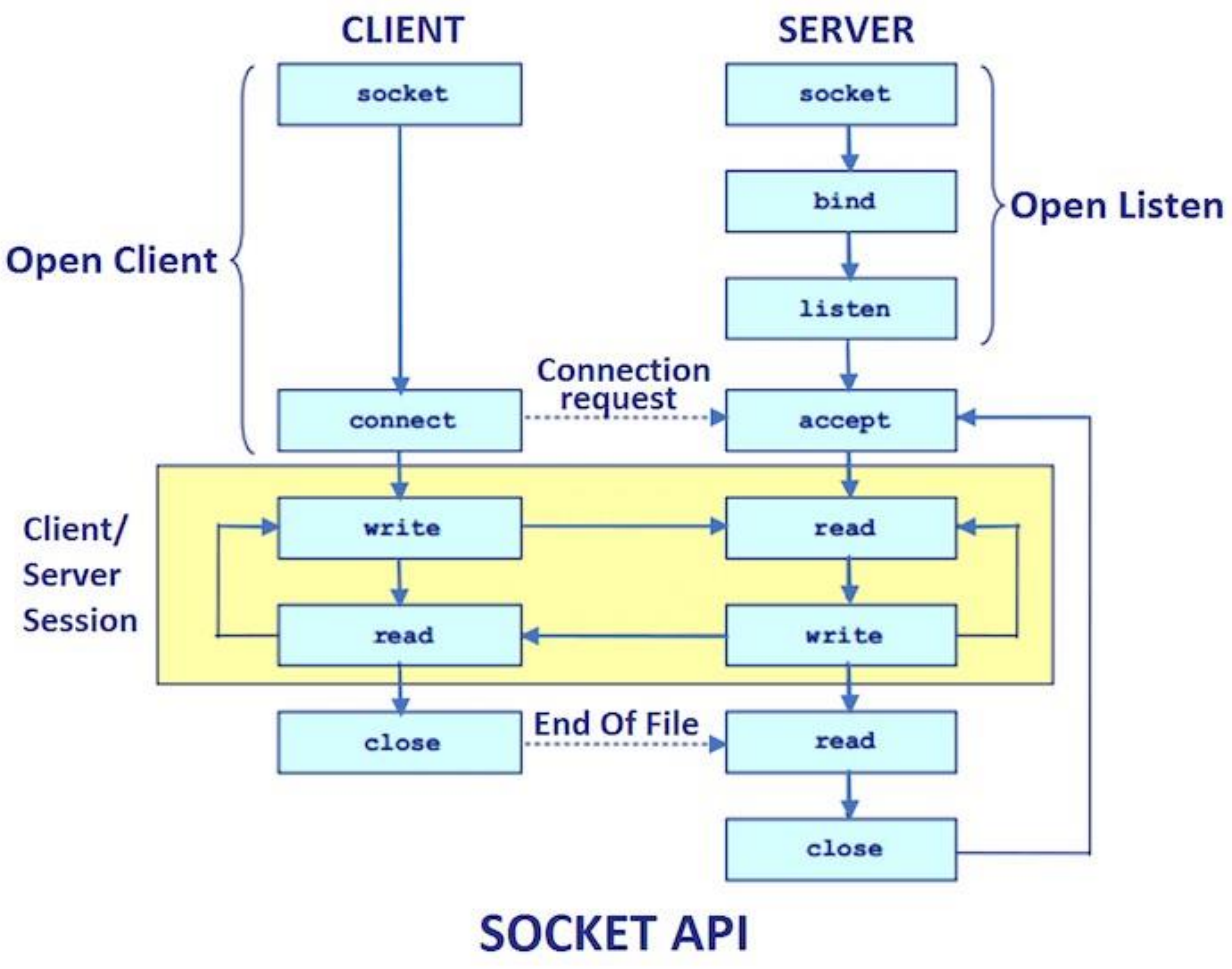
Java Socket programming can be connection-oriented or connection-less.

