Malla Reddy Engineering College(Autonomous)

Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad ,Telangana-500100
www.mrec.ac.in

Department of Information Technology

II B.TECH I SEM (A.Y.2024-25)

Lecture Notes

On

C0511-Object Oriented Programming through Java

2022-23 Onwards (MR22)	MALLA REDDY ENGINEERING COLLEGE (Autonomous)	B.Tech. III Semester		
Code: C0511	Object Oriented Programming through Java	L	T	P
Credits: 3		3	-	-

Prerequisites: Computer Programming

Course Objectives:

- To understand the basic object-oriented programming concepts and apply them in problem solving.
- To illustrate inheritance concepts for reusing the program.
- To demonstrate multitasking by using multiple threads and event handling
- To develop data-centric applications using JDBC.
- To understand the basics of java console and GUI based programming

MODULE-I: [10 Periods]

Object Oriented Thinking and Java Basics- Need for OOP paradigm, summary of oop concepts, coping with complexity, abstraction mechanisms. A way of viewing world – Agents, responsibility, messages, methods, History of Java, Java buzzwords, data types, variables, scope and lifetime of variables, arrays, operators, expressions, control statements, type conversion and casting, simple java program, concepts of classes, objects, constructors, methods, access control, this keyword, garbage collection, overloading methods and constructors, method binding, inheritance, overriding and exceptions, parameter passing, recursion, nested and inner classes, exploring string class.

Module II: [10 Periods]

Inheritance and Packages– Hierarchical abstractions, Base class object, subclass, subtype, substitutability, forms of inheritance specialization, specification, construction, extension, limitation, combination, benefits of inheritance, costs of inheritance. Member access rules, super uses, using final with inheritance, polymorphism-method overriding, abstract classes, the Object class. Defining, Creating and Accessing a Package, Understanding CLASSPATH, importing packages.

MODULE III: [09 Periods]

Interfaces - Defining an interface, differences between classes and interfaces, implementing interface, applying interfaces, variables in interface and extending interfaces. Exploring java.io.

Exception handling and Multithreading-- Concepts of exception handling, benefits of exception handling, Termination or resumptive models, exception hierarchy, usage of try, catch, throw, throws and finally, built in exceptions, creating own exception subclasses. String handling, Exploring java.util. Differences between multithreading and multitasking, thread life cycle, creating threads, thread priorities, synchronizing threads, inter thread communication, thread groups, daemon threads. Enumerations, autoboxing, annotations, generics.

MODULE IV: [10 Periods

Event Handling: Events, Event sources, Event classes, Event Listeners, Delegation event model, handling mouse and keyboard events, Adapter classes. The AWT class hierarchy, user interface components- labels, button, canvas, scrollbars, text components, check box, checkbox groups, choices,

Lists Panels – scrollpane, dialogs, menubar, graphics, layout manager – layout manager types – border, grid, flow, card and grid bag.

MODULE V: [09 Periods]

Applets – Concepts of Applets, differences between applets and applications, life cycle of an applet, types of applets, creating applets, passing parameters to applets. Swing – Introduction, limitations of AWT, MVC architecture, components, containers, exploring swing- JApplet, JFrame and JComponent, Icons and Labels, text fields, buttons – The JButton class, Check boxes, Radio buttons, Combo boxes, Tabbed Panes, Scroll Panes, Trees, and Tables.

Text Books:

1. Java the complete reference, 7th edition, Herbert schildt, TMH.
2. Understanding OOP with Java, updated edition, T. Budd, Pearson education.

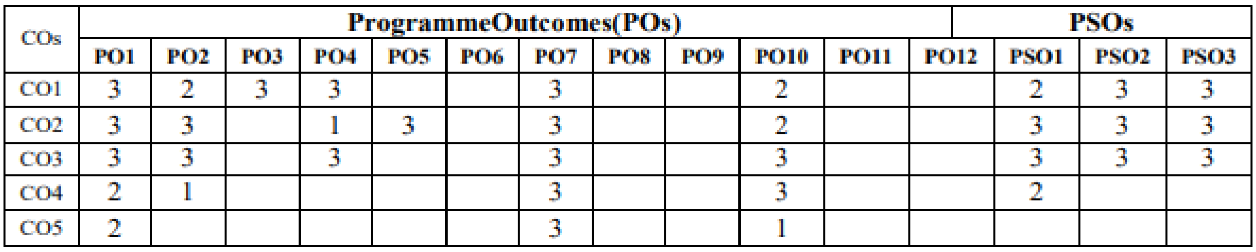
References:

1. An Introduction to programming and OO design using Java, J.Nino and F.A. Hosch, John wiley & sons.
2. An Introduction to OOP, third edition, T. Budd, Pearson education.
3. Introduction to Java programming, Y. Daniel Liang, Pearson education.
4. An introduction to Java programming and object-oriented application development, R.A. Johnson-Thomson.

5. Core Java 2, Vol 1, Fundamentals, Cay.S. Horstmann and Gary Cornell, eighth Edition, Pearson Education.
6. Core Java 2, Vol 2, Advanced Features, Cay.S. Horstmann and Gary Cornell, eighth Edition, Pearson Education
7. Object Oriented Programming with Java, R.Buyya, S.T.Selvi, X.Chu, TMH.
8. Java and Object Orientation, an introduction, John Hunt, second edition, Springer. 9. Maurach's Beginning Java2 JDK 5, SPD.

E-RESOURCES:

1. http://ndl.iitkgp.ac.in/document/xttk-4kfhvUwVlXBW-RPf64_TFk2i4LJhgQFPQ WAEt-Zobbm3twyubjRA1YOe9WVwkN2qGcxBwdHaPdi_mMQ
2. https://ndl.iitkgp.ac.in/result?q={"t":"search","k":"object%20oriented%20programming","s":["type=\"video\""],"b":{"filters":[]}}
3. http://www.rehancodes.com/files/oop-using-c++-by-joyce-farrell.pdf
4. http://www.nptel.ac.in/courses/106103115/36



COs	ProgrammeOutcomes(POs)												PSOs		
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO1	3	2	3	3			3			2			2	3	3
CO2	3	3		1	3		3			2			3	3	3
CO3	3	3		3			3			3			3	3	3
CO4	2	1					3			3			2		
CO5	2						3			1					

UNIT - I

Need for OOP paradigm: Object oriented paradigm is a significant methodology for the development of any software. Most of the architecture styles or patterns such as pipe and filter, data repository, and component-based can be implemented by using this paradigm.

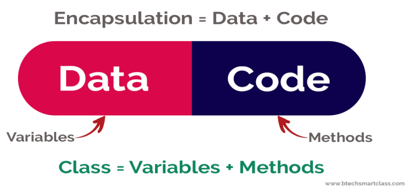
summary of OOP: OOP stands for Object-Oriented Programming. OOP is a programming paradigm in which every program is follows the concept of object. In other words, OOP is a way of writing programs based on the object concept.

The object-oriented programming paradigm has the following core concepts.

- Encapsulation
- Inheritance
- Polymorphism
- Abstraction

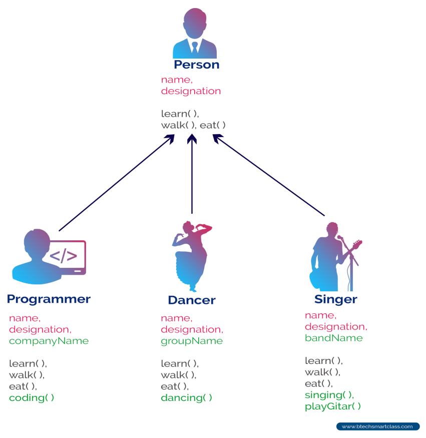
The popular object-oriented programming languages are Smalltalk, C++, Java, PHP, C#, Python, etc.

Encapsulation:



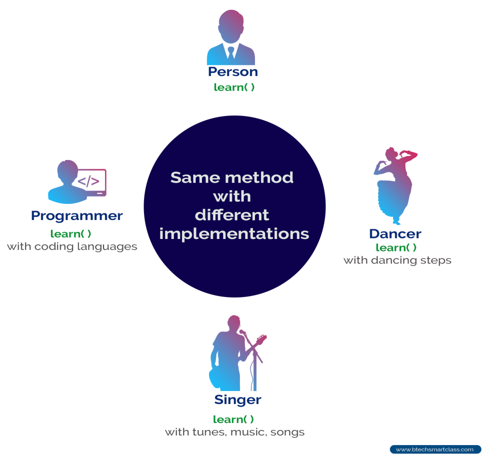
Encapsulation is the process of combining data and code into a single unit (object / class). In OOP, every object is associated with its data and code. In programming, data is defined as variables and code is defined as methods. The java programming language uses the class concept to implement encapsulation.

Inheritance:



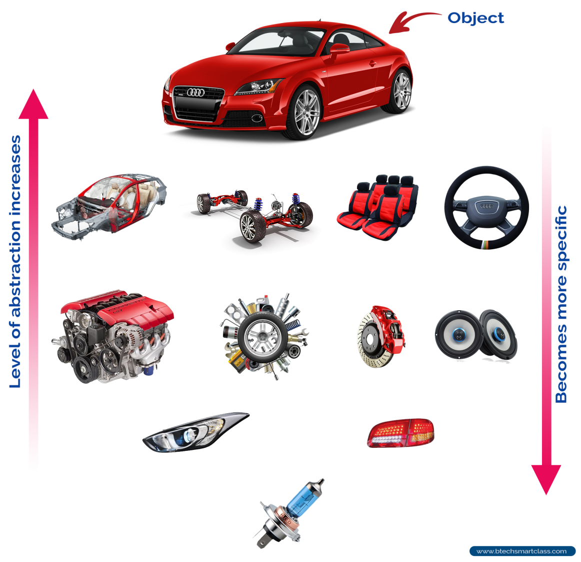
Inheritance is the process of acquiring properties and behaviors from one object to another object or one class to another class. In inheritance, we derive a new class from the existing class. Here, the new class acquires the properties and behaviors from the existing class.

Polymorphism:



Polymorphism is the process of defining same method with different implementation. That means creating multiple methods with different behaviors.

Abstraction:



Abstraction is hiding the internal details and showing only esential functionality. In the abstraction concept, we do not show the actual implemention to the end user, instead we provide only esential things. For example, if we want to drive a car, we does not need to know about the internal functionality like how wheel system works? how brake system works? how music system works? etc.

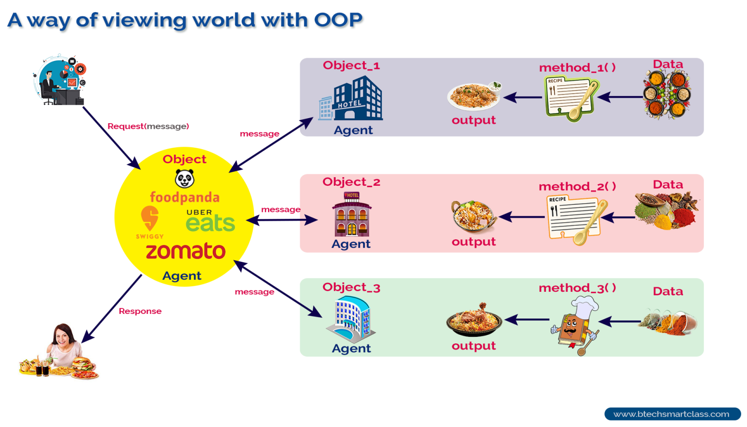
Coping with complexity: Coping with complexity in Object-Oriented Programming (OOP) is crucial for writing maintainable, scalable, and understandable code. Complexity can arise from various sources, including the structure of your classes and objects, interactions between objects, and the overall design of your software. Here are some strategies to help you cope with complexity in OOP:Encapsulation,Abstraction,Inheritance etc.

Abstraction mechanisms : Abstraction is a process of hiding the implementation details and showing only functionality to the user. Ex for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

A way of viewing the world: is an idea to illustrate the object-oriented programming concept with an example of a real-world situation.

Let us consider a situation, I am at my office and I wish to get food to my family members who are at my home from a hotel. Because of the distance from my office to home, there is no possibility of getting food from a hotel myself. So, how do we solve the issue?

To solve the problem, let me call zomato (an agent in food delevery community), tell them the variety and quantity of food and the hotel name from which I wish to delever the food to my family members. Look at the following image.



Agents and Communities

To solve my food delivery problem, I used a solution by finding an appropriate agent (Zomato) and pass a message containing my request. It is the responsibility of the agent (Zomato) to satisfy my request. Here, the agent uses some method to do this. I do not need to know the method that the agent has used to solve my request. This is usually hidden from me.

So, in object-oriented programming, problem-solving is the solution to our problem which requires the help of many individuals in the community. We may describe agents and communities as follows.

An object-oriented program is structured as a community of interacting agents, called objects. Where each object provides a service (data and methods) that is used by other members of the community.

In our example, the online food delivery system is a community in which the agents are zomato and set of hotels. Each hotel provides a variety of services that can be used by other members like zomato, myself, and my family in the community.

Messages and Methods

To solve my problem, I started with a request to the agent zomato, which led to still more requestes among the members of the community until my request has done. Here, the members of a community interact with one another by making requests until the problem has satisfied.

In object-oriented programming, every action is initiated by passing a message to an agent (object), which is responsible for the action. The receiver is the object to whom the message was sent. In response to the message, the receiver performs some method to carry out the request. Every message may include any additional information as arguments.

In our example, I send a request to zomato with a message that contains food items, the quantity of food, and the hotel details. The receiver uses a method to food get delivered to my home.

Responsibilities

In object-oriented programming, behaviors of an object described in terms of responsibilities.

