ISO 9001:2015 Certified Institution

Estd. : 2001

Balaji Institute of Technology & Science

Laknepally, NARSAMPET, Warangal (Rural) – 506331

Accredited By NBA (UG – CE, ME, ECE & CSE Programmes) **& NAAC**

(Affiliated to JNTUH, Hyderabad and Approved by the AICTE, New Delhi)

BITS NARSAMPET www.bitswgl.ac.in, email: principal@bitswgl.ac.in :: Ph. 98660 50044, Fax 08718-230521

COURSE FILE ON Object oriented through JAVA PROGRAMMING

Course Code – 22CS413PC

II B.Tech II-SEMESTER

A.Y.: 2024-2025

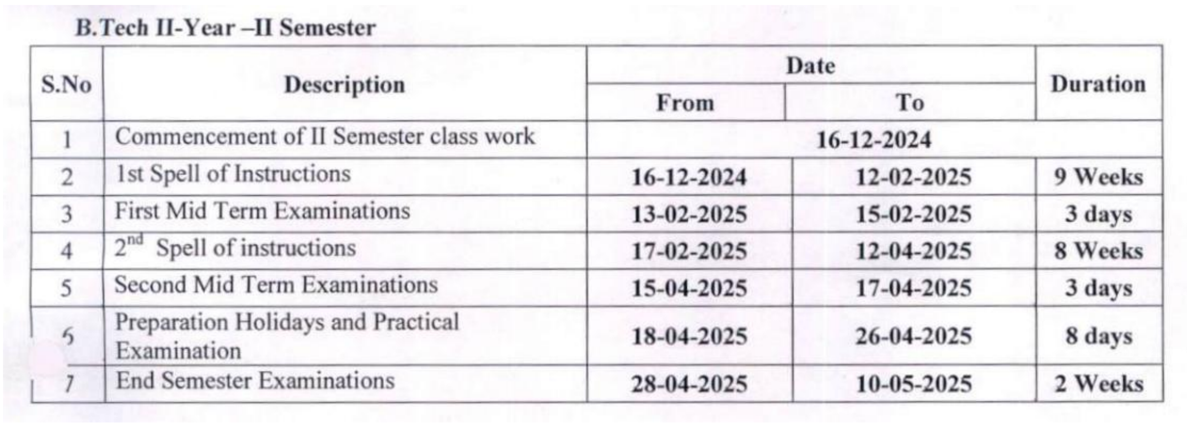
Prepared

by

Mr A.Raju

Assistant Professor

1.Academic Callender



B.Tech II-Year –II Semester

S.No	Description	Date		Duration
		From	To	
1	Commencement of II Semester class work	16-12-2024		
2	1st Spell of Instructions	16-12-2024	12-02-2025	9 Weeks
3	First Mid Term Examinations	13-02-2025	15-02-2025	3 days
4	2nd Spell of instructions	17-02-2025	12-04-2025	8 Weeks
5	Second Mid Term Examinations	15-04-2025	17-04-2025	3 days
6	Preparation Holidays and Practical Examination	18-04-2025	26-04-2025	8 days
7	End Semester Examinations	28-04-2025	10-05-2025	2 Weeks

2.CO-PO for Object Oriented Programming through Java

CO1: Understand the fundamental principles of Object-Oriented Programming (Encapsulation, Inheritance, Polymorphism, and Abstraction).
CO2: Implement Java programs using classes, objects, constructors, and methods.
CO3: Apply inheritance and polymorphism to create reusable and modular programs.
CO4: Utilize abstract classes and interfaces to design extensible applications.
CO5: Implement exception handling and multithreading in Java for robust applications.
CO6: Develop real-world applications using Java frameworks and libraries.

Name of the Subject: C215 (Object Oriented Programming through Java-22CS413PC)

Year of Study: 2021-2022

CO	**PO1**	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**
C215.1	3	2	3	-	-	2	-	-	-	-	-	2
C215.2	3	-	2	3	2	3	-	-		-	-	3
C215.3	2	-	2	2	-	-	-	-	-	1	-	2
C215.4	-	-	-	-	3	2	-	-	-	-	1	3
C215.5	3	3	-	2	-	2	-	-	-	-	-	2
Average	2.75	2.50	2.33	2.33	2.50	2.25	-	-	-	1.00	1.00	2.40

3.Syllubus

B.Tech. CSE (AI and ML) Syllabus R22-Regulations
BALAJI INSTITUTE OF TECHNOLOGY AND SCIENCE
(AUTONOMOUS)
22CS413PC: OBJECT ORIENTED PROGRAMMING THROUGH JAVA
B.Tech. II Year II Sem. L T P C
3 0 0 3

Course Objectives

UNIT - I

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.

UNIT - II

Inheritance, Packages and Interfaces – 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,differences between classes and interfaces, defining an interface, implementing interface, applying interfaces, variables in interface and extending interfaces. Exploring java.io.

UNIT - III

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.

UNIT - IV

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 – scroll pane, dialogs, menu bar, graphics, layout manager – layout manager types – border, grid, flow, card and grid bag.

UNIT - V

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 JC omponent, Icons and Labels, text fields, buttons – The JButton class, Check boxes, Radio buttons, Combo boxes, Tabbed Panes, Scroll Panes, Trees, and Tables

4.Lesson Plan

UNIT IObject Oriented Thinking and Java Basics (No. of Lectures – 17)

Topics (as per syllabus)	Sub Topics	Lect. No.	Scheduled Date	Delivered Date
Java Basics	• Prerequisites • Course Objectives and Course Outcomes	1	19.12.24	
	Need for OOP Paradigm, OOP Concepts	2	20.12.24	
	Abstraction Mechanisms, A Way of Viewing World – Agents, Responsibility, Messages, Methods	3	20.12.24	
	History of Java, Java Buzzwords	4	24.12.24	
	Data Types, Variables	5	25.12.24	
	Scope and Life Time of Variables	6	26.12.24	
	Arrays, Operators, Expressions	7	27.12.24	
	Control Statements	8	27.12.24	
	Type Conversion and Casting, Simple Java Program	9	31.12.24	
OOP Concepts	Concepts of Classes, Objects	10	1.1.2025	
	Constructors, Methods, Access Control	11	2.1.2025	
	This Keyword, Garbage Collection	12	3.1.2025	
	Overloading Methods and Constructors	13	3.1.2025	
	Inheritance, Overriding and Exceptions	14	7.1.2025	
Functions	Parameter Passing, Recursion	15	8.1.2025	
	Nested and Inner Classes	16	10.1.2025	
	Exploring String Class	17	10.1.2025	
Overview on UNIT – I	OOP and Java Basics • Assignment-1	18	16.1.2025	