Socket and ServerSocket classes are used for connection-oriented socket programming and DatagramSocket and DatagramPacket classes are used for connection-less socket programming.

The client in socket programming must know two information:

1. IP Address of Server, and
2. Port number.

Here, we are going to make one-way client and server communication. In this application, client sends a message to the server, server reads the message and prints it. Here, two classes are being used: Socket and ServerSocket. The Socket class is used to communicate client and server. Through this class, we can read and write message. The ServerSocket class is used at server-side. The accept() method of ServerSocket class blocks the console until the client is connected. After the successful connection of client, it returns the instance of Socket at server-side.



➢Factory Methods

The **InetAddress** class has no visible constructors. To create an **InetAddress** object, you have to use one of the available factory methods. *Factory methods* are merely a convention whereby static methods in a class return an instance of that class. This is done in lieu of overloading a constructor with various parameter lists when having unique method names makes the results much clearer. Three commonly used **InetAddress** factory methods are shown here:

static InetAddress getLocalHost( ) throws UnknownHostException

static InetAddress getByName(String *hostName*) throws UnknownHostException

static InetAddress[ ] getAllByName(String *hostName*) throws

UnknownHostException

The **getLocalHost( )** method simply returns the **InetAddress** object that represents the local host. The **getByName( )** method returns an **InetAddress** for a host name passed to it. If these methods are unable to resolve the host name, they throw an **UnknownHostException**.

The **getAllByName( )** factory method returns an array of **InetAddress**es that represent all of theaddresses that a particular name resolves to. It will also throw an **UnknownHostException** if it can't resolve the name to at least one address.

InetAddress also includes the factory method **getByAddress( ),** which takes an IPaddress and returns an **InetAddress** object. Either an IPv4 or an IPv6 address can be used.

➢Instance Methods

The **InetAddress** class has several other methods, which can be used on the objects returned by the methods just discussed. Here are some of the more commonly used methods:

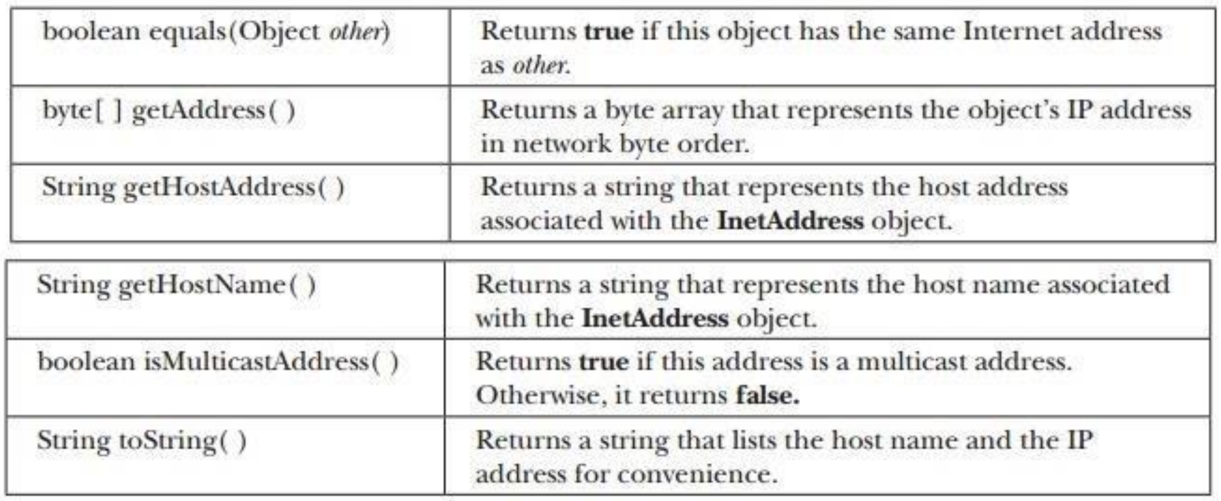
boolean equals(Object other) : Returns true if this object has the same Internet address as other.

byte[ ] getAddress( ) : Returns a byte array that represents the object's IP address in network byte order.