In our example, my request for action indicates only the desired outcome (food delivered to my family). The agent (zomato) free to use any technique that solves my problem. By discussing a problem in terms of responsibilities increases the level of abstraction. This enables more independence between the objects in solving complex problems.

Classes and Instances

In object-oriented programming, all objects are instances of a class. The method invoked by an object in response to a message is decided by the class. All the objects of a class use the same method in response to a similar message.

In our example, the zomato a class and all the hotels are sub-classes of it. For every request (message), the class creates an instance of it and uses a suitable method to solve the problem.

Classes Hierarchies

A graphical representation is often used to illustrate the relationships among the classes (objects) of a community. This graphical representation shows classes listed in a hierarchical tree-like structure. In this more abstract class listed near the top of the tree, and more specific classes in the middle of the tree, and the individuals listed near the bottom.

In object-oriented programming, classes can be organized into a hierarchical inheritance structure. A child class inherits properties from the parent class that higher in the tree.

Method Binding, Overriding, and Exception

In the class hierarchy, both parent and child classes may have the same method which implemented individually. Here, the implementation of the parent is overridden by the child. Or a class may provide multiple definitions to a single method to work with different arguments (overloading).

The search for the method to invoke in response to a request (message) begins with the class of this receiver. If no suitable method is found, the search is performed in the parent class of it. The search continues up the parent class chain until either a suitable method is found or the parent class chain is exhausted. If a suitable method is found, the method is executed. Otherwise, an error message is issued.

java buzz words:

Java is the most popular object-oriented programming language. Java has many advanced features, a list of key features is known as Java Buzz Words. The java team has listed the following terms as java buzz words.

Simple

Java programming language is very simple and easy to learn, understand, and code. Most of the syntaxes in java follow basic programming language C and object-oriented programming concepts are similar to C++. In a java programming language, many complicated features like pointers, operator overloading, structures, unions, etc. have been removed. One of the most useful features is the garbage collector it makes java more simple.

Secure

Java is said to be more secure programming language because it does not have pointers concept, java provides a feature "applet" which can be embedded into a web application. The applet in java does not allow access to other parts of the computer, which keeps away from harmful programs like viruses and unauthorized access.

Portable

Portability is one of the core features of java which enables the java programs to run on any computer or operating system. For example, an applet developed using java runs on a wide variety of CPUs, operating systems, and browsers connected to the Internet.

Object-oriented

Java is said to be a pure object-oriented programming language. In java, everything is an object. It supports all the features of the object-oriented programming paradigm. The primitive data types java also implemented as objects using wrapper classes, but still, it allows primitive data types to archive high-performance.

Robust

Java is more robust because the java code can be executed on a variety of environments, java has a strong memory management mechanism (garbage collector), java is a strictly typed language, it has a strong set of exception handling mechanism, and many more.

Architecture-neutral (or) Platform Independent

Java has invented to archive "write once; run anywhere, any time, forever". The java provides JVM (Java Virtual Machine) to to archive architectural-neutral or platform-independent. The JVM allows the java program created using one operating system can be executed on any other operating system.

Multi-threaded

Java supports multi-threading programming, which allows us to write programs that do multiple operations simultaneously.

Interpreted

Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. The byte code is interpreted to any machine code so that it runs on the native machine.

High performance

Java provides high performance with the help of features like JVM, interpretation, and its simplicity.

Distributed

Java programming language supports TCP/IP protocols which enable the java to support the distributed environment of the Internet. Java also supports Remote Method Invocation (RMI), this feature enables a program to invoke methods across a network.

Dynamic

Java is said to be dynamic because the java byte code may be dynamically updated on a running system and it has a dynamic memory allocation and deallocation (objects and garbage collector).

The history of Java :

1) James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991. The small team of sun engineers called Green Team.

2) Initially it was designed for small, embedded systems in electronic appliances like set-top boxes.

3) Firstly, it was called "Greentalk" by James Gosling, and the file extension was .gt.

4) After that, it was called Oak and was developed as a part of the Green project.

5) JDK 1.0 was released on January 23, 1996.

Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc.

Third generation (programming) language

- A third generation (programming) language (3GL) is a grouping of programming languages that introduced significant enhancements to second generation languages, primarily intended to make the programming language more programmer-friendly.
- English words are used to denote variables, programming structures and commands, and Structured Programming is supported by most 3GLs.
- Commonly known 3GLs are FORTRAN, BASIC, Pascal, JAVA and the C-family (C, C++, C#, Objective-C) of languages. Also known as a high-level programming language.

Difference between C, JAVA and PYTHON

C	JAVA	PYTHON
Compiled Language, Platform- dependent	Platform- dependent Compiled Programming Language, Platform independent	Interpreted Programming Language, Platform independent
Operator overloading is not supported.	Overloading of the operator is not supported.	Overloading of the operator is supported
Multiple inheritance is not supported in C.	Java provides partial multiple inheritance	Provides both single as well as multiple inheritance
Threads are not supported.	Multithreading capability is built-in.	Multithreading is supported.
A small number of libraries available.	Many concepts, such as UI, are supported by the library.	It comes with a large library set that allows it to be used for AI, data science, and other applications.

First Java Program

```
class Simple{
   public static void main(String args[]){
    System.out.println("Hello Java");
   }
}
```

Save the above file as Simple.java.

To compile:

```
javac Simple.java
```

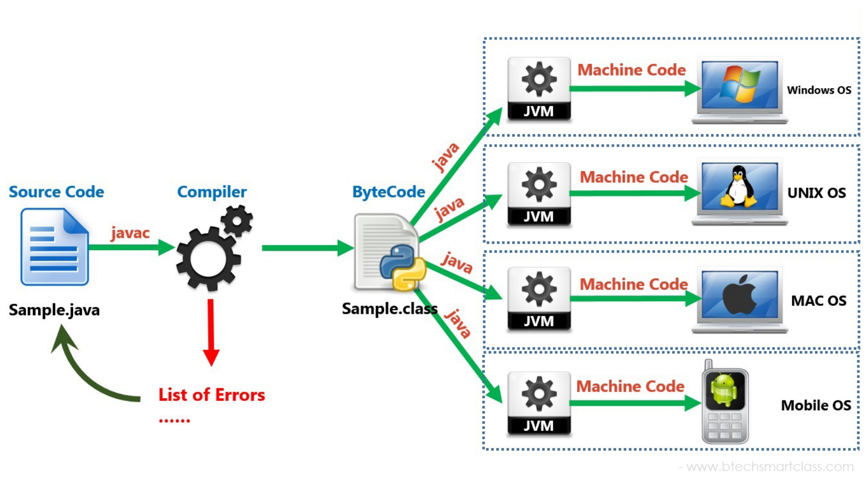
To execute:

```
java Simple
```

Execution Process of Java Program

The following three steps are used to create and execute a java program.

- Create a source code (.java file).
- Compile the source code using javac command.
- Run or execute .class file uisng java command.



Parameters used in First Java Program

class keyword is used to declare a class in Java.

public keyword is an access modifier that represents visibility. It means it is visible to all.

static is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method. The main() method is executed by the JVM, so it doesn't require creating an object to invoke the main() method. So, it saves memory.

void is the return type of the method. It means it doesn't return any value.

main represents the starting point of the program.

String[] args or String args[] is used for command line argument. We will discuss it in coming section.

System.out.println() is used to print statement. Here, System is a class, out is an object of the PrintStream class, println() is a method of the PrintStream class. We will discuss the internal working of System.out.println() statement in the coming section.

Difference between JDK, JRE, and JVM

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

The JVM performs the following main tasks:

- Loads code
- Verifies code
- Executes code

- Provides runtime environment

JRE : the Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

JDK : The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets. It physically exists. It contains JRE + development tools.

JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

- Standard Edition Java Platform
- Enterprise Edition Java Platform
- Micro Edition Java Platform

Variable: A variable is a named memory location used to store a data value.

Syntax: data_type variable_name;

int data=50;

Types of Variables

There are three types of variables in Java:

1) **Local Variable** A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists. A local variable cannot be defined with "static" keyword.

2) **Instance Variable :**A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as static. It is called an instance variable because its value is instance-specific and is not shared among instances.

3) **Static variable :** A variable that is declared as static is called a static variable. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

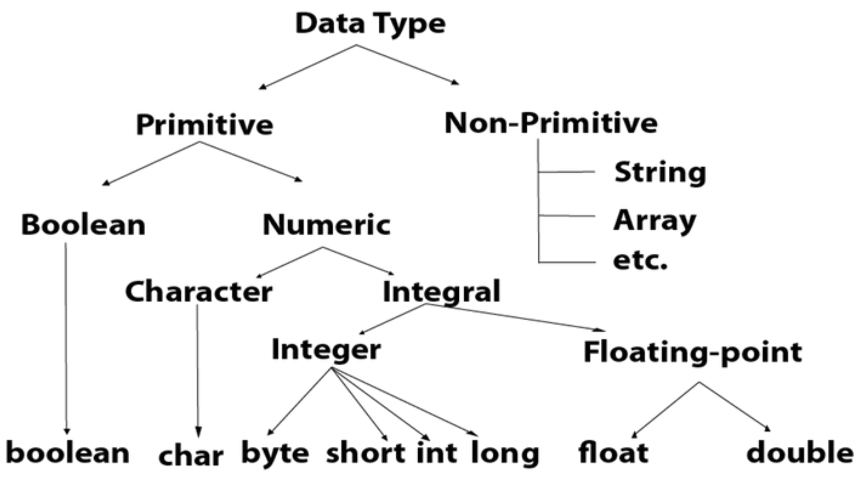
Data Types in Java

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

Primitive data types: The primitive data types include boolean, char, byte, short, int, long, float and double.

Non-primitive data types: The non-primitive data types include Classes, Interfaces, and Arrays.

Primitive data types:



Data Type	Default Value	Default size
boolean	FALSE	1 bit
char	\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Boolean Data Type: The Boolean data type is used to store only two possible values: true and false.
Example: Boolean one = false
Byte Data Type : The byte data type is a primitive data type. Its value-range lies between -128 to 127
Example: byte a = 10;
Short Data Type: The short data type is a primitive data type. Its value-range lies between -32,768 to 32,767
Example : short s = 100;
Int Data Type: The int data type is a primitive data type.
Example: int a = 100;
Long Data Type: The long data type is a primitive data type.
Example: long a = 1000L;

Float Data Type
float data type : The long data type is a primitive data types, its stores floating point values
Example: float f1 = 234.5f ;
Double Data Type: The Double data type is a primitive data type.
Example: double d1 = 12.3 ;
Char Data Type: The char data type is a single 16-bit Unicode character.The char data type is used to store characters.
Example : char letterA = 'A' ;

array: is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.
Types of Array in java
Single Dimensional Array :
Syntax

```
dataType[] arr; (or)
dataType []arr; (or)
dataType arr[];
```

Instantiation of an Array in Java : arrayRefVar=new datatype[size]
Example : int a[]={33,3,4,5};

Example :

```
class Test{
public static void main(String args[]){
int arr[]={33,3,4,5};
for(int i:arr)
System.out.println(i);
}}
```

Out put

```
33
3
4
5
```

Multidimensional Array
Syntax

```
dataType[][] arrayRefVar; (or)
dataType [][]arrayRefVar; (or)
dataType arrayRefVar[][]; (or)
dataType []arrayRefVar[]
```

Example to instantiate Multidimensional Array in Java

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

Example

```
class Test{
public static void main(String args[]){
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
for(int i=0;i<3;i++){
 for(int j=0;j<3;j++){
   System.out.print(arr[i][j]+" ");
 }
 System.out.println();
}
}}
```
Out put
```
1 2 3
2 4 5
4 4 5
```
Unicode System: Unicode is a universal international standard character encoding that is capable of representing most of the world's written languages.

Why java uses Unicode System?

Before Unicode, there were many language standards:

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

Operators in Java

Operator in Java is a symbol that is used to perform operations. For example: +, -, *, / etc.
There are many types of operators in Java which are given below:

Java Operator Precedence

Operator Type	Category	Precedence
Unary	postfix	expr++ expr--
	prefix	++expr --expr +expr -expr ~ !
Arithmetic	multiplicative	* / %
	additive	+ -
Shift	shift	<< >> >>>
Relational	comparison	< > <= >= instanceof
	equality	!= = =
Bitwise	bitwise AND	&
	bitwise exclusive OR	^

	bitwise inclusive OR	|
Logical	logical AND	&&
	logical OR	||
Ternary	ternary	? :
Assignment	assignment	+= -= *= /= %= &= ^= |= <<= >>= >>>=

Java Control Statements

Decision Making statements
- if statements
- switch statement

Loop statements
- do while loop
- while loop
- for loop
- for-each loop

Jump statements
- break statement
- continue statement

If Statement: In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

Simple if statement: If the condition is True, then the block of statements is executed and if it is False, then the block of statements is ignored.