Topics (as per syllabus)	Sub Topics	Lect. No.	Scheduled Date	Delivered Date
Inheritance	• Prerequisites • UNIT-II Objectives • Learning Outcomes	19	17.1.2025	
	Hierarchical Abstractions, Base Class Object, Subclass, Subtype, Substitutability	20	17.1.2025	
	Forms of Inheritance- Specialization, Specification	21	21.1.2025	
	Construction, Extension, Limitation, Combination, Benefits of Inheritance	22	22.1.2025	
	Member Access Rules, Super Uses	23	23.1.2025	
	Using Final with Inheritance	24	24.1.2025	
	Polymorphism- Method Overriding, Abstract Classes	25	24.1.2025	
Packages and Interfaces	Defining, Creating and Accessing a Package	26	28.1.2025	
	Understanding Classpath, Importing Packages, Differences between Classes and Interfaces	27	29.1.2025	
	Defining an Interface, Implementing Interface, Applying Interfaces	28	30.1.2025	
	Variables in Interface and Extending Interfaces	29	31.1.2025	
	Exploring Java.IO	30	31.1.2025	
Overview on UNIT - II	• Inheritance, Packages and Interfaces • Assignment-2	31	4.02.2025	
Review about Mid I Exam	• Review of Theory Questions • Tips to get good marks	32	5.02.2025	
Mid I Marks Distribution	• Marks Distribution • Counsel the students (Absent/got poor marks)			

Topics (as per syllabus)	Sub Topics	Lect. No.	Scheduled Date	Topic Delivered Date
Exception Handling	• Prerequisites • UNIT-III Objectives and Learning Outcomes	33	6.02.2025	
	Concepts of Exception Handling, Benefits of Exception Handling	34	7.02.2025	
	Termination or Resumptive Models, Exception Hierarchy	35	7.02.2025	
	Usage of Try, Catch, Throw, Throws and Finally	36	11.02.2025	
	Built in Exceptions	37	12.02.2025	
	Creating Own Exception Sub Classes	38	13.02.2025	
Multithreading	String Handling, Exploring Java.Util,	39	14.02.2025	
	Thread Life Cycle, Creating Threads, Thread Priorities	40	14.02.2025	
	Synchronizing Threads, Interthread Communication	41	18.02.2025	
Threads	Thread Groups, Daemon Threads, Enumerations	42	19.02.2025	
	Autoboxing, Annotations, Generics.	43	20.02.2025	
Overview on UNIT-III:	• Multithreading • Assignment-3	44	21.02.2025	

Topics (as per syllabus)	Sub Topics	Lect. No.	Scheduled Date	Delivered Date
Event Handling	Events, Event Sources, Event Classes	45	21.02.2025	
	Event Listeners, Delegation Event Model,	46	25.02.2025	
	Handling Mouse and Keyboard Events	47	27.02.2025	
	Adapter Classes, AWT Class Hierarchy	48	28.02.2025	
User Interface	Labels, Button, Canvas	49	28.02.2025	

Components	Scrollbars, Text Components, Check Box, Choices	50	4.03.2025	
	Lists Panels – Scrollpane, Dialogs, Menubar, Graphics	51	5.03.2025	
	Layout Manager Types – Border, Grid, Flow, Card	52	6.03.2025	
Overview on UNIT-IV	• Event Handling • Assignment-4	53	7.03.2025	
Event handling	Real-time examples	54	11.03.2025	
Applets	Concepts f Applets, Differences between Applets and Applications	55	12.03.2025	
	Life Cycle of an Applet, Types of Applets	56	13.03.2025	
	Creating Applets, Passing Parameters to Applets.	57	18.03.2025	
Swings	Introduction, Limitations of AWT,	58	19.03.2025	
	MVC Architecture	59	20.03.2025	
	Jframe and Jcomponent,	60	21.03.2025	
	Icons and Labels	61	25.03.2025	
	Text Fields	62	26.03.2025	
	Check Boxes, Radio Buttons	63	27.03.2025	
	Tabbed Panes, Scroll Panes, Trees, and Tables.	64	04.04.2025	
Overview on UNIT-V	• Applets and Swings • Assignment-5	65	08.04.2025	
Swings	Real-time examples	66	09.04.2025	
Review about Mid II Exam	• Review of theory and objective Questions	67	10.04.2025	
	• Tips to get good marks	68	11.04.2025	

5.Lecture Notes:

OBJECT ORIENTED PROGRAMMING THROUGH JAVA

UNIT-I

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.

OBJECT ORIENTED THINKING

- When computers were first invented, programming was done manually by toggling in a binary machine instructions by use of front panel.
- As programs began to grow, high level languages were introduced that gives the programmer more tools to handle complexity.
- The first widespread high level language is FORTRAN. Which gave birth to structured programming in 1960's. The Main problem with the high level language was they have no specific structure and programs becomes larger, the problem of complexity also increases.
- So C became the popular structured oriented language to solve all the above problems.
- However in SOP, when project reaches certain size its complexity exceeds. So in 1980's a new way of programming was invented and it was OOP. OOP is a programming methodology that helps to organize complex programs through the use of inheritance, encapsulation & polymorphism.

NEED FOR OOP PARADIGM

- Traditionally, the structured programming techniques were used earlier.
- There were many problems because of the use of structured programming technique.
- The structured programming made use of a top-down approach.
- To overcome these problems the object oriented programming concept was created.
- The object oriented programming makes use of bottom-up approach.
- It also manages the increasing complexity.
- The description of an object-oriented program can be given as, a data that controls access to code.
- The object-oriented programming technique builds a program using the objects along with a set of well-defined interfaces to that object.

- The object-oriented programming technique is a paradigm, as it describes the way in which elements within a computer program must be organized.
- It also describes how those elements should interact with each other.
- In OOP, data and the functionality are combined into a single entity called an object.
- Classes as well as objects carry specific functionality in order to perform operations and to achieve the desired result.
- The data and procedures are loosely coupled in procedural paradigm.
- Whereas in OOP paradigm, the data and methods are tightly coupled to form objects.
- These objects helps to build structure models of the problem domain and enables to get effective solutions.
- OOP uses various principles (or) concepts such as abstraction, inheritance, encapsulation and polymorphism. With the help of abstraction, the implementation is hidden and the functionality is exposed.
- Use of inheritance can eliminate redundant code in a program. Encapsulation enables the data and methods to wrap into a single entity. Polymorphism enables the reuse of both code and design.

SUMMARY OF OOP CONCEPTS

- Everything is an object.
- Computation is performed by objects communicating with each other, requesting that other objects perform actions. Objects communicate by sending & receiving *messages*. A message is a request for an action bundled with whatever arguments may be necessary to complete the task.
- Each object has its own *memory*, which consists of other objects.
- Every Object is an *instance* of class. A class simply represents a grouping of similar objects, such as integers or lists.
- The class is the repository for *behavior* associated with an object. That is all objects that are instances of same class can perform the same actions.
- Classes are organized into a singly rooted tree structure, called *inheritance hierarchy.*

OOP CONCEPTS

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.

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

Class

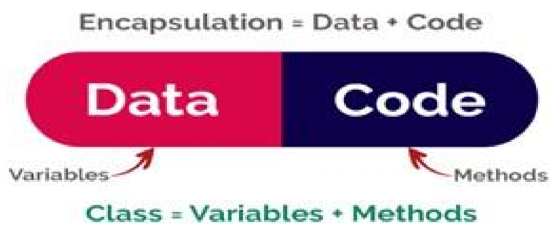
Class is a blue print which is containing only list of variables and methods and no memory is allocated for them. A class is a group of objects that has common properties.