String getHostAddress( ) : Returns a string that represents the host address associated with the InetAddress object.
String getHostName( ) : Returns a string that represents the host name associated with the InetAddress object.

boolean isMulticastAddress( ) : Returns true if this address is a multicast address. Otherwise, it returns false.

String toString( ) : Returns a string that lists the host name and the IP address for convenience.



boolean equals(Object *other*)	Returns **true** if this object has the same Internet address as *other*.
byte[ ] getAddress( )	Returns a byte array that represents the object's IP address in network byte order.
String getHostAddress( )	Returns a string that represents the host address associated with the **InetAddress** object.
String getHostName( )	Returns a string that represents the host name associated with the **InetAddress** object.
boolean isMulticastAddress( )	Returns **true** if this address is a multicast address. Otherwise, it returns **false**.
String toString( )	Returns a string that lists the host name and the IP address for convenience.

➢Java URL

The **Java URL** class represents an URL. URL is an acronym for Uniform Resource Locator. It points to a resource on the World Wide Web. For

example:

A URL contains many information:

1. **Protocol:** In this case, http is the protocol.
2. **Server name or IP Address:** In this case, www.javatpoint.com is the server name.
3. **Port Number:** It is an optional attribute. If we write http//ww.javatpoint.com:80/sonoojaiswal/ , 80 is the port number. If port number is not mentioned in the URL, it returns -1.
4. **File Name or directory name:** In this case, index.jsp is the file name.

Constructors of Java URL class

URL(String spec)

Creates an instance of a URL from the String representation.

URL(String protocol, String host, int port, String file)

Creates an instance of a URL from the given protocol, host, port number, and file.

URL(String protocol, String host, int port, String file, URLStreamHandler handler)

Creates an instance of a URL from the given protocol, host, port number, file, and handler.

URL(String protocol, String host, String file)

Creates an instance of a URL from the given protocol name, host name, and file name.

URL(URL context, String spec)

Creates an instance of a URL by parsing the given spec within a specified context.

URL(URL context, String spec, URLStreamHandler handler)

Creates an instance of a URL by parsing the given spec with the specified handler within a given context.

Commonly used methods of Java URL class

The java.net.URL class provides many methods. The important methods of URL class are given below.

Method	Description
public String getProtocol()	it returns the protocol of the URL.
public String getHost()	it returns the host name of the URL.
public String getPort()	it returns the Port Number of the URL.
public String getFile()	it returns the file name of the URL.

public String getAuthority()	it returns the authority of the URL.
public String toString()	it returns the string representation of the URL.
public String getQuery()	it returns the query string of the URL.
public String getDefaultPort()	it returns the default port of the URL.
public URLConnection openConnection()	it returns the instance of URLConnection i.e. associated with this URL.
public boolean equals(Object obj)	it compares the URL with the given object.
public Object getContent()	it returns the content of the URL.
public String getRef()	it returns the anchor or reference of the URL.
public URI toURI()	it returns a URI of the URL.

➢Java URLConnection class

The **Java URLConnection** class represents a communication link between the URL and the application. This class can be used to read and write data to the specified resource referred by the URL.

How to get the object of URLConnection class

The openConnection() method of URL class returns the object of URLConnection class. Syntax:

```
1.public URLConnection openConnection()throws IOException{}
```

Displaying source code of a webpage by URLConnecton class

The URLConnection class provides many methods, we can display all the data of a webpage by using the getInputStream() method. The getInputStream() method returns all the data of the specified URL in the stream that can be read and displayed.