Syntax:

```
if(condition) {
statement 1; //executes when condition is true
}
```

Example

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

Out put
Age is greater than 18

if-else statement: If the condition is True, then the if block of statements is executed and if it is False, then the else block of statements is executed.
Syntax:

```
if(condition) {
statement 1; //executes when condition is true
}
else{
statement 2; //executes when condition is false
}
```

Example
```
public class Example {
public static void main(String[] args) {
      int number=13;
      if(number%2==0){
      System.out.println("even number");
   }else{
      System.out.println("odd number");
   }
} }
```
Out put
```
odd number
```
if-else-if ladder: The if-else-if statement contains the if-statement followed by multiple else-if statements.
Syntax:

```
if(condition 1) {
statement 1; //executes when condition 1 is true
}
else if(condition 2) {
statement 2; //executes when condition 2 is true
}
else {
statement 2; //executes when all the conditions are false
}
```
Example
```
public class Example {
public static void main(String[] args) {
   int marks=65;

   if(marks<50){
      System.out.println("fail");
   }
   else if(marks>=50 && marks<60){
        System.out.println("D grade");
    }
    else if(marks>=60 && marks<70){
        System.out.println("C grade");
    }
    else if(marks>=70 && marks<80){
        System.out.println("B grade");
    }
    else if(marks>=80 && marks<90){
        System.out.println("A grade");
    }else if(marks>=90 && marks<100){
        System.out.println("A+ grade");
    }else{
        System.out.println("Invalid!");
    }
}
}
```

Out put
C grade

Nested if-statement: In nested if-statements, the if statement can contain a if or if-else statement inside another if or else-if statement.
Syntax:

```
if(condition 1) {
statement 1; //executes when condition 1 is true
if(condition 2) {
statement 2; //executes when condition 2 is true
}
else{
statement 2; //executes when condition 2 is false
}
}
```

Example

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

Out put
You are eligible to donate blood

Switch Statement: In Java, Switch statements are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched.

Points to be noted about switch statement:

- The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java
- Cases cannot be duplicate
- Default statement is executed when any of the case doesn't match the value of expression. It is optional.
- Break statement terminates the switch block when the condition is satisfied. It is optional, if not used, next case is executed.

syntax :

```
switch (expression){
   case value 1:
    statement 1;
    break;
   .
   .
   case value N:
    statement N;
    break;
   default:
    default statement;  }
```

Example

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

Out put

```
20
```

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

The for loop: is used to execute a single statement or a block of statements repeatedly as long as the given condition is TRUE.

Syntax:

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

Example

```
public class Example {
public static void main(String[] args) {
       for(int i=1;i<=3;i++){
        System.out.println(i);
    }
}
}
```

Out put

```
1
2
3
```

Java Nested for Loop: If we have a for loop inside the another loop, it is known as nested for loop. The inner loop executes completely whenever outer loop executes.

Syntax:

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

Example

```
public class Pyramid {
public static void main(String[] args) {
for(int i=1;i<=3;i++){
for(int j=1;j<=i;j++){
        System.out.print("* ");
}
System.out.println();//new line
}
}
}
```

Out put

```
*
* *
* * *
```

Java for-each Loop: The for-each loop is used to traverse array or collection in Java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation.

Syntax:

```
for(data_type variable : array_name){
//code to be executed
}
```

Example

```
public class Example {
public static void main(String[] args) {
      int arr[]={12,23,44};
       for(int i:arr){
       System.out.println(i);
    }
}
}
```

Out put

```
12
23
44
```

While Loop: The Java while loop is used to iterate a part of the program repeatedly until the specified Boolean condition is true. If the number of iteration is not fixed, it is recommended to use the while loop.

Syntax:

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

Example

```
public class Example {
public static void main(String[] args) {
    int i=1;
    while(i<=3){
       System.out.println(i);
    i++;
    }
}
}
```

Out put

```
1
2
3
```

the do-while: check the condition at the end of loop body. The Java do-while loop is executed at least once because condition is checked after loop body.

Syntax:

```
do{
//code to be executed / loop body
//update statement
}while (condition);
```

Example

```
public class Example {
public static void main(String[] args) {
   int i=1;
   do{
      System.out.println(i);
   i++;
   }while(i<=3);
}
}
```

Out put

```
1
2
3
```

Break Statement: When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

Syntax:

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

Example

```
public class BreakExample {
public static void main(String[] args) {
      for(int i=1;i<=3;i++){
      if(i==2){
                  break;
      }
      System.out.println(i);
   }
}
}
```

Out put

```
1
2
```

Continue Statement : The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately. It can be used with for loop or while loop.

Syntax:

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

```
public class ContinueExample {
public static void main(String[] args) {
      for(int i=1;i<=3;i++){
       if(i==2){
                   continue;//it will skip the rest statement
       }
       System.out.println(i);
   }
}
}
```

Out put

```
1
3
```

Java Comments : The Java comments are the statements in a program that are not executed by the compiler and interpreter.
There are three types of comments in Java.

- **Single Line Comment:** //This is single line comment
- **Multi Line Comment**

```
/*
This
is
multi line
comment
*/
```

- **Documentation Comment**

```
/**
*
*We can use various tags to depict the parameter
*or heading or author name
*We can also use HTML tags
*
*/
```

class: The java class is a template of an object. The class defines the blueprint of an object.

Syntax

```
class <ClassName>{
   data members declaration;
   methods defination;
}
```

Creating an Object: In java, an object is an instance of a class. When an object of a class is created, the class is said to be instantiated.

Syntax
<ClassName> <objectName> = new <ClassName>( );

Example

```
class Student{
 int id;
 String name;
}
class TestStudent1{
 public static void main(String args[]){
  Student s1=new Student();
  System.out.println(s1.id);
  System.out.println(s1.name);
 }
}
```

Out put

```
0
Null
```

There are 3 ways to initialize object in Java.

By reference variable

Example

```
class Student{
 int id;
 String name;
}
class TestStudent2{
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name="mrec";
  System.out.println(s1.id+" "+s1.name);//printing members with a white space
 }
}
```

Out put

```
101
mrec
```

By method

Example

```
class Student{
 int rollno;
 String name;
 void insertRecord(int r, String n){
  rollno=r;
  name=n;
 }
 void displayInformation(){System.out.println(rollno+" "+name);}
}
class TestStudent4{
 public static void main(String args[]){
  Student s1=new Student();
  s1.insertRecord(111,"Karan");
  s1.displayInformation();
   }
}
```

Out put
111
Karan

By constructor

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

    public static void main(String args[]){

    Student s1 = new Student4(111,"Karan");
         s1.display();
      }
}
```

Out put
111
Karan

constructor : A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes.

Every time an object is created using the new() keyword, at least one constructor is called.

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:

- Default constructor (no-arg constructor)
- Parameterized constructor

Default constructor : A constructor that has no parameters is known as default the constructor.

Example
```
class Defaultconst{
Defaultconst( ){System.out.println("Default constructor");}
public static void main(String args[]){
Defaultconst dc=new Defaultconst(); } }
```
out put

```
Default constructor
```

Parameterized constructor : A constructor that has parameters is known as parameterized constructor.

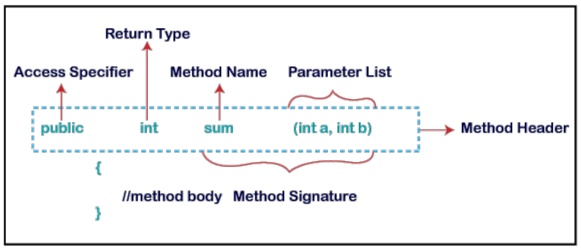
Example
```
class Student {
String name;
int id;
Student(String name, int id){
this.name = name;
this.id = id; } } class Test {
public static void main(String[] args){
Student std = new Student("avinash", 68);
System.out.println("Student Name :" + std.name + " and student Id :" + std.id); }}
```

Out put
```
Student Name: Avinash
and student Id : 68
```

A method
- A method is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a method.
- Methods are used to perform certain actions, and they are also known as functions.

Use of the methods To reuse code: define the code once, and use it many times.

Method Declaration



Types of Method
- Predefined Method
- User-defined Method

predefined methods are the method that is already defined in the Java class libraries is known as predefined methods ex length(), equals(), compareTo(), sqrt()

Example:

```
public class Demo {
public static void main(String[] args) {
System.out.print("The maximum number is: " + Math.max(9,7)); } }
```

Out put

```
9
```

- User-defined Method : The method written by the user or programmer is known as **a user-defined** method.

Example without parameters

```
public class Main {
static void myMethod() {
System.out.println("I just got executed!");}
 public static void main(String[] args) {
myMethod(); }. }
```

Out put

```
I just got executed!
```

Example with parameters

```
public class Main {
static void myMethod(String fname, int age) {
System.out.println(fname + " is " + age); }
public static void main(String[] args) {
myMethod("Jenny", 8); } }
```

Out put

```
Jenny is 18
```

These return types required a return statement at the end of the method. A return keyword is used for returning the resulted value.

Example

```
public class Main {
static int myMethod(int x, int y) {
return x + y; }
public static void main(String[] args) {
System.out.println(myMethod(5, 3)); }. }
```

Out put

```
8
```

Access Specifier: Access modifiers help to restrict the scope of a class, constructor, variable, method, or data member.

Types of Access Modifiers

- Default – No keyword required
- Private
- Protected
- Public

Private: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
Example

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");} }
public class Simple{
public static void main(String args[]){
A obj=new A();
System.out.println(obj.data);//Compile Time Error obj.msg();
} }
```

Out put

```
Compile Time Error obj.msg();
```

Default: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you don't use any modifier, it is treated as default by default.
Example

```
package pack;
class A{
void msg(){
System.out.println("Hello");} }

package mypack;
import pack.*;
class B{
public static void main(String args[ ]){
A obj = new A();//Compile Time Error obj.msg();
} }
```

Protected: The access level of a protected modifier is within the package and outside the package through child class.
Example

```
package pack;
public class A{
protected void msg(){
System.out.println(“Hello");
} }
```

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

Public: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Example
```
package pack;
public class A{
public void msg(){
System.out.println("Hello");} }
```

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

Out put
```
Hello
```

this:this is a keyword which is used to refer current object of a class. we can it to refer any member of the class. It means we can access any instance variable and method by using this keyword.
Example
```
class Demo{
int a=10;
void display( )
{
int a=200;
System.out.println("LOCAL VCARIABLE="+a);
System.out.println("INSTANCE VARIABLE="this.a);
}
public static void main(String args[]){
Demo obj=new Dem( );
obj.display( );
}
```
Out put
```
200
10
```

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

- To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

Advantage of Garbage Collection

- It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.
- It is automatically done by the garbage collector(a part of JVM) so we don't need to make extra efforts.

How can an object be unreferenced?

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

By nulling a reference:

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

By assigning a reference to another:

```
Employee e1=new Employee();
Employee e2=new Employee();
e1=e2;
```

By anonymous object:

```
new Employee();
```

Example

```
public class Student{
String name = “mahi”;
public void greet(){System.out.println(“hi:"+name);}
public void attend(){
System.out.println(“student attended");}
public static void main(String args[]){
new Student().greet();
new Student().attend() }
}
```

Out put

```
hi mahi
student attended
```

finalize() method

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

```
protected void finalize(){ }
```

gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

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

Out put

```
object is garbage collected
object is garbage collected
```

Method Overloading.

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

Different ways to overload the method

- By changing number of arguments
- By changing the data type

Method Overloading: changing no. of arguments

Example

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

Out put

```
22
33
```

Method Overloading: changing data type of arguments

```
class Adder{
static int add(int a, int b){
return a+b;}
static double add(double a, double b){
return a+b;} }
class TestOverloading2{
public static void main(String[] args){
```

```
System.out.println(Adder.add(11,11));
 System.out.println(Adder.add(12.3,12.6));
}}
```

Out put

```
22
24.9
```

Overloading constructor

Example

```
public class Student {
int id;
String name;
Student(){
 System.out.println("this a default constructor"); }
Student(int i, String n){
id = i;
name = n;
System.out.println("this a Parameterized Constructor"); }
public static void main(String[] args) {
Student s = new Student();
Student student = new Student(10, "mahi");
System.out.println("Student Id : "+student.id + "\nStudent Name : "+student.name); } }
```

Out put

```
this a default constructor
this a Parameterized Constructor
Student Id : 10
Student Name: mahi
```

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object.

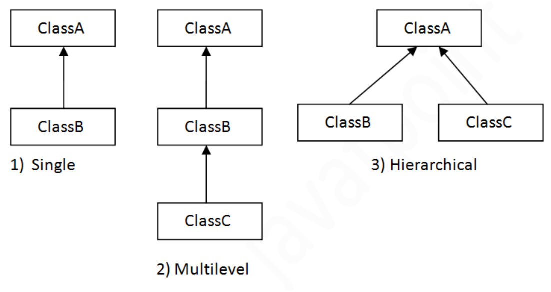
Why use inheritance in java

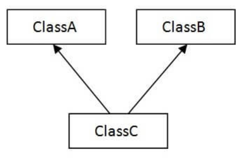
- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

The syntax of Java Inheritance

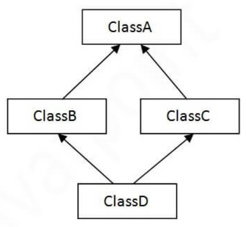
```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

Types of inheritance in java





4) Multiple



5) Hybrid

Single Inheritance Example

When a class inherits another class, it is known as a single inheritance.

```
class Animal{
}
class Dog extends Animal{
}
```

Multilevel Inheritance Example

When there is a chain of inheritance, it is known as multilevel inheritance. 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();
}}
```

Out put

```
eating…
barking…
weeping..
```

Hierarchical Inheritance Example

When two or more classes inherits a single class, it is known as hierarchical inheritance. class Animal{

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

```
System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){
System.out.println("meowing...");} }
class TestInheritance{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
c.bark();//C.T.Error
}}
```

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

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

Overriding
If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in Java.

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

Rules for Java Method Overriding

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

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

```
class Galaxy{
void camera(){System.out.println(“12 MP ");}
}
```

```
class Edge extends Galaxy{
void camera(){
System.out.println(“ 20 MP “);}
public static void main(String args[]){
 Edge obj = new Edge();
obj.camera();
}
}
```

Out put

```
20 MP
```

Rules for method overriding

Final methods can not be overridden
If we don’t want a method to be overridden, we declare it as final. Please see Using Final with Inheritance.

```
class Parent {
final void show() {}
}
class Child extends Parent {
void show() {}
}
```

Static methods can not be overridden
Static Method Also known as class level method and it is declared using a static keyword, its copy is shared by all the objects of a class.