Object

- Any entity that has state and behavior is known as an object.
- For example a chair, pen, table, keyboard, bike, etc. It can be physical or logical. An Object can be defined as an instance of a class.
- **Example:** A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

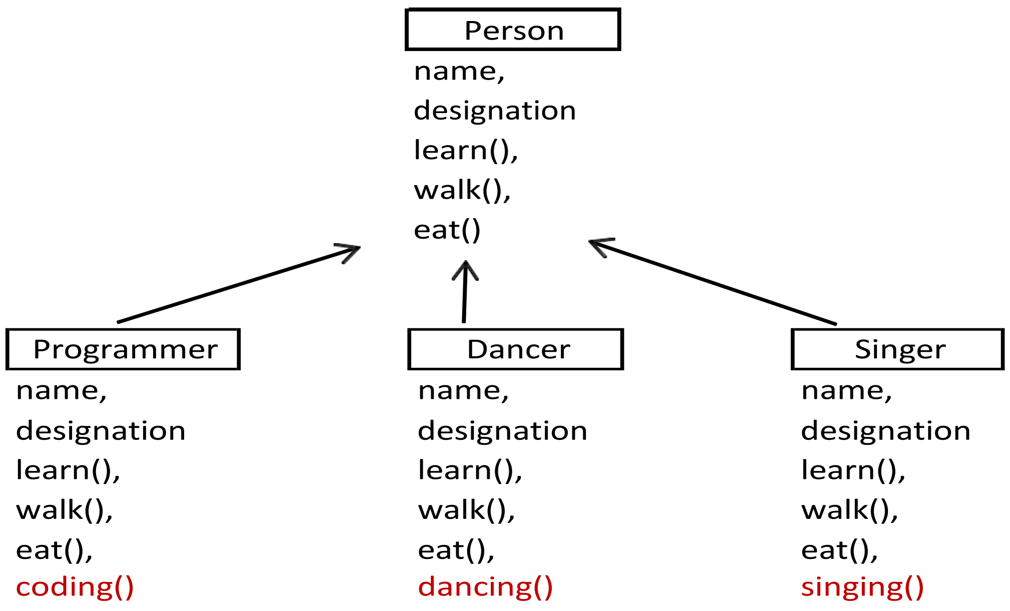
Encapsulation

- Encapsulation is the process of combining data and code into a single unit.
- 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.
- In the inheritance concept, the class which provides properties is called as parent class and the class which recieves the properties is called as child class.



Polymorphism

- Polymorphism is the process of defining same method with different implementation. That means creating multiple methods with different behaviors.
- The java uses method overloading and method overriding to implement polymorphism.
- Method overloading - multiple methods with same name but different parameters.
- Method overriding - multiple methods with same name and same parameters.

Abstraction

- Abstraction is hiding the internal details and showing only essential functionality.
- In the abstraction concept, we do not show the actual implementation to the end user, instead we provide only essential 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 Java, or any programming language, is an essential skill for software developers.
- As your Java projects grow in size and complexity, maintaining, debugging, and extending your code can become challenging.

Here are some strategies to help you cope with complexity in Java:

1. **Modularization**: Breaking down the code into smaller, self-contained modules, classes, or packages to manage complexity. Each module should have a specific responsibility and interact with others through well-defined interfaces.
2. **Design Patterns: Using** established design patterns to solve common architectural and design problems. Design patterns provide proven solutions to recurring challenges in software development.
3. **Encapsulation**: Restricting access to class members using access modifiers (public, private, protected) to hide implementation details and provide a clear API for interacting with the class.
4. **Abstraction**: Creating abstract classes and interfaces to define contracts that classes must adhere to, making it easier to work with different implementations.
5. **Documentation**: Writing comprehensive documentation, including code comments and Javadoc, to explain how the code works, its purpose, and how to use it.
6. **Testing**: Implementing unit tests to ensure that individual components of the code function correctly, which helps identify and prevent bugs.
7. **Code Reviews**: Collaborating with team members to review and provide feedback on code to catch issues and ensure code quality.
8. **Version Control**: Using version control systems like Git to manage changes, track history, and collaborate with others effectively.
9. **Refactoring**: Regularly improving and simplifying the codebase by removing redundancy and improving its structure.

ABSTRACTION MECHANISM

- In Java, abstraction is a fundamental concept in object-oriented programming that allows you to hide complex implementation details while exposing a simplified and well-defined interface.
- Abstraction mechanisms in Java include the use of abstract classes and interfaces.

A WAY OF VIEWING WORLD

- 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.

AGENTS AND COMMUNITIES

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. **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.

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.

MESSAGES & 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.

HISTORY OF JAVA

- Java is a object oriented programming language.
- Java was created based on C and C++.
- Java uses C syntax and many of the object-oriented features are taken from C++.
- Before Java was invented there were other languages like COBOL, FORTRAN, C, C++, Small Talk, etc.
- These languages had few disadvantages which were corrected in Java.
- Java also innovated many new features to solve the fundamental problems which the previous languages could not solve.
- Java was developed by James Gosling, Patrick Naughton, Chris warth, Ed Frank and Mike Sheridon at Sun Microsystems in the year 1991.
- This language was initially called as “OAK” but was renamed as “Java” in 1995.
- The primary motivation behind developing java was the need for creating a platform independent Language (Architecture Neutral), that can be used to create a software which can be embedded in various electronic devices such as remote controls, micro ovens etc.
- The problem with C, C++ and most other languages is that, they are designed to compile on specific targeted CPU (i.e. they are platform dependent), but java is platform Independent which can run on a variety of CPU’s under different environments.
- The secondary factor that motivated the development of java is to develop the applications that can run on Internet. Using java we can develop the applications which can run on internet i.e. Applet. So java is a platform Independent Language used for developing programs which are platform Independent and can run on internet.

JAVA BUZZWORDS(JAVA FEATURES)

- 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 Following list of Buzz Words

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Architecture-neutral (or) Platform Independent
- Multi-threaded
- Interpreted
- High performance
- Distributed
- Dynamic

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 .
- If a program yields the same result on every machine, then that program is called portable.
- Java programs are portable
- This is the result of java System independence nature.

Object-oriented

- Java is an object oriented programming language.
- This means java programs use objects and classes.

Robust

- Robust means strong.
- Java programs are strong and they don’t crash easily like a C or C++ programs

There are two reasons

- Java has got excellent inbuilt exception handling features. An exception is an error that occurs at runtime. If an exception occurs, the program terminates suddenly giving rise to problems like loss of data. Overcoming such problem is called exception handling.
- Most of the C and C++ programs crash in the middle because of not allocating sufficient memory or forgetting the memory to be freed in a program. Such problems will not occur in java because the user need not allocate or deallocate the memory in java. Everything will be taken care of by JVM only.

Architecture-neutral (or) Platform Independent

- Java has invented to archive "write once; run anywhere, anytime, forever".
- The java provides JVM (Java Virtual Machine) 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 programs are compiled to generate byte code.
- This byte code can be downloaded and interpreted by the interpreter in JVM.
- If we take any other language, only an interpreter or a compiler is used to execute the program.
- But in java, we use both compiler and interpreter for the execution.