Example of Java URLConnection class

```
import java.io.*;
import java.net.*;
publicclass URLConnectionExample {
publicstaticvoid main(String[] args){
try{
URL url=new URL("http://www.javatpoint.com/java-tutorial");
URLConnection urlcon=url.openConnection();
InputStream stream=urlcon.getInputStream();
int i;
while((i=stream.read())!=-1){
System.out.print((char)i); }}
}catch(Exception e){System.out.println(e);}  }}
```

Example of Java URL class

```
//URLDemo.java
import java.net.*;
publicclass URLDemo{
publicstaticvoid main(String[] args){  try{
URL url=new URL("http://www.javatpoint.com/java-tutorial");
System.out.println("Protocol: "+url.getProtocol());
System.out.println("Host Name: "+url.getHost());
System.out.println("Port Number: "+url.getPort());
System.out.println("File Name: "+url.getFile());
}catch(Exception e){System.out.println(e);}  }}
```

➢Creating a server that sends data:

Java Socket Server Examples(TCP/IP)

1. ServerSocket API

The **ServerSocket**class is used to implement a server program. Here are the typical steps involve in developing a server program:

1. Create a server socket and bind it to a specific port number
2. Listen for a connection from the client and accept it. This results in a client socket is created for the connection.
3. Read data from the client via an InputStream obtained from the client socket.
4. Send data to the client via the client socket's OutputStream.
5. Close the connection with the client.

The steps 3 and 4 can be repeated many times depending on the protocol agreed between the server and the client.

The steps 1 to 5 can be repeated for each new client. And each new connection should be handled by a separate thread.

Create a Server Socket:

Create a new object of the ServerSocket class by using one of the following constructors:

- ServerSocket(int port): creates a server socket that is bound to the specified port number. The maximum number of queued incoming connections is set to 50 (when the queue is full, new connections are refused).
- ServerSocket(int port, int backlog): creates a server socket that is bound to the specified port number and with the maximum number of queued connections is specified by the backlog parameter.
- ServerSocket(int port, int backlog, InetAddress bindAddr): creates a server socket and binds it to the specified port number and a local IP address

Listen for a connection:

Once a ServerSocket instance is created, call accept() to start listening for incoming client requests:

```
Socket socket = serverSocket.accept();
```

Note that the accept() method blocks the current thread until a connection is made. And the connection is represented by the returned Socket object.

➢Read data from the client:

Once a Socket object is returned, you can use its InputStream to read data sent from the client like this:

```
InputStream input = socket.getInputStream();
```

The InputStream allows you to read data at low level: read to a byte array. So if you want to read the data at higher level, wrap it in an InputStreamReader to read data as characters:

```
InputStreamReader reader = new InputStreamReader(input);
int character = reader.read();  // reads a single character
```

You can also wrap the InputStream in a BufferedReader to read data as String, for mo convenient:

```
BufferedReader reader = new BufferedReader(new InputStreamReader(input));
String line = reader.readLine();    // reads a line of text
```

Send data to the client:

Use the OutputStream associated with the Socket to send data to the client, for example:

```
OutputStream output = socket.getOutputStream();
```

As the OutputStream provides only low-level methods (writing data as a byte array), y can wrap it in a PrintWriter to send data in text format, for example:

```
PrintWriter writer = new PrintWriter(output, true);

writer.println("This is a message sent to the server");
```

The argument true indicates that the writer flushes the data after each method call (au flush).

Close the client connection:

Invoke the close() method on the client Socket to terminate the connection with the clien

```
socket.close();
```

This method also closes the socket's InputStream and OutputStream, and it c throw IOException if an I/O error occurs when closing the socket. Terminate t server:

A server should be always running, waiting for incoming requests from clients. In case server must be stopped for some reasons, call the close() method on the ServerSock instance:

```
serverSocket.close();
```

When the server is stopped, all currently connected clients will be disconnected.

The ServerSocket class also provides other methods which you can consult in Javadochere

The following program demonstrates how to implement a simple server th returns the current date time for every new client. Here's the code:

```
import java.io.*; import
java.net.*; import
java.util.Date;
public class TimeServer {

public static void main(String[] args) { if
(args.length < 1) return;

int port = Integer.parseInt(args[0]);

try (ServerSocket serverSocket = new ServerSocket(port)) {

System.out.println("Server is listening on port " + port);

while (true) {
Socket socket = serverSocket.accept();

System.out.println("New client connected");

OutputStream output = socket.getOutputStream(); PrintWriter writer =
new PrintWriter(output, true);

writer.println(new Date().toString());}
} catch (IOException ex) {
System.out.println("Server      exception:      "      +      ex.getMessage());
ex.printStackTrace();}}}
```