Private methods can not be overridden
Private methods in Java are not visible to any other class

The overriding method must have the same return type (or subtype)
From Java 5.0 onwards it is possible to have different return types for an overriding method in the child class, but the child’s return type should be a sub-type of the parent’s return type
Example

```
class Parent {
public Object method(){
System.out.println("This is the method in parent");
return new Object(); }}
 class Child extends Parent {
public String method(){
 System.out.println(
"This is the method in Child ");
return "Hello, World!";}}
public class Test {
public static void main(String[] args){
SuperClass obj1 = new SuperClass(); obj1.method();
SubClass obj2 = new SubClass(); obj2.method(); }}
```

Overriding and Access Modifiers
The access modifier for an overriding method can allow more, but not less, access than the overridden method.
Example
```
class Parent {
private void m1(){
System.out.println("From parent m1()”);}}
class Child extends Parent {
public void m1(){
System.out.println("From child m1()");}}
class Main {
public static void main(String[] args){
Child obj = new Child();
obj.m1();}}
```

Out put
```
From child m1()
```

polymorphism
The word polymorphism means having many forms.
- Compile-time Polymorphism/ static /Early Binding
- Runtime Polymorphism/Dynamic Binding/Late Binding

Compile-time Polymorphism/ static /Early Binding
When type of the object is determined at compiled time(by the compiler), it is known as static binding. Ex overloading
If there is any private, final or static method in a class, there is static binding.

Runtime Polymorphism/Dynamic Binding/Late Binding
When type of the object is determined at run-time, it is known as dynamic binding. Ex overriding
Example
```
class Animal{
void eat(){
System.out.println("animal is eating...");}
}
class Dog extends Animal{
void eat(){
System.out.println("dog is eating...");}
public static void main(String args[]){
Animal b1= new Animal();
b1.eat();//animal is eating...
Dog b2= new Dog();
b2.eat();//dog is eating...
Animal a=new Dog();
 a.eat(); // dog is eating...
} }
```
Out put
```
animal is eating...
```

```
dog is eating...
dog is eating...
```

Recursion in Java
Recursion in java is a process in which a method calls itself continuously.
Syntax:

```
returntype methodname(){
methodname();//calling same method
}
```

Example

```
public class Example {
   static int factorial(int n){
         if (n == 1)
           return 1;
         else
           return(n * factorial(n-1));
   }

public static void main(String[] args) {
System.out.println("Factorial of 3 is: "+factorial(3));
}
}
```

Out put

```
Factorial of 3 is: 6
```

Working of above program:

```
     factorial(3)
         factorial(2)
            factorial(1)
               return 1
            return 2*1 = 2
         return 3*2 = 6
```

Parameter passing(Call by Value)
There is only call by value in java, not call by reference. If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method.

```
class Operation{
 int data=50;

 void change(int data){
 data=data+100;//changes will be in the local variable only
 }

 public static void main(String args[]){
   Operation op=new Operation();
   System.out.println("before change "+op.data);
   op.change(500);
   System.out.println("after change "+op.data);   }  }
```
Out put
```
before change 50
after change 50
```

Example of call by value
In case of call by reference original value is changed if we made changes in the called method. If we pass object in place of any primitive value, original value will be changed.

```
class Operation2{
 int data=50;

 void change(Operation2 op){
 op.data=op.data+100;//changes will be in the instance variable
 }


 public static void main(String args[]){
   Operation2 op=new Operation2();

   System.out.println("before change "+op.data);
   op.change(op);//passing object
   System.out.println("after change "+op.data);

 }
}
```
Out put
```
before change 50
after change 150
```

nested and inner classes
In Java, it is possible to define a class within another class, such classes are known as nested classes.
Syntax:
```
class OuterClass
{
...
    class NestedClass
    {
        ...
    }
}
```

Example
```
class OuterClass {
```

```
class InnerClass {
void display(){
System.out.println("inner class ");}}
public class Demo {
public static void main(String[] args)
{
OuterClass o= new OuterClass( );
OuterClass.InnerClass i= o.new InnerClass( )
// or OuterClass.InnerClass i= new OuterClass. new InnerClass( );
// or  new OuterClass. new InnerClass( ) . display();
i.display();
}
}
```

Out put

```
inner class
```

Example

```
class OuterClass {
class InnerClass {
public void display(){
System.out.println("inner class ");}}
public void view( ){
InnerClass i=new InnerClass( )
i.display();
}
public class Demo {
public static void main(String[] args)
{
OuterClass o= new OuterClass( );
i.view();
}
}
```

Out put

```
inner class
```

string: 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.

Creating a String
There are two ways to create string in Java:

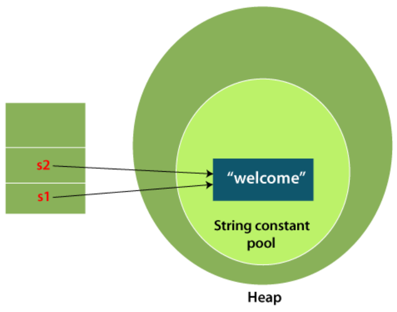
1. String literal
String s = "mrec";

2. Using new keyword
String s = new String ("mrec");

- 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 instance
```



string constant pool: String objects are stored in a special memory area known as the "string constant pool”

Example :

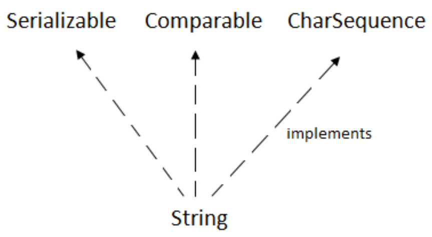
```
public class StringExample{
public static void main(String args[]){
String s1="java";//creating string by Java string literal
char ch[]={'s','t','r','i','n','g','s'};
String s2=new String(ch);//converting char array to string
String s3=new String("example");//creating Java string by new keyword
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}}
```

Out put

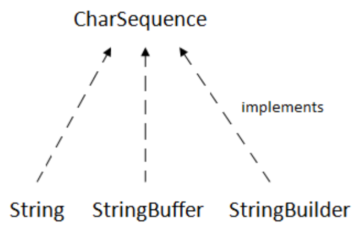
```
java
strings
example
```

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.



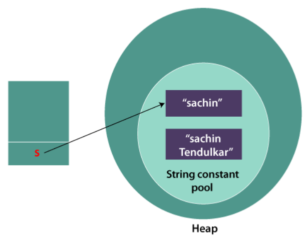
CharSequence Interface
The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in Java by using these three classes.



Immutable String: String references are used to store various attributes like username, password, etc. In Java, String objects are immutable. Immutable simply means unmodifiable or unchangeable.
Example

```
class Test{
 public static void main(String args[]){
   String s="Sachin";
   s.concat(" Tendulkar");//concat() method appends the string at the end
   System.out.println(s);//will print Sachin because strings are immutable objects
 }
}
```

Out put
Sachin



As you can see in the above figure that two objects are created but s reference variable still refers to "Sachin" not to "Sachin Tendulkar".
But if we explicitly assign it to the reference variable, it will refer to "Sachin Tendulkar" object.

Example

```
class Test{
 public static void main(String args[]){
   String s="Sachin";
   s=s.concat(" Tendulkar");//concat() method appends the string at the end
   System.out.println(s);//will print Sachin because strings are immutable objects   }  }
```

Out put
Sachin Tendulkar

String trim() method

The String class trim() method eliminates white spaces before and after the String.
Example
String s=" Sachin ";
s.trim();

String charAt() Method
The String class charAt() method returns a character at specified index.
Example
String s="Sachin";
s.charAt(0);//S

String length() Method
The String class length() method returns length of the specified String.
Example
String s="Sachin";
s.length();//6

String valueOf() Method

The String class valueOf() method coverts given type such as int, long, float, double, boolean, char and char array into String.

Example
int a=10;
String s=String.valueOf(a);
System.out.println(s+10); // 1010

String replace() Method
The String class replace() method replaces all occurrence of first sequence of character with second sequence of character.
Example
String s1="Java is a programming language.";
String replaceString=s1.replace("Java","C ");//replaces all occurrences of "Java" to "C "

String startsWith() and endsWith() method
The method startsWith() checks whether the String starts with the letters passed as arguments and endsWith() method checks whether the String ends with the letters passed as arguments.

Example
String s="Sachin";
s.startsWith("Sa");//true
s.endsWith("n");//true

String toUpperCase() and toLowerCase() method
The Java String toUpperCase() method converts this String into uppercase letter and String toLowerCase() method into lowercase letter.
Example
String s="Sachin";

```
s.toUpperCase();
s.toLowerCase();
```

Example

```
public class Test{
public static void main(String args[])
{
String s="Sachin";
int a=10;
System.out.println(s.toUpperCase());//SACHIN
System.out.println(s.toLowerCase());//sachin
System.out.println(s.startsWith("Sa"));//true
System.out.println(s.endsWith("n"));//true
System.out.println(s.charAt(0));//S
System.out.println(s.charAt(3));//h
System.out.println(s.length());//6
s=String.valueOf(a);
System.out.println(s+10);//1010
String s1=s.replace("10"," Sachin ");
System.out.println(s1);// Sachin
System.out.println(s1.trim());//Sachin
}
}
```

Exception Handling: in Java is one of the effective means to handle runtime errors so that the regular flow of the application can be preserved. Java Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

What are Java Exceptions?

In Java, Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions. Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object. It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.

Major reasons why an exception Occurs

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out-of-disk memory)
- Code errors
- Opening an unavailable file

Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc.

Difference between Error and Exception

Error: An Error indicates a serious problem that a reasonable application should not try to catch.

Exception: Exception indicates conditions that a reasonable application might try to catch

UNIT II

Substitutability or Liskov Substitution Principle (LSP) : in Java refers to the principle that a derived class (subclass) should be able to substitute its base class (superclass) without affecting the correctness of the program. This is a fundamental concept in object-oriented programming and is often associated with the Liskov Substitution Principle (LSP).

Example

```
public class Shape {
   public void draw() {
      System.out.println("Drawing a shape.");    }.  }
```

Now, you want to create specific shape classes (subclasses) like Circle and Rectangle that inherit from the Shape class and override the draw method to provide their own implementations:

```
public class Circle extends Shape {
   public void draw() {
      System.out.println("Drawing a circle.");    }  }

public class Rectangle extends Shape {
    public void draw() {
      System.out.println("Drawing a rectangle.");    }  }
```

In this scenario, substitutability means that you can use objects of the derived classes (Circle and Rectangle) wherever you use objects of the base class (Shape) without causing issues. For example:

```
public class GraphicsApp {
   public static void main(String[] args) {
      Shape shape1 = new Circle();
      Shape shape2 = new Rectangle();
      shape1.draw();
     shape2.draw();   }   }
```

The ability to substitute Circle and Rectangle objects for Shape objects without breaking the program demonstrates substitutability. It allows you to work with different shapes interchangeably, making your code more flexible and extensible.

Example

```
class Vehicle  {
   public int Weels() {
      return 2;   }
public Boolean Engine() {
      return true;    } }
class Car extends Vehicle{
   public int Weels() {
      return 4;  }  }
class Bicycle extends Vehicle {
   public int Weels() {
        return 2;    }
public Boolean Engine() {
        return null;    } }
public class Test {
public static void main (String args[]){
Vehicle myVehicle = new Bicycle();
System.out.println(myVehicle.Engine().toString());} }
```

Out put

```
Exception in thread "main" java.lang.NullPointerException
            at Test.main(Test.java:22)
```

To salve able problem here

Example

```
class Vehicle  {
    public int Weels() {
        return 2;    } }
class EngineVehicle extends Vehicle{
public Boolean Engine() {
        return true;    }}
class Car extends EngineVehicle{
    public int Weels() {
        return 4;  }  }
class Bicycle extends Vehicle {
    public int Weels() {
        return 2;    } }
public class Test {
public static void main (String args[]){
Vehicle myVehicle = new Bicycle();
System.out.println(myVehicle.Weels());} }
```

Out put

```
2
```

Forms of inheritance:

One of the main purposes is substitutability. The substitutability means that when a child class acquires properties from its parent class, the object of the parent class may be substituted with the child class object.

For example, if B is a child class of A, anywhere we expect an instance of A we can use an instance of B.

The substitutability can achieve using inheritance, whether using extends or implements keywords.

The following are the different forms of inheritance in java.

Specialization: Converting a super class type into a sub class type is called 'Specialization'. It holds the principle of substitutability.

Specification: the parent class just specifies which methods should be available to the child class but doesn't implement them. The java provides concepts like abstract and interfaces to support this form of inheritance. It holds the principle of substitutability.
Construction: where the child class may change the behavior defined by the parent class (overriding). It does not hold the principle of substitutability.
Eextension: where the child class may add its new properties. It holds the principle of substitutability.
Limitation: where the subclass restricts the inherited behavior. It does not hold the principle of substitutability.
Combination: where the subclass inherits properties from multiple parent classes. Java does not support multiple inheritance type.

Benefits of Inheritance

- Inheritance helps in code reuse. The child class may use the code defined in the parent class without re-writing it.
- Inheritance can save time and effort as the main code need not be written again.
- Inheritance provides a clear model structure which is easy to understand.
- An inheritance leads to less development and maintenance costs.

Costs of Inheritance

- Inheritance decreases the execution speed due to the increased time and effort it takes, the program to jump through all the levels of overloaded classes.
- Inheritance makes the two classes (base and inherited class) get tightly coupled. This means one cannot be used independently of each other.
- The changes made in the parent class will affect the behavior of child class too.
- The overuse of inheritance makes the program more complex.

Subclass

- A class which inherits the characteristics and behavior of another class is called a subclass.
- In Java, a subclass is defined using the extends keyword, which indicates that it is inheriting from a superclass.
- A subclass can add new fields and methods, override methods from the superclass, and extend the functionality of the superclass.

Example of defining a subclass in Java:

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

Subtype

- A subtype is a concept related to the type system and polymorphism. It is used to indicate that an instance of a derived class (subclass) can be treated as an instance of its base class (superclass) while preserving the program's correctness.
- Subtyping is fundamental to the concept of polymorphism

Example