High performance

- The problem with interpreter inside the JVM is that it is slow.
- Because of Java programs used to run slow.
- To overcome this problem along with the interpreter.
- Java soft people have introduced JIT (Just in Time) compiler, to enhance the speed of execution.
- So now in JVM, both interpreter and JIT compiler work together to run the program.

Distributed

- Information is distributed on various computers on a network.
- Using Java, we can write programs, which capture information and distribute it to the client.
- This is possible because Java can handle the protocols like TCP/IP and UDP.

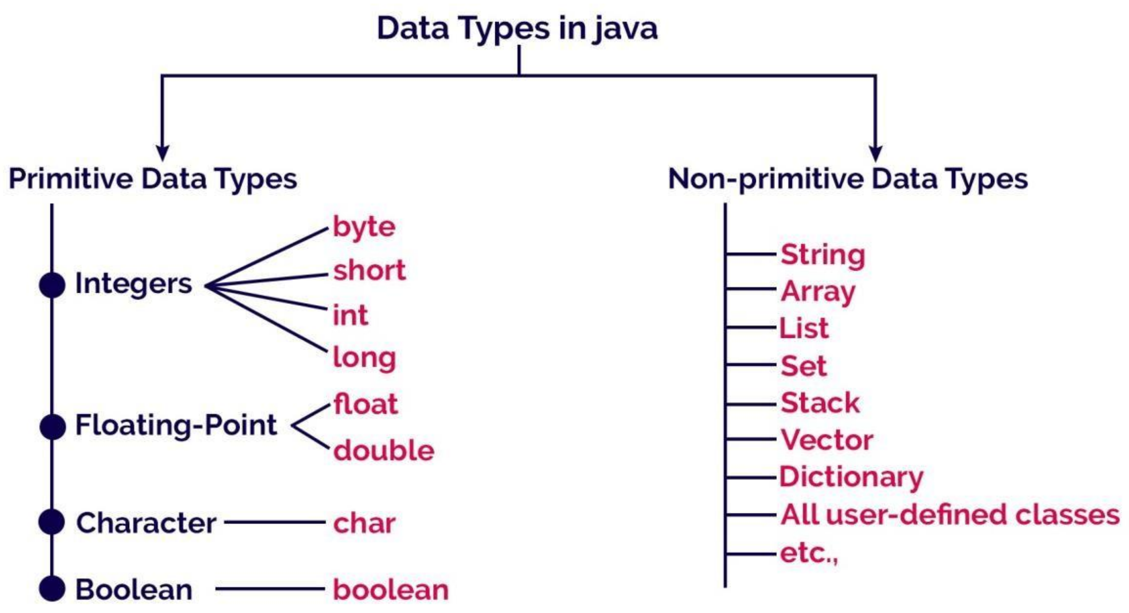
Dynamic

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).

DATA TYPES IN JAVA

Java programming language has a rich set of data types. The data type is a category of data stored in variables. In java, data types are classified into two types and they are as follows.

- Primitive Data Types
- Non-primitive Data Types



Primitive Data Types

The primitive data types are built-in data types and they specify the type of value stored in a variable and the memory size.

Integer Data Types

Integer Data Types represent integer numbers, i.e numbers without any fractional parts or decimal points.

Data Type	Memory Size	Minimum and Maximum values	Default Value
byte	1 byte	-128 to +128	0
short	2 bytes	-32768 to +32767	0
int	4 bytes	-2147483648 to +2147483647	0
long	8 bytes	-9223372036854775808 to +9223372036854775807	0L

Float Data Types

Float data types are represent numbers with decimal point.

Data Type	Memory Size	Minimum and Maximum values	Default Value
float	4 byte	-3.4e38 to -1.4e-45 and 1.4e-45 to 3.4e38	0.0f
double	8 bytes	-1.8e308 to -4.9e-324 and 4.9e-324 to 1.8e308	0.0d

Note: Float data type can represent up to 7 digits accurately after decimal point.

Double data type can represent up to 15 digits accurately after decimal point.

Character Data Type

Character data type are represents a single character like a, P, &, *,..etc.

Data Type	Memory Size	Minimum and Maximum values	Default Value
char	2 bytes	0 to 65538	\u0000

Boolean Data Types

Boolean data types represent any of the two values, true or false. JVM uses 1 bit to represent a Boolean value internally.

Data Type	Memory Size	Minimum and Maximum values	Default Value
boolean	1 byte	0 or 1	0 (false)

VARIABLES

Variable is a name given to a memory location where we can store different values of the same data type during the program execution.

The following are the rules to specify a variable name...

- A variable name may contain letters, digits and underscore symbol
- Variable name should not start with digit.
- Keywords should not be used as variable names.
- Variable name should not contain any special symbols except underscore(_).
- Variable name can be of any length but compiler considers only the first 31 characters of the variable name.

Declaration of Variable

Declaration of a variable tells to the compiler to allocate required amount of memory with specified variable name and allows only specified datatype values into that memory location.

Syntax: datatype variablename;

Example : int a;

Syntax : data_type variable_name_1, variable_name_2,...;

Example : int a, b;

Initialization of a variable:

Syntax: datatype variablename = value;

Example : int a = 10;

Syntax : data_type variable_name_1=value, variable_name_2 = value;

Example : int a = 10, b = 20;

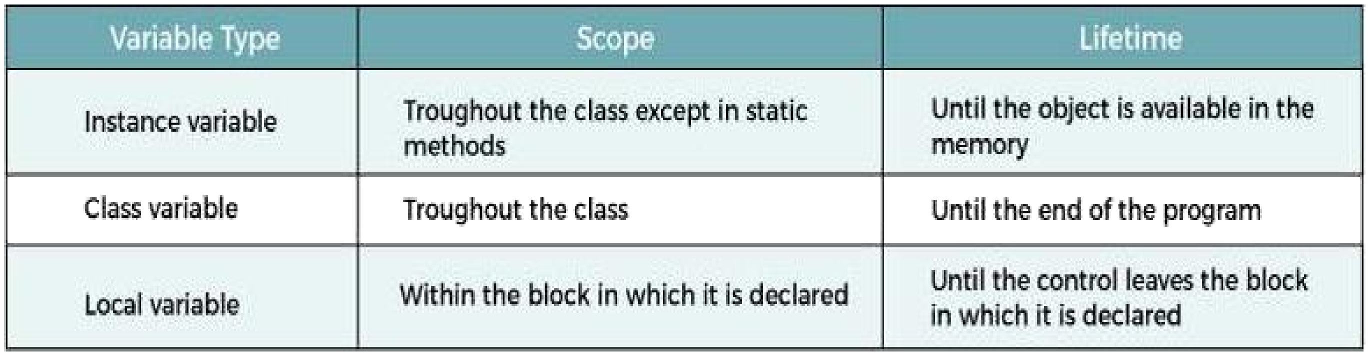
SCOPE AND LIFETIME OF A VARIABLE

- In programming, a variable can be declared and defined inside a class, method, or block.
- It defines the scope of the variable i.e. the visibility or accessibility of a variable.
- Variable declared inside a block or method are not visible to outside.
- If we try to do so, we will get a compilation error. Note that the scope of a variable can be nested.
- **Lifetime** of a variable indicates how long the variable stays alive in the memory.