And the following code is for a client program that simply connects to the serv and prints the data received, and then terminates:

```
import java.net.*;
import java.io.*;

public class TimeClient {

   public  static  void  main(String[]  args)  {
if (args.length < 2) return;

       String hostname = args[0];
       int port = Integer.parseInt(args[1]);

       try (Socket socket = new Socket(hostname, port)) {

          InputStream input = socket.getInputStream();
          BufferedReader      reader      =      new      BufferedReader(new
InputStreamReader(input));

          String time = reader.readLine();

          System.out.println(time);


       } catch (UnknownHostException ex) {

          System.out.println("Server not found: " + ex.getMessage());

       } catch (IOException ex) {

          System.out.println("I/O error: " + ex.getMessage());
       }
   } }
```

➢Two way communications between server and client:

File: MyServer.java

```
import java.io.*;
import java.net.*;
publicclass MyServer {
publicstaticvoid main(String[] args){
try{
ServerSocket ss=new ServerSocket(6666);
Socket s=ss.accept();//establishes connection
DataInputStream dis=new DataInputStream(s.getInputStream());
String  str=(String)dis.readUTF();
System.out.println("message= "+str);
ss.close();
}catch(Exception e){System.out.println(e);}  }}
```

File: MyClient.java

```
import java.io.*;
import java.net.*;
publicclass MyClient {
publicstaticvoid main(String[] args) {
try{
Socket s=new Socket("localhost",6666);
DataOutputStream                                    dout=new
DataOutputStream(s.getOutputStream());
dout.writeUTF("Hello Server");
dout.flush();
dout.close();
s.close();
}catch(Exception e){System.out.println(e);}  }}
```

➢Java Database Connectivity with 5 Step

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows: ○Register the Driver class ○Create connection ○Create statement ○Execute queries ○Close connection

1) Register the driver class

The **forName()** method of Class class is used to register the driver class. This method is used to dynamically load the driver class.

Syntax of forName() method **publicstaticvoid** forName(String className)**throws** ClassNotFoundExcept ion

2) Create the connection object

The **getConnection()** method of DriverManager class is used to establish connection with the database.

Syntax of getConnection() method

1. 1) **publicstatic** Connection getConnection(String url)**throws** SQLException

2. 2) **publicstatic** Connection getConnection(String url,String name,String pa ssword)
3. **throws** SQLException

Example to establish connection with the Oracle database

1. Connection con=DriverManager.getConnection(
2. "jdbc:oracle:thin:@localhost:1521:xe","system","password");

3) Create the Statement object

The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.

Syntax of createStatement() method **public** Statement

createStatement()**throws** SQLException

Example to create the statement object

1.Statement stmt=con.createStatement();

4) Execute the query

The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.

Syntax of executeQuery() method **public** ResultSet

executeQuery(String sql)**throws** SQLException

Example to execute query

```
1.  ResultSet rs=stmt.executeQuery("select * from emp");
2.  while(rs.next()){
3.  System.out.println(rs.getInt(1)+" "+rs.getString(2));  }
```

5) Close the connection object

➢ By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

Syntax of close() method **publicvoid**

close()**throws** SQLException

Example to close connection

```
1.con.close();
```

➢ Java Database Connectivity with Oracle

To connect java application with the oracle database, we need to follow 5 following steps. In this example, we are using Oracle 10g as the database. So we need to know following information for the oracle database:

1. **Driver class:** The driver class for the oracle database is **oracle.jdbc.driver.OracleDriver**.
2. **Connection URL:** The connection URL for the oracle10G database is **jdbc:oracle:thin:@localhost:1521:xe** where jdbc is the API, oracle is the database, thin is the driver, localhost is the server name

on which oracle is running, we may also use IP address, 1521 is the port number and XE is the Oracle service name. You may get all these information from the tnsnames.ora file.