```
Superclass obj = new Subclass(); // Subtype relationship
```

In the example above, Subclass is a subclass of Superclass, and objects of Subclass can be treated as objects of Superclass due to the subtype relationship. This allows for flexibility and polymorphic behavior in your code, which is a fundamental aspect of object-oriented programming.

Base class object: you can create an object of a base (superclass) class just like you would create an object of any other class. When you create an object of a base class, you can access its own fields and methods, but you cannot access the fields and methods that are specific to its subclasses. However, if you assign an object of a subclass to a reference variable of the base class type, you can access the fields and methods of the base class and any overridden methods of the subclass. This is known as polymorphism.

Example

```
class Vehicle {
   public void start() {
      System.out.println("The vehicle is starting.");    }
class Car extends Vehicle {
public void start() {
      System.out.println("Car is starting.”);  }
public class Test{
public static void main(String[] args) {
Vehicle myVehicle = new Car("Toyota");
myVehicle.start();       }.  }
```

Out put

```
Car is starting.
```

Hierarchical abstractions:

- Refer to the use of inheritance and class hierarchies to create a structured and organized system of classes and objects.
- Hierarchical abstractions in Java are mainly achieved through the use of classes and interfaces.
- Hierarchical abstractions in Java help you create a well-structured and organized codebase, allowing you to reuse code, create relationships between classes, and model complex systems with clear and maintainable code.

Super keyword: The super keyword in Java is a reference variable that is used to refer to the immediate parent class’s object.

Usage of Java super Keyword

- super can be used to refer immediate parent class instance variable.
- super can be used to invoke immediate parent class method.
- super() can be used to invoke immediate parent class constructor.

Example

```
class Animal{
String color="white";  }
class Dog extends Animal{
String color="black";
void color(){
System.out.println(super.color);}  }
class Test{
public static void main(String args[]){
```

```
Dog d=new Dog();
d.color();  }}
```
Out put

```
White
```
Example

```
class Animal{
void eat(){System.out.println("eating...");}  }
class Dog extends Animal{
void eat(){
super.eat();
System.out.println("dog eating...");}    }  }
class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
d.eat();  }}
```
Out put

```
eating…
Dog eating…
```

Example

```
class Animal{
Animal(){System.out.println("animal is created");}  }
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");  }  }
class Test{
public static void main(String args[]){
Dog d=new Dog();  }}
```
Out put

```
animal is created
dog is created
```

Final Keyword

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

- variable
- method
- class

Example

```
class Test {
  public static void main (String[] args) {
      final int count = 10;
     count = 15;  } }
```

Example

```
class Bik{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;   }
 public static void main(String args[]){
 Bike obj=new  Bike();
 obj.run();  }
```
Out put
```
Output:Compile Time Error
```

Example
```
class Bike{
  final void run(){System.out.println("running");}   }
class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}
   public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();    }  }
```

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

Abstract class:

A class which is declared as abstract is known as an abstract class. It can have abstract and non-abstract methods. It needs to be extended and its method implemented.

Syntax
```
abstract class class_name {
//abstract or non-abstract methods    }
```

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

Abstract Method: A method declared using the abstract keyword within an abstract class and does not have a definition (implementation) is called an abstract method.

Syntax:
```
abstract return_type method_name( [ argument-list ] );
```
Example

```
abstract class Bike{
  abstract void run();  }
```

An interface: in Java is a blueprint of a class. It has static constants and abstract methods.

use of interface

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

Syntax:

```
interface <interface_name>{
   // declare constant fields
   // declare methods that abstract
   // by default.  }
```

Example

```
interface Printable{
void print();  }
interface Showable{
void print();  }
class Test implements Printable, Showable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
TestInterface3 obj = new TestInterface3();
obj.print();   }  }
```

Output

```
Hello
```

Variables in Interfaces

- The variable in an interface is public, static, and final by default.
- If any variable in an interface is defined without public, static, and final keywords then, the compiler automatically adds the same.
- No access modifier is allowed except the public for interface variables.
- Every variable of an interface must be initialized in the interface itself.
- The class that implements an interface can not modify the interface variable, but it may use as it defined in the interface.

Difference between Class and Interface :

Class	Interface
The keyword used to create a class is "class"	The keyword used to create an interface is "interface"
Objects can be created.	Objects cannot be created.
Classes do not support multiple inheritance.	The interface supports multiple inheritance.

It can be inherited by another class using the keyword ‘extends’.	It can be inherited by a class by using the keyword ‘implements’ and it can be inherited by an interface using the keyword ‘extends’.
It can contain constructors.	It cannot contain constructors.
It cannot contain abstract methods.	It contains abstract methods only.
Variables in a class can be static, final, or neither.	All variables are static and final.

Package

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

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.

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

How to compile java package

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

```
javac -d directory javafilename
```

Example

```
javac -d . Simple.java
```

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

How to run java package program

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

```
To Compile: javac -d . Simple.java
To Run: java mypack.Simple
```

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

How to access package from another package?
There are three ways to access the package from outside the package.
import package.*;
import package.classname;
fully qualified name.

Using packagename.*

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

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

//save by B.java
package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

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

Example of package by import package.classname

```
//save by A.java

package pack;
```

```
public class A{
  public void msg(){System.out.println("Hello");}
}

//save by B.java
package mypack;
import pack.A;

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

Using fully qualified name

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

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

Example of package by import fully qualified name

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

//save by B.java
package mypack;
class B{
  public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
}
```

Member access rules

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

Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Java User Input or Scanner class : is used to get user input, and it is found in the java.util package.
To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation.

Input Types

Method	**Description**
nextBoolean()	Reads a boolean value from the user
nextByte()	Reads a byte value from the user
nextDouble()	Reads a double value from the user
nextFloat()	Reads a float value from the user
nextInt()	Reads a int value from the user
nextLine()	Reads a String value from the user
nextLong()	Reads a long value from the user
nextShort()	Reads a short value from the user
nextLine()	Reads a Strings from the user

Example

```
import java.util.Scanner;
class Test {
public static void main(String[] args) {
String userName;
Scanner myObj = new Scanner(System.in);  // Create a Scanner object
System.out.println("Enter username");

   userName = myObj.nextLine();  // Read user input
   System.out.println("Username is: " + userName);  // Output user input  } }
```

Out put

```
Enter username
```

Mrec
Username is: Mrec

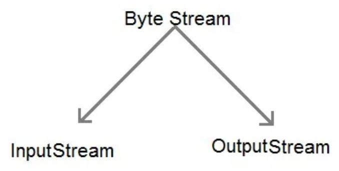
Java IO Stream
Java performs I/O through Streams. A Stream is linked to a physical layer by java I/O system to make input and output operation in java. In general, a stream means continuous flow of data.

Java encapsulates Stream under java.io package. Java defines two types of streams. They are,

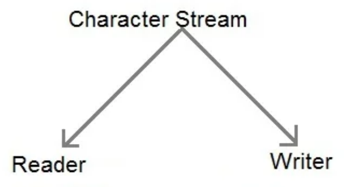
Byte Stream : It provides a convenient means for handling input and output of byte.

Character Stream : It provides a convenient means for handling input and output of characters. Character stream uses Unicode and therefore can be internationalized.

Java Byte Stream Classes: Byte stream is defined by using two abstract class at the top of hierarchy, they are InputStream and OutputStream.



Java Character Stream Classes: Character stream is also defined by using two abstract class at the top of hierarchy, they are Reader and Writer.



Java.io.File Class
Java File class is Java’s representation of a file or directory pathname.

How to Create a File Object?
A File object is created by passing in a string that represents the name of a file, a String, or another File object.
Example,

File a = new File(“hello.txt”);
Constructors of Java File Class

1)File f = new File(String name);
Creates a Java File Object to Represent Name of specified File OR Directory Present in Current Working Directory.

Example
File f= new File("abc.txt");

2) File f = new File(String subdir, String name);
Creates a Java File Object to Represent Name of the File OR Directory Present in specific Directory.
Example
File f1 = new File("mrec", " demo.txt"); OR
File f1 = new File(f, "demo.txt");

3) File f = new File(File subdir, String name)
Write a Code to Create a File Named with abc.txt in Current Working Directory
Example
File f= new File(“E:\\xyz", “demo.txt");

Example
```
import java.io.*;
public class Test {
public static void main(String args[]) throws Exception{
File f = new File("text.txt");
if(f.createNewFile())
System.out.println("File created");
else
System.out.println("File already exists");  } }
```

Out put:
```
File created
```

The File class has many useful methods for creating and getting information about files. For example:

Method	**Type**	**Description**
canRead()	Boolean	Tests whether the file is readable or not
canWrite()	Boolean	Tests whether the file is writable or not
createNewFile()	Boolean	Creates an empty file
delete()	Boolean	Deletes a file
exists()	Boolean	Tests whether the file exists
getName()	String	Returns the name of the file
length()	Long	Returns the size of the file in bytes

list()	String[]	Returns an array of the files in the directory
mkdir()	Boolean	Creates a directory

FileWriter Class
Java FileWriter class of java.io package is used to write data in character form to file.

FileWriter Class constructors
FileWriter(File file): It constructs a FileWriter object given a File object.
Example
FileWriter fw = new FileWriter(File file);

FileWriter(File file, boolean append): It constructs a FileWriter object given a File object. If the second argument is true, then bytes will be written to the end of the file, if the file exists.

Example
FileWriter fw = new FileWriter(File file, boolean append);

Methods of FileWriter Class
write(int a): This method writes a single character specified by int a.
write(char ch[]): This method writes an array of characters specified by ch[ ].
write(String st): This method writes a string value specified by 'st' into the file.

Example
```
import java.io.FileWriter;
import java.io.IOException;
public class Test {
public static void main(String[] args) throws Exception {
FileWriter myWriter = new FileWriter("text.txt");
myWriter.write("Welcome to mrec");
myWriter.close();
System.out.println("Successfully wrote to the file."); }}
```

Out put
```
Welcome to mrec
```

BufferedWriter Class
Java BufferedWriter class is used to provide buffering for Writer instances. It makes the performance fast.

BufferedWriter Class constructors
BufferedWriter(Writer wrt): It is used to create a buffered character output stream that uses the default size for an output buffer.

BufferedWriter(Writer wrt, int size): It is used to create a buffered character output stream that uses the specified size for an output buffer.

Methods of BufferedWriter
void newLine(): It is used to add a new line by writing a line separator.
void flush(): It is used to flushes the input stream.
void close(): It is used to closes the input stream

Example
```
import java.io.*;
public class BufferedWriterExample {
public static void main(String[] args) throws Exception {
   FileWriter writer = new FileWriter("text.txt");
   BufferedWriter buffer = new BufferedWriter(writer);
   buffer.write("Welcome to mrec.");
   buffer.close();
   System.out.println("Success");     }  }
```

Out put
```
Welcome to mrec
```

FileReader class
FileReader in Java is a class in the java.io package which can be used to read a stream of characters from the files.

FileReader class constructors
FileReader(String filename) : Creates a new FileReader with a a given FileName to read
Example
```
FileReader fileReader = new FileReader("abc.txt");
```

FileReader(File f): Creates a new FileReader with the the given File to read

Example
```
FileReader fileReader = new FileReader(fileReader );
```

Methods of Java FileReader Class

Int read() : The read() method reads and passes a single character or -1 if the stream is ended.
Int read(char[] ch): It reads a stream of characters and stores them in the given Character array.
close(): It closes the stream and releases the system resources associated with it.

Example
```
import java.io.FileReader;
public class FileReaderExample {
   public static void main(String args[])throws Exception{
        FileReader fr=new FileReader("test.txt");
        int i;
        while((i=fr.read())!=-1)
        System.out.print((char)i);
        fr.close();       }    }
```

Out put

```
Welcome to mrec
```

BufferedReader Class

Java BufferedReader class is used to read the text from a character-based input stream. It can be used to read data line by line by readLine() method. It makes the performance fast.

BufferedReader Class constructors

BufferedReader(Reader rd): It is used to create a buffered character input stream that uses the default size for an input buffer.

BufferedReader(Reader rd, int size): It is used to create a buffered character input stream that uses the specified size for an input buffer.

Methods of Java BufferedReader Class

String readLine(): It is used for reading a line of text.

void close(): It closes the input stream and releases any of the system resources associated with the stream.

Example

```
import java.io.*;
public class Test{
public static void main(String args[])throws Exception{
  FileReader fr=new FileReader("text.txt");
    BufferedReader br=new BufferedReader(fr);
    String name=br.readLine();
    System.out.println("Welcome "+name);
}
}
```

Out put

```
Welcome mrec
```

Delete file

To delete a file in Java, use the delete() method:

Example

```
import java.io.File;  // Import the File class
public class Test{
  public static void main(String[] args) {
    File myObj = new File("text.txt");
    if (myObj.delete())
      System.out.println("Deleted the file: " + myObj.getName());
     else
      System.out.println("Failed to delete the file.”); } }
```

Out put

```
Failed to delete the file.
```

III Module

Exception Handling

The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

What is Exception in Java?

Dictionary Meaning: Exception is an abnormal condition, an exception is an event that disrupts the normal flow of the program

What is Exception Handling?

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

Advantage of Exception Handling

The core advantage of exception handling is to maintain the normal flow of the application.

Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception

- Checked Exception
- Unchecked Exception
- Error

Checked Exceptions: Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler. Examples IOException FileNotFoundException, ClassNotFoundException, SQLException

Unchecked Exceptions: Unchecked exceptions are called run-time exceptions because these exceptions are checked at run-time by the compiler. Examples ArithmeticException, NullPointerException, NumberFormatException, ArrayIndexOutOfBoundsException

Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

the exception handling mechanism uses five keywords namely try, catch, finally, throw, and throws.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.

throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions.

Common Scenarios of Java Exceptions

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

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

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

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

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

Syntax

```
try {
  //  Block of code to try
}
catch(Exception e) {
  //  Block of code to handle errors
}
```

Example

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

Out put

```
java.lang.ArithmeticException: / by zero
rest of the code
```

Java Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler.

Example

```
public class Test {

    public static void main(String[] args) {
           try{
                int a[]=new int[5];
                a[5]=30/0;    }
               catch(ArithmeticException e)  {
                   System.out.println("Arithmetic Exception occurs");     }
               catch(ArrayIndexOutOfBoundsException e)        {
                   System.out.println("ArrayIndexOutOfBounds Exception occurs");      }
               System.out.println("rest of the code");   } }
```

Out put

```
Arithmetic Exception occurs
rest of the code
```

Java Nested try block

try block inside another try block is permitted. It is called as nested try block.