TYPES OF VARIABLES IT'S SCOPE

There are three types of variables in Java:

1. local variable
2. instance variable
3. static variable



Variable Type	Scope	Lifetime
Instance variable	Troughout the class except in static methods	Until the object is available in the memory
Class variable	Troughout the class	Until the end of the program
Local variable	Within the block in which it is declared	Until the control leaves the block in which it is declared

Local Variables

- Variables declared inside the methods or constructors or blocks are called as local variables.
- The scope of local variables is within that particular method or constructor or block in which they have been declared.
- Local variables are allocated memory when the method or constructor or block in which they are declared is invoked and memory is released after that particular method or constructor or block is executed.
- Access modifiers cannot be assigned to local variables.
- It can't be defined by a static keyword.
- Local variables can be accessed directly with their name.

Program

```
class LocalVariables
{
        public void show()
        {
                int a = 10;
                System.out.println("Inside show method, a = " + a);
        }
        public void display()
        {
                int b = 20;
                System.out.println("Inside display method, b = " + b);
                //System.out.println("Inside display method, a = " + a); // error
        }
        public static void main(String args[])
        {
                LocalVariables obj = new LocalVariables();
                obj.show();
                obj.display();
        }
}
```

Instance Variables:

- Variables declared outside the methods or constructors or blocks but inside the class are called as instance variables.
- The scope of instance variables is inside the class and therefore all methods, constructors and blocks can access them.
- Instance variables are allocated memory during object creation and memory is released during object destruction. If no object is created, then no memory is allocated.
- For each object, a separate copy of instance variable is created.
- Heap memory is allocated for storing instance variables.
- Access modifiers can be assigned to instance variables.
- It is the responsibility of the JVM to assign default value to the instance variables as per the type of
- Variable.
- Instance variables can be called directly inside the instance area.
- Instance variables cannot be called directly inside the static area and necessarily requires an object reference for calling them.

Program

```
class InstanceVariable
{
        int x = 100;
        public void show()
        {
                System.out.println("Inside show method, x = " + x);
                x = x + 100;
        }
        public void display()
        {
                System.out.println("Inside display method, x = " + x);
        }
        public static void main(String args[])
        {
                ClassVariables obj = new ClassVariables();
                obj.show();
                obj.display();
        }
}
```

Static variables

- Static variables are also known as class variable.
- Static variables are declared with the keyword 'static ' .
- A static variable is a variable whose single copy in memory is shared by all the objects, any modification to it will also effect other objects.
- Static keyword in java is used for memory management, i.e it saves memory.
- Static variables gets memory only once in the class area at the time of class loading.
- Static variables can be invoked without the need for creating an instance of a class.
- Static variables contain values by default. For integers, the default value is 0. For Booleans, it is false. And for object references, it is null.

Syntax: static datatype variable name;

Example: static int x=100;

Syntax: classname.variablename;

Example

```
class Employee
{
   static int empid=500;
   static void emp1()
          {
                    empid++;
                    System.out.println("Employee  id:"+empid);
          }
}
class Sample
{
          public static void main(String args[])
          {
                    Employee.emp1();
                    Employee.emp1();
                    Employee.emp1();
                    Employee.emp1();
                    Employee.emp1();
                    Employee.emp1();

          }
}
```

ARRAYS

- An array is a collection of similar data values with a single name.
- An array can also be defined as, a special type of variable that holds multiple values of the same data type at a time.
- In java, arrays are objects and they are created dynamically using new operator.
- Every array in java is organized using index values.
- The index value of an array starts with '0' and ends with 'zise-1'.
- We use the index value to access individual elements of an array.

In java, there are two types of arrays and they are as follows.

- One Dimensional Array
- Multi Dimensional Array

One Dimensional Array

In the java programming language, an array must be created using new operator and with a specific size. The size must be an integer value but not a byte, short, or long. We use the following syntax to create an array.

Syntax

```
data_type array_name[ ] = new data_type[size];
                (or)
data_type[ ] array_name = new data_type[size];
```

Example

```
class Onedarray
{
        public static void main(String args[])
        {
                int a[]=new int[5];
                a[0]=10;
                a[1]=20;
                a[2]=70;
                a[3]=40;
                a[4]=50;
                for(int i=0;i<5;i++)
                System.out.println(a[i]);
        }
}
```

- In java, an array can also be initialized at the time of its declaration.
- When an array is initialized at the time of its declaration, it need not specify the size of the array and use of the new operator.
- Here, the size is automatically decided based on the number of values that are initialized.

Example

```
int list[ ] = {10, 20, 30, 40, 50};
```

Multidimensional Array

- In java, we can create an array with multiple dimensions. We can create 2-dimensional, 3-dimensional, or any dimensional array.
- In Java, multidimensional arrays are arrays of arrays.
- To create a multidimensional array variable, specify each additional index using another set of square brackets.

Syntax

```
data_type array_name[ ][ ] = new data_type[rows][columns];
                    (or)
data_type[ ][ ] array_name = new data_type[rows][columns];
```

- When an array is initialized at the time of declaration, it need not specify the size of the array and use of the new operator.
- Here, the size is automatically decided based on the number of values that are initialized.

Example

```
class Twodarray
{
        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();
                }
        }
}
```

OPERATORS

An operator is a symbol that performs an operation. An operator acts on some variables called operands to get the desired result.

Example: a + b

Here a, b are operands and + is operator.

Types of Operators

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment or Decrement operators
6. Conditional operator
7. Bit wise operators

1. Arithmetic Operators: Arithmetic Operators are used for mathematical calculations.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modular

Program: Java Program to implement Arithmetic Operators

```
class ArithmeticOperators
{
        public static void main(String[] args)
                {
                        int a = 12, b = 5;
                        System.out.println("a + b = " + (a + b));
                        System.out.println("a - b = " + (a - b));
                        System.out.println("a * b = " + (a * b));
                        System.out.println("a / b = " + (a / b));
                        System.out.println("a % b = " + (a % b));
                }
}
```

2. Relational Operators: Relational operators are used to compare two values and return a true or false result based upon that comparison. Relational operators are of 6 types

Operator	Description
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to

Program: Java Program to implement Relational Operators

```
class  RelationalOperator
{
        public static void main(String[] args)
        {
                int a = 10;
                int b = 3;
                int c = 5;
                System.out.println("a > b: " + (a > b));
                System.out.println("a < b: " + (a < b));
                System.out.println("a >= b: " + (a >= b));
                System.out.println("a <= b: " + (a <= b));
                System.out.println("a == c: " + (a == c));
                System.out.println("a != c: " + (a != c));
        }
}
```

3. Logical Operator: The Logical operators are used to combine two or more conditions .Logical operators are of three types

1. Logical AND (&&),
2. Logical OR (||)
3. Logician NOT (!)

1. Logical AND (&&) : Logical AND is denoted by double ampersand characters (&&).it is used to check the combinations of more than one conditions. if any one condition false the complete condition becomes false.