3. **Username:** The default username for the oracle database is **system**.
4. **Password:** It is the password given by the user at the time of installing the oracle database.

Create a Table

Before establishing connection, let's first create a table in oracle database. Following is the SQL query to create a table.

```
create table emp(id number(10),name varchar2(40),age number(3));
```

Example to Connect Java Application with Oracle database

In this example, we are connecting to an Oracle database and getting data from **emp** table. Here, **system** and **oracle** are the username and password of the Oracle database.

```
import java.sql.*;
class OracleCon{
publicstaticvoid main(String args[]){
try{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
Statement stmt=con.createStatement();
ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next())
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
con.close();  }catch(Exception e){ System.out.println(e);}  }}
```

To connect java application with the Oracle database ojdbc14.jar file is required to be loaded.
download the jar file ojdbc14.jar

Two ways to load the jar file:

1. paste the ojdbc14.jar file in jre/lib/ext folder
2. set classpath

1) paste the ojdbc14.jar file in JRE/lib/ext folder:

Firstly, search the ojdbc14.jar file then go to JRE/lib/ext folder and paste the jar file here.

2) set classpath:

There are two ways to set the classpath:

- temporary
- permanent

How to set the temporary classpath:

Firstly, search the ojdbc14.jar file then open command prompt and write:

```
1.C:>set classpath=c:\folder\ojdbc14.jar;.;
```

How to set the permanent classpath:

Go to environment variable then click on new tab. In variable name write **classpath** and in variable value paste the path to ojdbc14.jar by appending ojdbc14.jar;.; as
C:\oraclexe\app\oracle\product\10.2.0\server\jdbc\lib\ojdbc14.jar;.;

➢Improving the Performance of a JDBC program:

1. Using batch operations.

You can read your big query and store results in some kind of buffer. And only when buffer is full you should run subquery for all data collected in buffer. This significantly reduces number of SQL statements to execute.

2. Using efficient maps to store content from many selects.

If your records are no so big you can store them all at once event for 4 mln table.

3 jdbc fetch size.

When you load a lot of records from database it is very, very important to set proper fetch size on your jdbc connection. This reduces number of physical hits to database socket and speeds your process

4. PreparedStatement

Use PreparedStatement rather than Statement.

5. Number of sql statements.

Always try to minimize number of sql statements you send to database.

Java performance tuning tips relevent for tuning JDBC usage. The top tips are:

- Use prepared statements. Use parametrized SQL.
- Tune the SQL to minimize the data returned (e.g. not 'SELECT *').
- Minimize transaction conflicts (e.g. locked rows).
- Use connection pooling.
- Try to combine queries and batch updates.
- Use stored procedures.
- Cache data to avoid repeated queries.
- Close resources (Connections, Statements, ResultSets) when finished with.
- Select the fastest JDBC driver.

Topic	**Description**
Closingobjects whennot in use	Describes the importance of closing JDBC driver objects when they are no longer needed.
Managing transaction size	Describes techniques for improving transaction performance.

Working withstatements and result sets	Describes techniques for improving performance when using the Statement or ResultSet objects.
Using	Describes an adaptive buffering feature, which is designed to
adaptivebuffering	retrieve any kind of large-value data without the overhead of server cursors.
Sparsecolumns	Discusses the JDBC driver's support for SQL Server sparse columns.
Preparedstatementmetadata caching forthe JDBC driver	Discusses the techniques for improving performance with prepared statement queries.
Using bulkcopy API forbatch insertoperation	Describes how to enable Bulk Copy API for batch insert operations and its benefits.
Not sending Stringparametersas Unicode	When working with **CHAR**, **VARCHAR**, and **LONGVARCHAR** data, users can set the connection property **sendStringParametersAsUnicode** to false for optimal performance gain.

Short Questions

1.What is the Java networking?
2. What is the inet address?
3.How to knowing IP address?
4.what are the stages in JDBC program?

Long Questions

5. Briefly explain about networking classes and interfaces ?
6. Explain about URL - URL connection classes

7. Explain the how to create server that sends and receives data ?
8. How to improving the performance of JDBC Program and explain?