```
public class Test{
public static void main(String args[]){
 try{
    try{
     System.out.println("going to divide by 0");
     int b =39/0;    }
     catch(ArithmeticException e)    {
      System.out.println(e);  }
    System.out.println("other statement");      }
  catch(Exception e)    {
    System.out.println("handled the exception (outer catch)");  }
  System.out.println("normal flow..");     }    }
```

Out put

```
going to divide by 0
java.lang.ArithmeticException: / by zero
other statement
normal flow..
```

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

Why use Java finally block?

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

Example

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

Out put

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

The throw keyword

The throw statement allows you to create a custom error.

Syntax

```
throw new exception_class("error message");
```

Example

```
public class Main {
  static void checkAge(int age) {
    if (age < 18) {
      throw new ArithmeticException("Access denied - You must be at least 18 years old.");   }
    else {
      System.out.println("Access granted - You are old enough!");  }  }
  public static void main(String[] args) {
    checkAge(15); // Set age to 15 (which is below 18...)  }. }
```

throws: with the help of the throws keyword, we can provide information to the caller of the method about the exception.

Syntax

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

Example

```
class Test {
    public static void main(String[] args) throws InterruptedException   {
        Thread.sleep(10000);
        System.out.println("Hello");} }
```

Out put

```
Hello
```

Multithreading: Multithreading in Java is a process of executing multiple threads simultaneously.

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)**

- Executing multiple tasks simultaneously where each task is a separate independent Process
- 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 loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

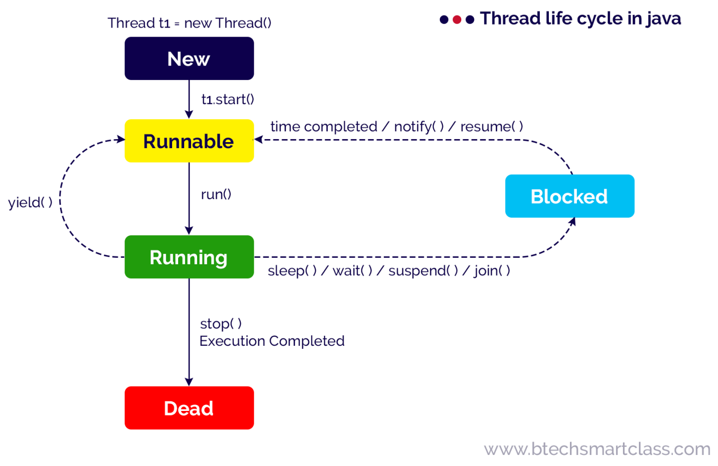
- Executing multiple tasks simultaneously where each task is a separate independent part of the same program
- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

What is Thread in java

A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Life cycle of a Thread (Thread States)

In Java, a thread always exists in any one of the following states. These states are:



New: When a thread object is created using new, then the thread is said to be in the New state. This state is also known as Born state.

Example

Thread t1 = new Thread();

Runnable / Ready: When a thread calls start( ) method, then the thread is said to be in the Runnable state. This state is also known as a Ready state.

Example

t1.start( );

Running When a thread calls run( ) method, then the thread is said to be Running. The run( ) method of a thread called automatically by the start( ) method.

Blocked / Waiting : A thread in the Running state may move into the blocked state due to various reasons like sleep( ) method called, wait( ) method called, suspend( ) method called, and join( ) method called, etc.

Dead / Terminated:A thread in the Running state may move into the dead state due to either its execution completed or the stop( ) method called. The dead state is also known as the terminated state.

How to create a thread in Java

There are two ways to create a thread:

- By extending Thread class
- By implementing Runnable interface.

extending Thread class

follow the step given below.

Step-1: create a class that extends Thread class.
Step-2: Override the run( ) method with the code
Step-3: Create the object of the newly created class in the main( ) method.
Step-4: Call the start( ) method on the object created in the above step.

Example

```
class Mythread extends Thread {
public void run(){
for(int i=0;i<3;i++)
System.out.println("child thread");}}
class Test {
public static void main(String args[]){
Mythread t1=new Mythread();
t1.start();
for(int i=0;i<3;i++)
System.out.println("main thread");}}
```

Output:

```
child thread
main thread
```

implementing Runnable interface

follow the step given below.

To create a thread using Runnable interface, follow the step given below.

Step-1: Create a class that implements Runnable interface.
Step-2: Override the run( ) method with the code
Step-3: Create the object of the newly created class in the main( ) method.
Step-4: Create the Thread class object by passing above created object as parameter to the Thread class constructor.
Step-5: Call the start( ) method on the Thread class object created in the above step

Example

```
class Multi implements Runnable{
public void run(){
for(int i=0;i<3;i++)
System.out.println("child thread"); }  }
class Test {
public static void main(String args[]){
Multi m1=new Multi();
Thread t1 =new Thread(m1);
t1.start();
for(int i=0;i<3;i++)
System.out.println("main thread");}}
```

Output:

```
child thread
main thread
```

Thread Scheduler in Java

- A component of Java that decides which thread to run or execute and which thread to wait is called a thread scheduler in Java.
- However, if there is more than one thread in the runnable state, the thread scheduler to pick one of the threads and ignore the other ones.
- There are some criteria that decide which thread will execute first. There are two factors for scheduling a thread i.e. Priority and Time of arrival.

Priority: Priority of each thread lies between 1 to 10. If a thread has a higher priority, it means that thread has got a better chance of getting picked up by the thread scheduler.

Time of Arrival: Suppose two threads of the same priority enter the runnable state, then priority cannot be the factor to pick a thread from these two threads. In such a case, arrival time of thread is considered by the thread scheduler. A thread that arrived first gets the preference over the other threads.

thread priorities:

- Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling). Thread class provides two methods setPriority(int), and getPriority( ) to handle thread priorities.
- The Thread class also contains three constants that are used to set the thread priority, and they are listed below.
- MAX_PRIORITY - It has the value 10 and indicates highest priority.
- NORM_PRIORITY - It has the value 5 and indicates normal priority.
- MIN_PRIORITY - It has the value 1 and indicates lowest priority.

setPriority( ) method: used to set the priority of a thread. It takes an integer range from 1 to 10 as an argument and returns nothing (void).

Example

```
threadObject.setPriority(4);
```

or

```
threadObject.setPriority(MAX_PRIORITY);
```

getPriority( ) method: used to access the priority of a thread. It does not takes anyargument and returns name of the thread as String.

Example

```
String threadName = threadObject.getPriority();
```

Example

```
class SampleThread extends Thread{
        public void run() {
                System.out.println("Current Thread: " + Thread.currentThread().getName());}}
public class Test {
        public static void main(String[] args) {
                SampleThread t1 = new SampleThread();
                SampleThread t2 = new SampleThread();
                t1.setName("first");
                t2.setName("second");
                t1.setPriority(4);
                t2.setPriority(Thread.MAX_PRIORITY);
                t1.start();
                t2.start();}}
```

Out put

```
Current Thread: second
Current Thread: first
```

Synchronization : is the capability to control the access of multiple threads to any shared resource.

```
class Table{
 synchronized void printTable(int n){//synchronized method
   for(int i=1;i<=3;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}    }     }   }
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;  }
public void run(){
t.printTable(5);  }   }
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;  }
public void run(){
t.printTable(10);  }  }
public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();  }  }
```

Out put

```
5
10
15
10
20
30
```

Synchronized Block in Java

- Synchronized block can be used to perform synchronization on any specific resource of the method.
- Suppose we have 50 lines of code in our method, but we want to synchronize only 5 lines, in such cases, we can use synchronized block.

Syntax

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

Inter-thread Communication: Inter-thread communication is all about allowing synchronized threads to communicate with each other.

It is implemented by following methods of Object class:

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

wait()

The 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.

notify()

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

Syntax:

```
public final void notify()
```

notifyAll()

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

Syntax:

```
public final void notifyAll()
```

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

Example

```
Thread.currentThread().getThreadGroup().getName()
```

Daemon Thread : Daemon thread in Java is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

Example

```
public class DaemonThread extends Thread{
 public void run( ){
for(int I=0;i<3;i++){
System.out.println("daemon thread work");
try{
thread.sleep(200);    }
catch( InterruptedException e){    }   }  }
class Test {
 public static void main(String[] args){
  DaemonThread t1=new TestDaemonThread();//creating thread
  t1.setDaemon(true);//now t1 is daemon thread
  t1.start();//starting threads
 System.out.println("end of main thread");    }  }
```

Out put

```
daemon thread work
user thread work
```

Java Enums : The Enum in Java is a data type which contains a fixed set of constants.
It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY) , According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.

Defining Java Enum

```
enum Season { WINTER, SPRING, SUMMER, FALL }
```

Example

```
class EnumExample1{
public enum Season { WINTER, SPRING, SUMMER }
public static void main(String[] args) {
for (Season s : Season.values())
System.out.println(s);
}}
```

Out put

```
WINTER
SPRING
SUMMER
```

Java Wrapper Classes

Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects.

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

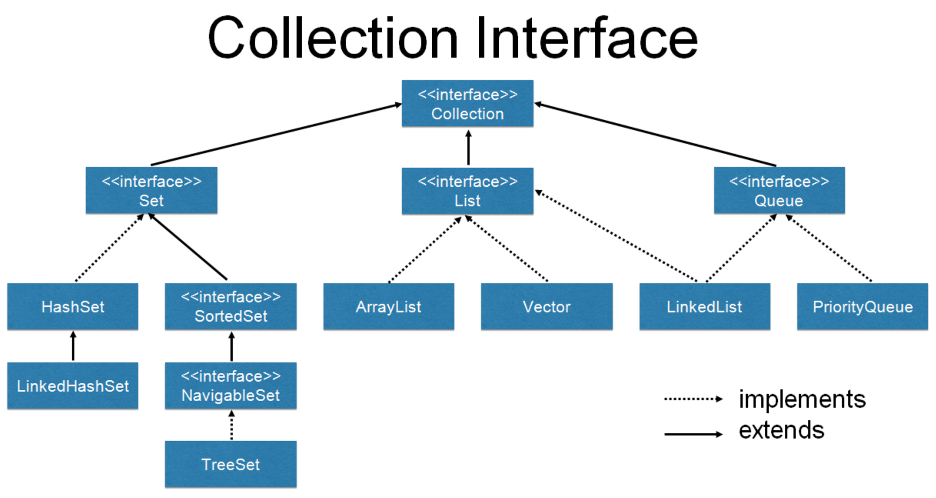
Autoboxing and Unboxing: The automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing and opposite operation is known as unboxing.

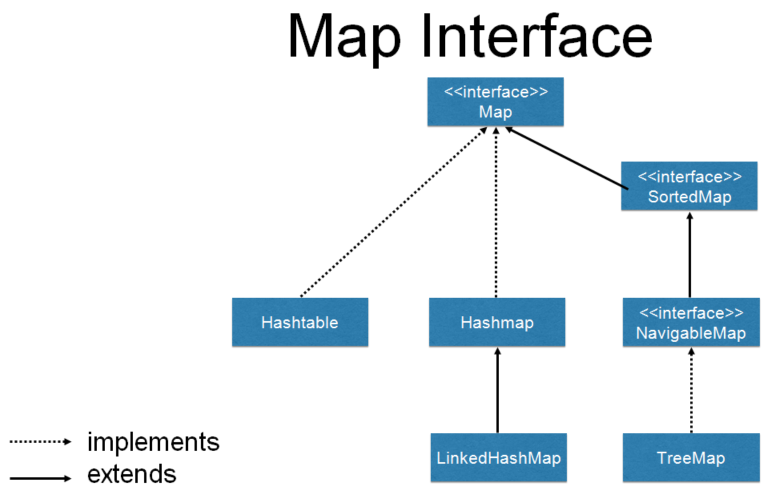
Autoboxing
Example
Integer I =10;
Compiler replace with this code
Integer I = Integer.valueof(10);

Unboxing:
Example
Integer I = new Integer(10);
int i= I;
Compiler replace with this code
int I=I.intValue();

Java Annotations Java Annotation is a tag that represents the metadata i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

@Override @Override annotation assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.





Collections in Java : Collection is an interface which can be used to represent a group of individual objects as a single entity.
Collections: is an utility class present in java.util. package to define several utility methods (like Sorting, Searching...) for Collection objects.

List Interface : List is child interface of Collection.
If we want to represent a group of individual objects as a single entity where duplicates are allowed and insertion order preserved then we should go for List.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.
To instantiate the List interface, we must use :

```
List <data-type> list1= new ArrayList();
List <data-type> list2 = new LinkedList();
List <data-type> list3 = new Vector();
List <data-type> list4 = new Stack();
```

LinkedList
LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized.