Truth table of Logical AND

Condition1	Condition2	Condition1 && Condition2
True	True	True
True	False	False
False	True	False
False	False	False

2. Logical OR (||) : Logical OR is denoted by double pipe characters (||). it is used to check the combinations of more than one conditions. if any one condition true the complete condition becomes true.

Truth table of Logical OR

Condition1	Condition2	Condition1 && Condition2
True	True	True
True	False	True
False	True	True
False	False	False

3. Logician NOT (!): Logical NOT is denoted by exclamatory characters (!), it is used to check the opposite result of any given test condition. i.e, it makes a true condition false and false condition true.

Truth table of Logical NOT

Condition1	!Condition2
True	False
False	True

Example of Logical Operators

```
class  LogicalOp
{
        public static void main(String[] args)
        {
            int x=10;
            System.out.println(x==10 && x>=5));
            System.out.println(x==10 || x>=5));
            System.out.println ( ! ( x==10 ));
        }
}
```

4. Assignment Operator: Assignment operators are used to assign a value (or) an expression (or) a value of a variable to another variable.

Syntax : variable name=expression (or) value

Example : x=10;
y=20;

The following list of Assignment operators are.

Operator	Description	Example	Meaning
+=	Addition Assignment	x + = y	x= x + y
-=	Addition Assignment	x - = y	x= x - y
***=**	Addition Assignment	x * = y	x= x * y
/=	Addition Assignment	x / = y	x= x / y
%=	Addition Assignment	x % = y	x= x % y

Example of Assignment Operators

```
class AssignmentOperator
{
        public static void main(String[] args)
        {
                int a = 4;
                int var;
                var = a;
                System.out.println("var using =: " + var);
                 var += a;
                System.out.println("var using +=: " + var);
                 var *= a;
                System.out.println("var using *=: " + var);
        }
 }
```

5: Increment And Decrement Operators : The increment and decrement operators are very useful. ++ and == are called increment and decrement operators used to add or subtract. Both are unary operators.

The syntax of the operators is given below.

These operators in two forms : prefix (++x) and postfix(x++).

++<variable name> --<variable name>

<variable name>++ <variable name>--

Operator	Meaning
++x	Pre Increment
--x	Pre Decrement
x++	Post Increment
x--	Post Decrement

Where

1 : ++x : Pre increment, first increment and then do the operation.

2 : - -x : Pre decrement, first decrements and then do the operation.

3 : x++ : Post increment, first do the operation and then increment.

4 : x- - : Post decrement, first do the operation and then decrement.

Example

```
class Increment
 {
        public static void main(String[] args)
        {
                int var=5;
                System.out.println (var++);
                System.out.println (++var);
                System.out.println (var--);
                System.out.println (--var);
        }
}
```

6 : Conditional Operator: A conditional operator checks the condition and executes the statement depending on the condition. Conditional operator consists of two symbols.

1 : question mark (?).

2 : colon (:).

Syntax: condition ? exp1 : exp2;

It first evaluate the condition, if it is true (non-zero) then the "exp1" is evaluated, if the condition is false (zero) then the "exp2" is evaluated.

Example :

```
class  ConditionalOperator
{
        public static void main(String[] args)
        {
             int februaryDays = 29;
            String result;
            result = (februaryDays == 28) ? "Not a leap year" : "Leap year";
            System.out.println(result);
        }
}
```

7. Bitwise Operators:

- Bitwise operators are used for manipulating a data at the bit level, also called as bit level programming. Bit-level programming mainly consists of 0 and 1.
- They are used in numerical Computations to make the calculation process faster.
- The bitwise logical operators work on the data bit by bit.
- Starting from the least significant bit, i.e. LSB bit which is the rightmost bit, working towards the MSB (Most Significant Bit) which is the leftmost bit.

A list of Bitwise operators as follows…

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise Complement
<<	Left Shift
>>	Right Shift

1. Bitwise AND (&):

- Bitwise AND operator is represented by a single ampersand sign (&).
- Two integer expressions are written on each side of the (&) operator.
- if any one condition false (0) the complete condition becomes false (0).

Truth table of Bitwise AND

Condition1	Condition2	Condition1 & Condition2
0	0	0
0	1	0
1	0	0
1	1	1

Example : int x = 10;

int y = 20;

x & y = ?

x = 0000 1010

y = 0000 1011

x & y = 0000 1010 = 10

2. Bitwise OR:

- Bitwise OR operator is represented by a single vertical bar sign (|).
- Two integer expressions are written on each side of the (|) operator.
- if any one condition true (1) the complete condition becomes true (1).

Truth table of Bitwise OR

Condition1	Condition2	Condition1 | Condition2
0	0	0
0	1	1
1	0	1
1	1	1

Example : int x = 10;

int y = 20;

x | y = ?

x = 0000 1010

y = 0000 1011

x | y = 0000 1011 = 11

3. Bitwise Exclusive OR :

- The XOR operator is denoted by a carrot (^) symbol.
- It takes two values and returns true if they are different; otherwise returns false.
- In binary, the true is represented by 1 and false is represented by 0.

Truth table of Bitwise XOR

Condition1	Condition2	Condition1 ^ Condition2
0	0	0
0	1	1
1	0	1
1	1	0

Example : int x = 10;

int y = 20;

x ^ y = ?

x = 0000 1010

y = 0000 1011

x ^ y = 0000 0001 = 1

4. Bitwise Complement (~):

- The bitwise complement operator is a unary operator.
- It is denoted by ~, which is pronounced as tilde.
- It changes binary digits 1 to 0 and 0 to 1.
- bitwise complement of any integer N is equal to - (N + 1).
- Consider an integer 35. As per the rule, the bitwise complement of 35 should be -(35 + 1) = -36.

Example : int x = 10; find the ~x value.

x = 0000 1010

~x= 1111 0101

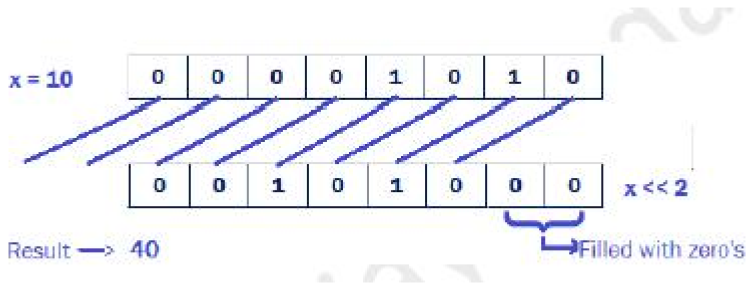
5. Bitwise Left Shift Operator (<<) :

- This Bitwise Left shift operator (<<) is a binary operator.
- It shifts the bits of a number towards left a specified no.of times.

Example:

int x = 10;

x << 2 = ?