Question papers for Sample:

S.V.U. COLLEGE OF COMMERCE MANAGEMENT AND COMPUTER SCIENCE :: TIRUPATHI
DEPARTMENT OF COMPUTER SCIENCE
Time:2hours **INTERNAL EXAMINATIONS -I**
MAX MARKS:30

Section-A
Answer any five from the following 5*2=10m
1.what is access protection?
2.what are the importing packages?
3.What are the exception types?
4.what is thread model?
5.what is polymorphism?
6.what is inheritance?
7.Explain about constructors.
8.what is method overriding in java?

Section- B
ANSWER THE FOLLOWING 2*10=20m
UNIT -1
9.Explain the 1.what are the applet basics?
2.what is the HTML applet tag ?
3.what are the adapter classes?
4.what is layout manager? of object oriented programming?
Or
10.Write about inheritance to classes and it's methods?
UNIT-2
11.Explain about packages and interfaces?
Or
12.Discuss about the concept of exception handling?

S.V.U. COLLEGE OF COMMERCE MANAGEMENT AND COMPUTER SCIENCE :: TIRUPATHI
DEPARTMENT OF COMPUTER SCIENCE

Time:2hours **INTERNAL EXAMINATIONS -2** MAX MARKS:30

Section-A

Answer any five from the following 5*2=10m

1.what are the float and byte?
2.what are the two differences between byte and byte stream?
3.what are the collection classes ?
4.what are the character streams?
5.what are the applet basics?
6.what is the HTML applet tag ?
7.what are the adapter classes?
8.what is layout manager?

Section-B

Answer the following. 2*10=20m

Unit-1

9.a)What is the difference between abstract class and inheritance?
b)write a simple java program using threads

Or

10.what are java bulit in exceptions ? Explain Breifly

Unit-2

11.write an applet code to demonstrate the passing the parameters to applet

Or

12.How to create and run an applet and give an example of an applet

MASTER OF COMPUTER APPLICATIONS DEGREE EXAMINATION
First semester
Paper MCA 103:object oriented programming with java
(Under C.B.S.C Revised Regulations w.e.f.2021-2023)

(Common paper to University and all Affiliated by Colleges)
Time:3 hours
Max.Marks:70

PART- A

Answer any 5 questions from the following 5*2=10M

1.a) What are the commands used for compilation and execution of java programs?
b) What is java bytecode? What is JVM?
c) What is a package? Write the syntax to define a “package”.
d) What does Java API package contain?
e) What are the run time errors and logical errors in Java?
f) What is an exception? What are two exception types?
g) What are the properties of hash table?
h) Differentiate between Iterator and for-each.
i) What are the differences between an applet and stand alone java application?
j) What are the methods in applet life cycle?

PART-B

Answer the following 10*5=50M

Unit-1

2.a) Exaplain briefly class, public, static, void, main, string[] and system.out.println() key words.
b) Write a java method to find minimum value in given two values.

OR

3.a) Discuss about precedence of operators and associativity.

b) Explain the polymorphism and overloading with an example.

Unit-2

4.a) Write the benefits of packages and interfaces.
b) How can we add a class to a package? Write about relative and absolute paths.

OR

5.a) Write the differences between interface and abstract class.
b) Write the procedure to a create package with multiple public classes.

Unit-3

6.a) What is exception handling? Explain an example of exception handling in the case of division by zero.
b)Write a simple java program to create threads.

OR

7.a) Write about some Java's built in exceptions.
b) With an example, demonstrate the concept of thread synchronization.

Unit-4

8. With syntax, explain the following utility classes.
a) String Tokenizer b) Date and Calendar c) Scanner.

OR

9.a) Compare and contrast any two collection algorithms.
b) Explain the process of accessing collection through iterator.

Unit-5

10.a) Write the step wise procedure to create and run an applet.
b) List the event classes and Listener Interfaces.

OR

11. Write an applet code to demonstrate parameter passing to applet.