```
import java.util.*;
public class TestJavaCollection2{
public static void main(String args[]){
LinkedList<String> al=new LinkedList<String>();
al.add("Ravi");
al.add("Vijay");
al.add("Ravi");
al.add("Ajay");
Iterator<String> itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next());  }  }  }
```

Adding an element at the specific position : al.add(1, "Gaurav");
Adding second list elements to the first list : al.addAll(al2);
Adding an element at the first position : al.addFirst("Lokesh");
Adding an element at the last position :al.addLast(“Harsh");

Vector
Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.
Consider the following example.

```
import java.util.*;
public class TestJavaCollection3{
public static void main(String args[]){
Vector<String> v=new Vector<String>();
v.add("Ayush");
v.add("Amit");
v.add("Ashish");
v.add("Garima");
Iterator<String> itr=v.iterator();
while(itr.hasNext()){
System.out.println(itr.next());  }  }  }
```

Output:

```
Ayush
Amit
Ashish
Garima
```

Stack
The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.
Consider the following example.

Example

```
import java.util.*;
public class TestJavaCollection4{
public static void main(String args[]){
Stack<String> stack = new Stack<String>();
stack.push("Ayush");
stack.push("Garvit");
stack.push("Amit");
stack.push("Ashish");
stack.push("Garima");
stack.pop();
Iterator<String> itr=stack.iterator();
while(itr.hasNext()){
```

```
System.out.println(itr.next());  }  }  }
```

Out put

```
Ayush
Garvit
Amit
Ashish
Garima
```

set interface

- It is the child interface of Collection.
- If we want to represent a group of individual objects as a single entity where duplicates are not allowed and insertion order not preserved then we should go for Set.

SortedSet:

- It is the child interface of Set.
- If we want to represent a group of individual objects as a single entity where duplicates are not allowed but all objects should be inserted according to some sorting orde then we should go for SortedSet.

NavigableSet:

It is the child interface of SortedSet if defines several methods for navigation purposes.

Example

```
import java.util.*;
public class setExample{
   public static void main(String[] args)   {
      Set<String> data = new LinkedHashSet<String>();
      data.add("Ashish");
      data.add("Ayush");
      data.add("Garima");
      System.out.println(data);    }    }
```

Out put

```
[Ashish , Ayush ,Garima]
```

Operations on the Set Interface

On the Set, we can perform all the basic mathematical operations like intersection, union and difference.

Example

set1 = [22, 45, 33, 66, 55, 34, 77] and set2 = [33, 2, 83, 45, 3, 12, 55].

Intersection: The intersection operation returns all those elements which are present in both the set. The intersection of set1 and set2 will be [33, 45, 55].

Union: The union operation returns all the elements of set1 and set2 in a single set, and that set can either be set1 or set2. The union of set1 and set2 will be [2, 3, 12, 22, 33, 34, 45, 55, 66, 77, 83].

Difference: The difference operation deletes the values from the set which are present in another set. The difference of the set1 and set2 will be [66, 34, 22, 77].

Queue :

- It is child interface of Collection.
- If we want to represent a group of individual objects prior to processing then we should go for Queue.

Example

Before sending a mail all mail id's we have to store somewhere and in which order we saved in the same order mail's should be delivered (First in First out) for this requirement Queue concept is the best choice.

Java Map Interface

- Map is not the child interface of Collection.
- If we want to represent a group of individual objects as key value pairs then should go for Map.
- Both key and value are objects, duplicated keys are not allowed but values can be duplicated

SortedMap:

- It is the child interface of map.
- If we want to represent a group of key value pairs according to some sorting order of keys then we should go for SortedMap

NavigableMap:

It is the child interface of sorted map, it defines several utility methods for navigation purpose.

```
import java.util.*;
public class Test {
public static void main(String[] args) {
   Map map=new HashMap();
    map.put(1,"Amit");
   map.put(5,"Rahul");
   map.put(2,"Jai");
   map.put(6,"Amit");
   Set set=map.entrySet();//Converting to Set so that we can traverse
   Iterator itr=set.iterator();
   while(itr.hasNext()){
            Map.Entry entry=(Map.Entry)itr.next();
      System.out.println(entry.getKey()+" "+entry.getValue());     }  }  }
```

Out put

```
1  Amit
  5  Rahul
  2  Jai
  6 Amit
```

Generics in Java

The Java Generics programming is introduced in J2SE 5 to deal with type-safe objects. It makes the code stable by detecting the bugs at compile time.

Advantage of Java Generics

1) **Type-safety:** We can hold only a single type of objects in generics. It doesn?t allow to store other objects.

Arrays:

```
String [ ]  s= new String [10];
s[0]= “Amit”;
s[1]=new Integer(10);// CE
```

Collection

```
ArrayList  l =new ArrayList();
l.add(“Amit”);
l.add(new Integer(10));
```

Retrieval

```
String name1=(String)l.get(0);
String name2=(String)l.get(1);// CE
```

With Generics, it is required to specify the type of object we need to store.

```
List<Integer> list = new ArrayList<Integer>();
list.add(10);
list.add("10");// compile-time error
```

2) **Type casting is not required:** There is no need to typecast the object.

Arrays:

```
String [ ]  s= new String [10];
s[0]= “Amit”;
```

Retrieval

```
String name=s[0];
```

Collection

```
ArrayList  l =new ArrayList();
l.add(“Amit”);
```

Retrieval

```
String name1=l.get(0);//CE
```

```
List<Integer> l = new ArrayList<Integer>();
l.add(10);
```

Retrieval

```
String name=l.get[0];
```

3) **Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime.

Generic class
Like the generic class, we can create a generic method that can accept any type of arguments.

1.4v

```
class AL {
add(Obj o)
Object.get(int Integer)
}
```

1.5v

```
class AL<T> {
add(T t);
T.get(int Integer);
}
AL <String> l = new AL<String )();
```

Compiler replace with this code

```
class AL <String>
{
add(string s);
String get(int integer);
}
```

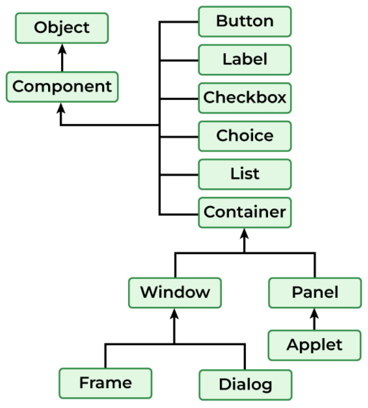
IV Module

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
- 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.

Hierarchy of Applet



Components: AWT provides various components such as buttons, labels, text fields, checkboxes, etc used for creating GUI elements for Java Applications.
Containers: AWT provides containers like panels, frames, and dialogues to organize and group components in the Application.
Layout Managers: Layout Managers are responsible for arranging data in the containers sone of the layout managers are BorderLayout, FlowLayout, etc.
Event Handling: AWT allows the user to handle the events like mouse clicks, key presses, etc. using event listeners and adapters.
Graphics and Drawing: It is the feature of AWT that helps to draw shapes, insert images and write text in the components of a Java Application.

Lifecycle of Java Applet
- Applet is initialized.
- Applet is started.
- Applet is painted.
- Applet is stopped.
- Applet is destroyed.

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

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

How to run an Applet?
There are two ways to run an applet
- By html file.
- By appletViewer tool (for testing purpose).

Simple example of Applet by html file:
```
//First.java
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet{
public void paint(Graphics g){
g.drawString("welcome",150,150);  }  }
```

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

Simple example of Applet by appletviewer tool:
```
//First.java
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet{
public void 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
```

Button:A button is basically a control component with a label that generates an event when pushed.

Example
```
  Button b = new Button("Click Here");
```

Label: The object of the Label class is a component for placing text in a container. It is used to display a single line of read only text.

Example

```
Label l1;
   l1 = new Label ("First Label.”);
```

Canvas

The Canvas class controls and represents a blank rectangular area where the application can draw or trap input events from the user. It inherits the Component class.

```
Frame f = new Frame("Canvas Example");
   // adding canvas to frame
   f.add(new MyCanvas());

   // setting layout, size and visibility of frame
   f.setLayout(null);
   f.setSize(400, 400);
   f.setVisible(true);
  }
```

Scrollbar

The object of Scrollbar class is used to add horizontal and vertical scrollbar. Scrollbar is a GUI component allows us to see invisible number of rows and columns.

```
Scrollbar s = new Scrollbar();
```

text components

TextField

The object of a TextField class is a text component that allows a user to enter a single line text and edit it. It inherits TextComponent class, which further inherits Component class.

```
 TextField t1;
      t1 = new TextField("Welcome.");
   t1.setBounds(50, 100, 200, 30);
```

Checkbox

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

```
Checkbox checkbox1 = new Checkbox("C++");
      checkbox1.setBounds(100, 100,  50, 50);
```

CheckboxGroup

The object of CheckboxGroup class is used to group together a set of Checkbox. At a time only one check box button is allowed to be in "on" state and remaining check box button in "off" state. It inherits the object class.

```
 CheckboxGroup cbg = new CheckboxGroup();
        Checkbox checkBox1 = new Checkbox("C++", cbg, false);
        checkBox1.setBounds(100,100, 50,50);
        Checkbox checkBox2 = new Checkbox("Java", cbg, true);
        checkBox2.setBounds(100,150, 50,50);
```

Choice: The object of Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of a menu. It inherits Component class.

```
  Choice c = new Choice();
 c.setBounds(100, 100, 75, 75);
         c.add("Item 1");
        c.add("Item 2");
        c.add("Item 3");
        c.add("Item 4");
        c.add("Item 5");
```

Panel: The Panel is a simplest container class. It provides space in which an application can attach any other component. It inherits the Container class.

```
Panel panel=new Panel();
        panel.setBounds(40,80,200,200);
        panel.setBackground(Color.gray);
```

Dialog: The Dialog control represents a top level window with a border and a title used to take some form of input from the user. It inherits the Window class.

```
Frame f= new Frame();
        d = new Dialog(f , "Dialog Example", true);
```

MenuItem and Menu

The object of MenuItem class adds a simple labeled menu item on menu. The items used in a menu must belong to the MenuItem or any of its subclass.

```
MenuBar mb=new MenuBar();
         Menu menu=new Menu("Menu");
         Menu submenu=new Menu("Sub Menu");
         MenuItem i1=new MenuItem("Item 1");
         MenuItem i2=new MenuItem("Item 2");
         menu.add(i1);
         submenu.add(i2);
         menu.add(submenu);
         mb.add(menu);
```

Graphics in Applet

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

```
public class GraphicsDemo extends Applet{
```

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

Java LayoutManagers

The LayoutManagers are used to arrange components in a particular manner. The Java LayoutManagers facilitates us to control the positioning and size of the components in GUI forms. There are the following classes that represent the layout managers:

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

BorderLayout

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

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

```
 f = new JFrame();

       JButton b1 = new JButton(“NORTH");
   JButton b2 = new JButton(“SOUTH");
      f.add(b1, BorderLayout.NORTH);
   f.add(b2, BorderLayout.SOUTH);
```

GridLayout : The Java GridLayout class is used to arrange the components in a rectangular grid. One component is displayed in each rectangle.

```
GridLayoutExample()
{
frameObj = new JFrame();
JButton btn1 = new JButton("1");
JButton btn2 = new JButton("2");
frameObj.add(btn1); frameObj.add(btn2);
frameObj.setLayout(new GridLayout());
```

FlowLayout: The Java FlowLayout class is used to arrange the components in a line, one after another (in a flow). It is the default layout of the applet or panel.
Fields of FlowLayout class

```
public static final int LEFT
public static final int RIGHT
public static final int CENTER
public static final int LEADING
public static final int TRAILING
```

```
FlowLayoutExample()
{
      frameObj = new JFrame();
     JButton b1 = new JButton("1");
   JButton b2 = new JButton("2");

     frameObj.add(b1); frameObj.add(b2);

    frameObj.setLayout(new FlowLayout());
```

CardLayout: The Java CardLayout class manages the components in such a manner that only one component is visible at a time. It treats each component as a card that is why it is known as CardLayout.

```
crd = new CardLayout();
```

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

```
 GridBagLayout layout = new GridBagLayout();
```

Events

An event can be defined as changing the state of an object or behavior by performing actions. Actions can be a button click, cursor movement, keypress through keyboard or page scrolling, etc.

Event Source : Description
Button : Generates action events when the button is pressed.
Check box : Generates item events when the check box is selected or deselected.
Choice : Generates item events when the choice is changed.
List : Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu item : Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar : Generates adjustment events when the scroll bar is manipulated.
Text components : Generates text events when the user enters a character.
Window : Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Java Event classes and Listener interfaces

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

Steps to perform Event Handling
Following steps are required to perform event handling:

- Register the component with the Listener

Registration Methods
For registering the component with the Listener, many classes provide the registration methods. For **example**

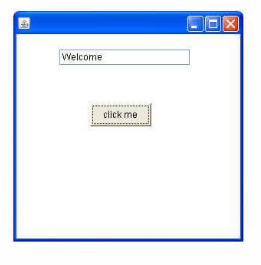
Button
public void addActionListener(ActionListener a){}
MenuItem
public void addActionListener(ActionListener a){}
TextField
public void addActionListener(ActionListener a){}
public void addTextListener(TextListener a){}
TextArea
public void addTextListener(TextListener a){}
Checkbox
public void addItemListener(ItemListener a){}
Choice

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

Java event handling by implementing ActionListener

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

Out put



The Delegation Model

The Delegation Model is available in Java since Java 1.1. it provides a new delegation-based event model using AWT to resolve the event problems. It provides a convenient mechanism to support complex Java programs.

Design Goals

The design goals of the event delegation model are as following:

- It is easy to learn and implement

- It supports a clean separation between application and GUI code.
- It provides robust event handling program code which is less error-prone (strong compile-time checking)
- It is Flexible, can enable different types of application models for event flow and propagation.
- It enables run-time discovery of both the component-generated events as well as observable events.
- It provides support for the backward binary compatibility with the previous model.

MouseListener

The Java MouseListener is notified whenever you change the state of mouse. It is notified against MouseEvent. The MouseListener interface is found in java.awt.event package. It has five methods.

Methods of MouseListener interface

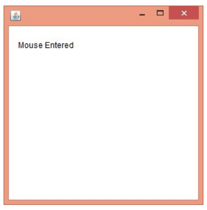
The signature of 5 methods found in MouseListener interface are given below:

```
public abstract void mouseClicked(MouseEvent e);
public abstract void mouseEntered(MouseEvent e);
public abstract void mouseExited(MouseEvent e);
public abstract void mousePressed(MouseEvent e);
public abstract void mouseReleased(MouseEvent e);
```

Example

```
import java.awt.*;
import java.awt.event.*;
public class MouseListenerExample extends Frame implements MouseListener{
   Label l;
   MouseListenerExample(){
      addMouseListener(this);
      l=new Label();
      l.setBounds(20,50,100,20);
      add(l);
      setSize(300,300);
      setLayout(null);
      setVisible(true);
   }
   public void mouseClicked(MouseEvent e) {
      l.setText("Mouse Clicked");
   }
   public void mouseEntered(MouseEvent e) {
      l.setText("Mouse Entered");
   }
   public void mouseExited(MouseEvent e) {
      l.setText("Mouse Exited");
   }
   public void mousePressed(MouseEvent e) {
      l.setText("Mouse Pressed");
   }
   public void mouseReleased(MouseEvent e) {
      l.setText("Mouse Released");
```

```
    }
public static void main(String[] args) {
    new MouseListenerExample();
}
}
```

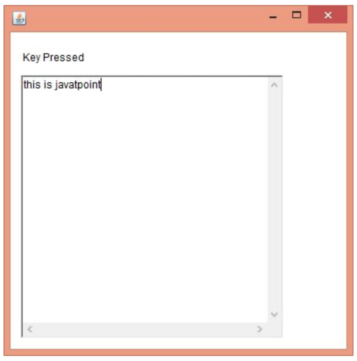

Mouse Entered

Java KeyListener Interface

The Java KeyListener is notified whenever you change the state of key. It is notified against KeyEvent. The KeyListener interface is found in java.awt.event package, and it has three methods.

public abstract void keyPressed (KeyEvent e); It is invoked when a key has been pressed.
public abstract void keyReleased (KeyEvent e);It is invoked when a key has been released.
public abstract void keyTyped (KeyEvent e);It is invoked when a key has been typed.

```
import java.awt.*;
import java.awt.event.*;
public class KeyListenerExample extends Frame implements KeyListener {
 Label l;
    TextArea area;
    KeyListenerExample() {
            l = new Label();
        l.setBounds (20, 50, 100, 20);
        area = new TextArea();
        area.setBounds (20, 80, 300, 300);
        area.addKeyListener(this);
        add(l);
add(area);
        setSize (400, 400);
        setLayout (null);
        setVisible (true);
    }
    public void keyPressed (KeyEvent e) {
        l.setText ("Key Pressed");
    }
    public void keyReleased (KeyEvent e) {
        l.setText ("Key Released");
    }
    public void keyTyped (KeyEvent e) {
        l.setText ("Key Typed");
    }
      public static void main(String[] args) {
        new KeyListenerExample();
    }
}
```


Key Pressed

```
this is javatpoint
```

Adapter Classes

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. So it saves code.

Pros of using Adapter classes:

- It assists the unrelated classes to work combinedly.
- It provides ways to use classes in different ways.
- It increases the transparency of classes.
- It provides a way to include related patterns in the class.
- It provides a pluggable kit for developing an application.
- It increases the reusability of the class.
- 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.

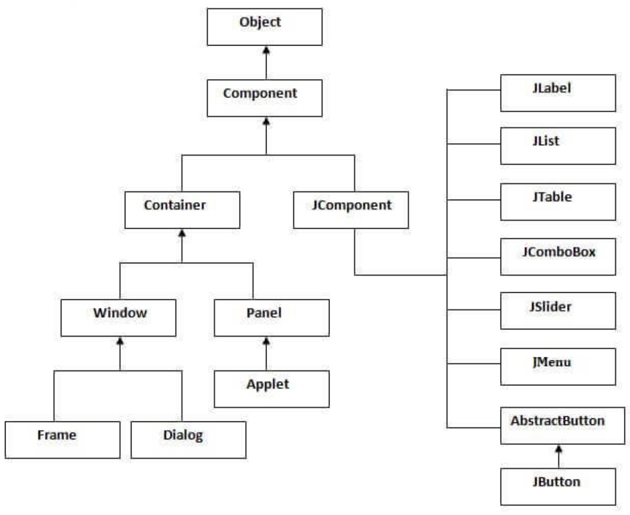
V Module

Swing

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

Difference between AWT and Swing

No.	Java AWT	Java Swing
1)	AWT components are **platform-dependent**.	Java swing components are **platform-independent**.
2)	AWT components are **heavyweight**.	Swing components are **lightweight**.
3)	AWT **doesn't support pluggable look and feel**.	Swing **supports pluggable look and feel**.
4)	AWT provides **less components** than Swing.	Swing provides **more powerful components** such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT **doesn't follows MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing **follows MVC**.



Hierarchy of Java Swing classes

JApplet class
As we prefer Swing to AWT. Now we can use JApplet that can have all the controls of swing. The JApplet class extends the Applet class.

Example
```
import java.applet.*;
import javax.swing.*;
import java.awt.event.*;
public class EventJApplet extends JApplet implements ActionListener{
JButton b;
JTextField tf;
public void init(){
tf=new JTextField();
tf.setBounds(30,40,150,20);
b=new JButton("Click");
b.setBounds(80,150,70,40);
add(b);add(tf);
b.addActionListener(this);
setLayout(null);  }
public void actionPerformed(ActionEvent e){
tf.setText("Welcome");  }  }
```

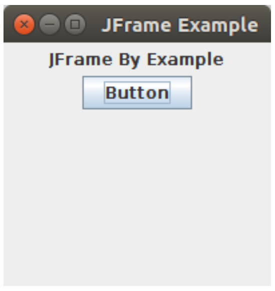
myapplet.html

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

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.

```
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
public class JFrameExample {
   public static void 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);
    }
}
```



JComponent
The JComponent class is the base class of all Swing components except top-level containers. Swing components whose names begin with "J" are descendants of the JComponent class. For example, JButton, JScrollPane, JPanel, JTable etc. But, JFrame and JDialog don't inherit JComponent class because they are the child of top-level containers.

Java JComponent Example

```
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JComponent;
import javax.swing.JFrame;
class MyJComponent extends JComponent {
      public void paint(Graphics g) {
        g.setColor(Color.green);
        g.fillRect(30, 30, 100, 100);
      }
}
public class JComponentExample {
      public static void main(String[] arguments) {
        MyJComponent com = new MyJComponent();
        // create a basic JFrame
        JFrame.setDefaultLookAndFeelDecorated(true);
        JFrame frame = new JFrame("JComponent Example");
        frame.setSize(300,200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // add the JComponent to main frame
        frame.add(com);
```

```
    frame.setVisible(true);
  } }
```

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

```
JLabel l1,l2;
  l1=new JLabel("First Label.");
  l1.setBounds(50,50, 100,30);
```

JTextField
The object of a JTextField class is a text component that allows the editing of a single line text. It inherits JTextComponent class.

```
 JTextField t1;
  t1=new JTextField("Welcome to Javatpoint.");
  t1.setBounds(50,100, 200,30);
```

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

```
JButton b=new JButton("Click Here");
  b.setBounds(50,100,95,30);
```

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

```
JCheckBox checkBox1 = new JCheckBox("C++");
    checkBox1.setBounds(100,100, 50,50);
    JCheckBox checkBox2 = new JCheckBox("Java", true);
    checkBox2.setBounds(100,150, 50,50);
```

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

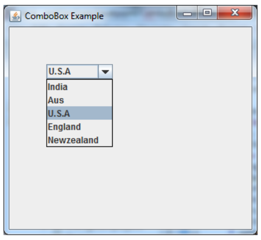
```
JRadioButton r1=new JRadioButton("A) Male");
JRadioButton r2=new JRadioButton("B) Female");
```

```
r1.setBounds(75,50,100,30);
r2.setBounds(75,100,100,30);
```

JComboBox

The object of Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of a menu. It inherits JComponent class.

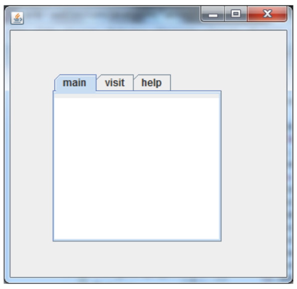
```
String country[]={"India","Aus","U.S.A","England","Newzealand"};
    JComboBox cb=new JComboBox(country);
    cb.setBounds(50, 50,90,20);
```



JTabbedPane

The JTabbedPane class is used to switch between a group of components by clicking on a tab with a given title or icon. It inherits JComponent class.

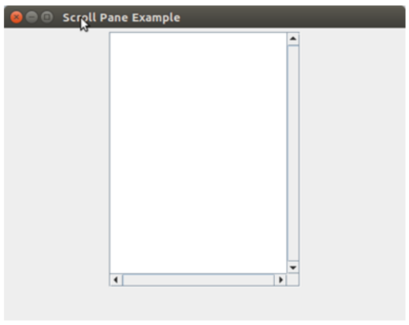
```
 JTabbedPane tp=new JTabbedPane();
    tp.setBounds(50,50,200,200);
    tp.add("main",p1);
    tp.add("visit",p2);
    tp.add("help",p3);
```



JScrollPane

A JscrollPane is used to make scrollable view of a component. When screen size is limited, we use a scroll pane to display a large component or a component whose size can change dynamically.

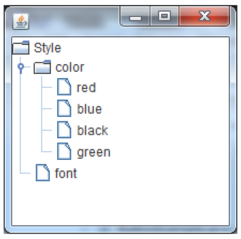
```
  JScrollPane scrollableTextArea = new JScrollPane(textArea);

        scrollableTextArea.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR
_ALWAYS);
        scrollableTextArea.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALW
AYS);
```



JTree

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

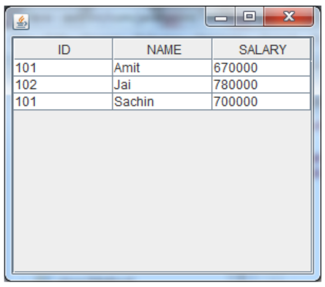
```
 DefaultMutableTreeNode style=new DefaultMutableTreeNode("Style");
   DefaultMutableTreeNode color=new DefaultMutableTreeNode("color");
   DefaultMutableTreeNode font=new DefaultMutableTreeNode("font");
   style.add(color);
   style.add(font);
   DefaultMutableTreeNode red=new DefaultMutableTreeNode("red");
   DefaultMutableTreeNode blue=new DefaultMutableTreeNode("blue");
   DefaultMutableTreeNode black=new DefaultMutableTreeNode("black");
   DefaultMutableTreeNode green=new DefaultMutableTreeNode("green");
   color.add(red); color.add(blue); color.add(black); color.add(green);
```



JTable

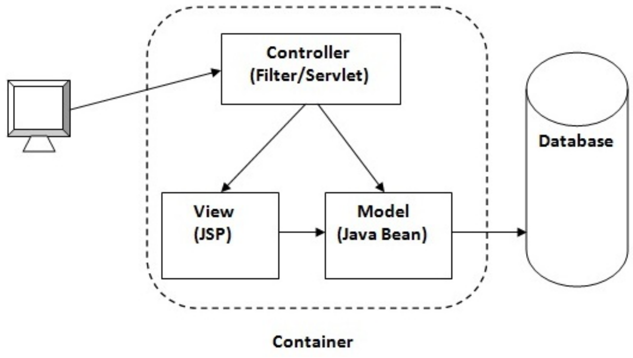
The JTable class is used to display data in tabular form. It is composed of rows and columns.

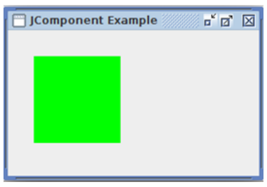
```
String data[][]={ {"101","Amit","670000"},
                  {"102","Jai","780000"},
                  {"101","Sachin","700000"}};
   String column[]={"ID","NAME","SALARY"};
   JTable jt=new JTable(data,column);
   jt.setBounds(30,40,200,300);
```



ID	NAME	SALARY
101	Amit	670000
102	Jai	780000
101	Sachin	700000

MVC stands for Model View and Controller. It is a design pattern that separates the business logic, presentation logic and data.
Controller acts as an interface between View and Model. Controller intercepts all the incoming requests.
Model represents the state of the application i.e. data. It can also have business logic.
View represents the presentaion i.e. UI(User Interface).




JComponent Example