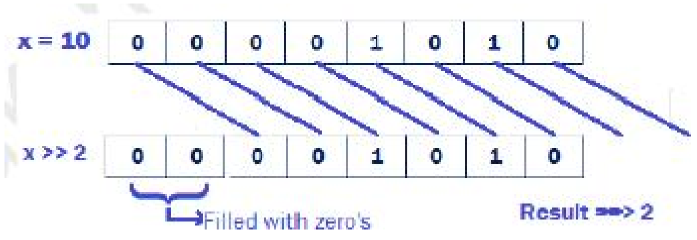
6. Bitwise Right Shift Operator (>>) :

- This Bitwise Right shift operator (>>) is a binary operator.
- It shifts the bits of a number towards right a specified no.of times.

Example:

int x = 10;

x >> 2 = ?



EXPRESSIONS

- In any programming language, if we want to perform any calculation or to frame any condition etc., we use a set of symbols to perform the task. These set of symbols makes an expression. **In the java programming language, an expression is defined as follows..**
- An expression is a collection of operators and operands that represents a specific value.
- In the above definition, an operator is a symbol that performs tasks like arithmetic operations, logical operations, and conditional operations, etc.

Expression Types

In the java programming language, expressions are divided into THREE types. They are as follows.

- **Infix Expression**
- **Postfix Expression**
- **Prefix Expression**

The above classification is based on the operator position in the expression.

Infix Expression

The expression in which the operator is used between operands is called infix expression. The infix expression has the following general structure.

Example

a+b

Postfix Expression

The expression in which the operator is used after operands is called postfix expression. The postfix expression has the following general structure.

Example

ab+

Prefix Expression

The expression in which the operator is used before operands is called a prefix expression. The prefix expression has the following general structure.

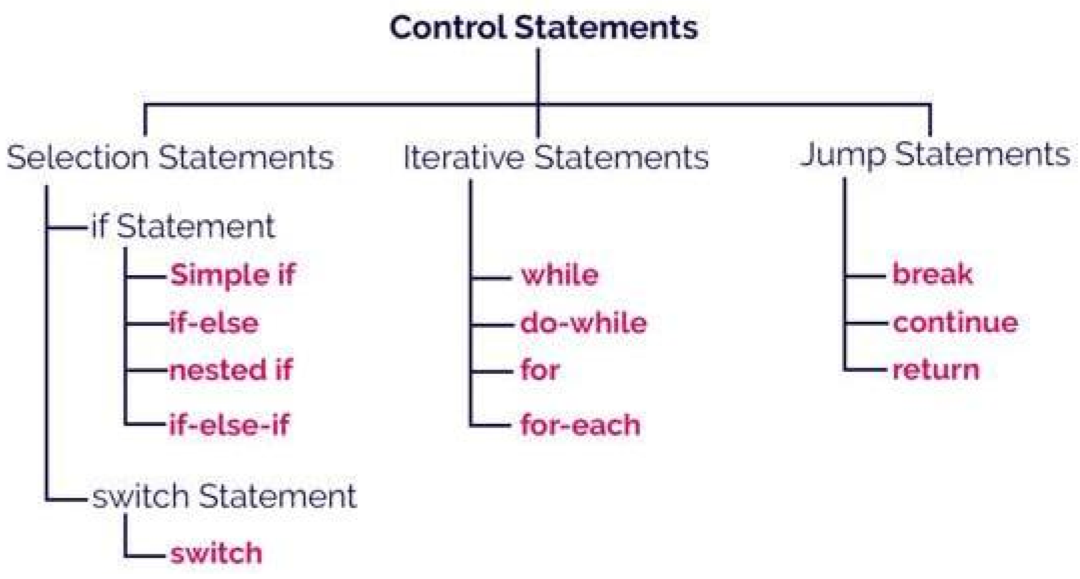
Example

+ab

CONTROL STATEMENTS

- In java, the default execution flow of a program is a sequential order.
- But the sequential order of execution flow may not be suitable for all situations.
- Sometimes, we may want to jump from line to another line, we may want to skip a part of the program, or sometimes we may want to execute a part of the program again and again.
- To solve this problem, java provides control statements.

Types of Control Statements



1. Selection Control Statements

In java, the selection statements are also known as decision making statements or branching statements. The selection statements are used to select a part of the program to be executed based on a condition.

Java provides the following selection statements.

- if statement
- if-else statement
- if-elif statement
- nested if statement
- switch statement

if statement in java

- In java, we use the if statement to test a condition and decide the execution of a block of statements based on that condition result.
- The if statement checks, the given condition then decides the execution of a block of statements. 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(condtion)
{
        if-block of statements;
}
statement after if-block;
```

Example

```
public class IfStatementTest
{
        public static void main(String[] args)
       {
                int x=10;
                if(x>0)
                x++;
                System.out.println("x value is:"+x);
        }
}
```

In the above execution, the number 12 is not divisible by 5. So, the condition becomes False and the condition is evaluated to False. Then the if statement ignores the execution of its block of statements.

if-else statement in java

- In java, we use the if-else statement to test a condition and pick the execution of a block of statements out of two blocks based on that condition result.
- The if-else statement checks the given condition then decides which block of statements to be executed based on the condition result.
- If the condition is True, then the true block of statements is executed and if it is False, then the false block of statements is executed.

Syntax

```
if(condtion)
{
        true-block of statements;
}
else
{
        false-block of statements;
}
statement after if-block;
```

Example

```
public class IfElseStatementTest
{
        public static void main(String[] args)
        {
                int a=29;
                if(a % 2==0)
                        System.out.println("Even Number is :"+a);
                else
                        System.out.println("Odd Number is :"+a);
        }
}
```

Nested if statement in java

Writing an if statement inside another if-statement is called nested if statement.

Syntax

```
if(condition_1)
{
   if(condition_2)
   {
      inner if-block of statements;
      ...
   }
   ...
}
```

Example

```
public class NestedIfStatementTest
{
        public static void main(String[] args)
        {
                int num=1;
                if(num<10)
                {
                        if(num==1)
                        {
                                System.out.print("The value is equal to 1);
                        }
                        else
                        {
                                System.out.print("The value is greater than 1");
                        }
                }
                else
                {
                        System.out.print("The value is greater than 10");
                }
                System.out.print("Nested if - else statement ");
        }
}
```

if-else if statement in java

Writing an if-statement inside else of an if statement is called if-else-if statement.

Syntax

```
if(condition_1)
{
    condition_1 true-block;
                ...
}
else if(condition_2)
{
                condition_2 true-block;
                condition_1 false-block too;
                ...
}
```

Example

```
public class IfElseIfStatementTest
{
        public static void main(String[] args)
        {
                int x = 30;
                if( x == 10 )
                {
                        System.out.print("Value of X is 10");
                }
                else if( x == 20 )
                {
                        System.out.print("Value of X is 20");
                }
                else if( x == 30 )
                {
                        System.out.print("Value of X is 30");
                }
                else
                {
                        System.out.print("This is else statement");
                }
        }
}
```

Switch

- Using the switch statement, one can select only one option from more number of options very easily.
- In the switch statement, we provide a value that is to be compared with a value associated with each option. Whenever the given value matches the value associated with an option, the execution starts from that option.
- In the switch statement, every option is defined as a **case**.

Syntax:

```
switch (expression)
{
          case value1: // statement sequence
                   break;
          case value2: // statement sequence
                  break;
            ....
           case valueN:
}
```

Example

```
class SampleSwitch
{
            public static void main(String args[])
            {
                        char color ='g';
                        switch(color )
                        {
                                    case 'r':
                                                System.out.println("RED") ; break ;
                                    case 'g':
                                                System.out.println("GREEN") ; break ;
                                    case 'b':
                                                System.out.println("BLUE") ; break ;
                                    case 'w':
                                                System.out.println("WHITE") ; break ;
                                    default:
                                                System.out.println("No color") ;
                         }
          }
  }
```

2. Iteration Statements

- The java programming language provides a set of iterative statements that are used to execute a statement or a block of statements repeatedly as long as the given condition is true.
- The iterative statements are also known as looping statements or repetitive statements. Java provides the following iterative statements.
    1. while statement
    2. do-while statement
    3. for statement
    4. for-each statement

while statement in java

The while statement is used to execute a single statement or block of statements repeatedly as long as the given condition is TRUE. The while statement is also known as Entry control looping statement.

Syntax

```
while(condition)
{
     // body of loop
}
```

Example

```
public class WhileTest
{
        public static void main(String[] args)
        {
                int num = 1;
                while(num <= 10)
                {
                        System.out.println(num);
                        num++;
                }
                System.out.println("Statement after while!");
        }
}
```

do-while statement in java

- The do-while statement is used to execute a single statement or block of statements repeatedly as long as given the condition is TRUE.
- The do-while statement is also known as the Exit control looping statement.

Syntax

```
do
{
    // body of loop
} while (condition);
```

Example

```
public class DoWhileTest
{
        public static void main(String[] args)
        {
                int num = 1;
                do
              {
                        System.out.println(num);
                        num++;
                }while(num <= 10);
                System.out.println("Statement after do-while!");
         }
}
```

for statement in java

The for statement 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; inc/dec)
        {
             // body
        }
```

If only one statement is being repeated, there is no need for the curly braces.

In for-statement, the execution begins with the **initialization** statement. After the initialization statement, it executes **Condition.** If the condition is evaluated to true, then the block of statements executed otherwise it terminates the for-statement. After the block of statements execution, the **modification** statement gets executed, followed by condition again.

Example

```
public class ForTest
{
        public static void main(String[] args)
       {
                for(int i = 0; i < 10; i++)
              {
                        System.out.println("i = " + i);
                }
                System.out.println("Statement after for!");
        }
}
```

3. Jump Statements

The java programming language supports jump statements that used to transfer execution control from one line to another line.

The java programming language provides the following jump statements.

1. break statement
2. continue statement

break

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

Example

```
class BreakStatement
{
        public static void main(String args[] )
        {
                int i;
                i=1;
                while(true)
                {
                        if(i >10)
                        break;
                        System.out.print(i+" ");
                        i++;
                }
        }
}
```

Continue

This command skips the whole body of the loop and executes the loop with the next iteration. On finding continue command, control leaves the rest of the statements in the loop and goes back to the top of the loop to execute it with the next iteration (value).

Example

```
/* Print Number from 1 to 10 Except 5 */
class NumberExcept
{
        public static void main(String args[] )
        {
                int i;
                for(i=1;i<=10;i++)
                {
                        if(i==5)
                           continue;
                        System.out.print(i +" ");
                }
        }
}
```

TYPE CONVERSION AND CASTING

Type Casting

- When a data type is converted into another data type by a programmer using the casting operator while writing a program code, the mechanism is known as **type casting**.
- In typing casting, the destination data type may be smaller than the source data type when converting the data type to another data type, that's why it is also called **narrowing conversion**.

Syntax

```
destination_datatype = (target_datatype)variable;
```

(): is a casting operator.

target_datatype : is a data type in which we want to convert the source data type.

Example

```
float x;
byte y;
y=(byte)x;
```

Program

```
public class NarrowingTypeCastingExample
{
    public static void main(String args[])
    {
        double d = 166.66;
        int i = (int)d;
        System.out.println("Before conversion: "+d);
        System.out.println("After conversion into int type: "+i);
    }
}
```

Output

```
Before conversion: 166.66
After conversion into int type: 166
```

Type Conversion

- If a data type is automatically converted into another data type at compile time is known as type conversion.
- The conversion is performed by the compiler if both data types are compatible with each other.
- Remember that the destination data type should not be smaller than the source type.
- It is also known as **widening** conversion of the data type.

Example

```
int a = 20;
Float b;
b = a;      // Now the value of variable b is 20.000
```

Program

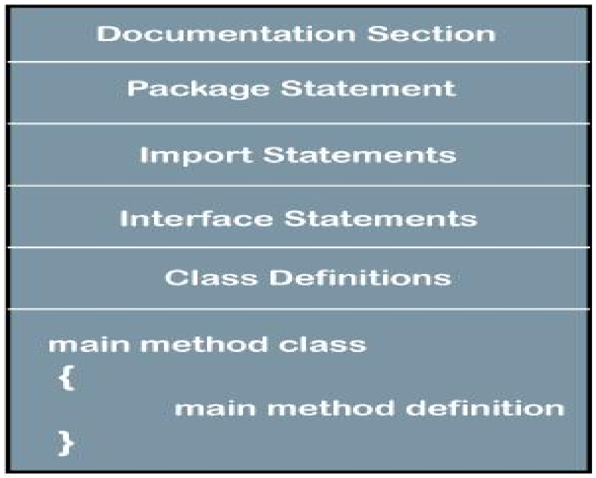
```
public class WideningTypeCastingExample
{
    public static void main(String[] args)
    {
            int x = 7;
            float y = x;
            System.out.println("After conversion, float value "+y);
}
}
```

Output :

```
After conversion, the float value is: 7.0
```

STRUCTURE OF JAVA PROGRAM

Structure of a Java program contains the following elements:



Structure of Java Program

Documentation Section

The documentation section is an important section but optional for a Java program.

It includes **basic information** about a Java program. The information includes the **author's name, date of creation, version, program name, company name,** and **description** of the program. It improves the readability of the program. Whatever we write in the documentation section, the Java compiler ignores the statements during the execution of the program. To write the statements in the documentation section, we use **comments**.

Comments there are three types

1. Single-line Comment: It starts with a pair of forwarding slash **(//)**.

Example : //First Java Program

2. Multi-line Comment: It starts with a **/*** and ends with ***/.** We write between these two symbols.

Example : /* It is an example of
multiline comment */

3. Documentation Comment: It starts with the delimiter **(/**)** and ends with ***/**.

SAMPLE JAVA PROGRAM

```
/* This is First Java Program */
                Class sample
                {
                        public static void main(String args[])
                        {
                                System.out.println("Hello Java Programming");
                        }
                }
```

Parameters used in First Java Program

Let's see what is the meaning of class, public, static, void, main, String[], System.out.println().

class keyword is used to declare a class in java.

public keyword is an access modifier which 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 to create 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 execution

String[] args is used for command line argument.

System.out.println() is used to print statement. Here, System is a class, out is the object of PrintStream class, println() is the method of PrintStream class.

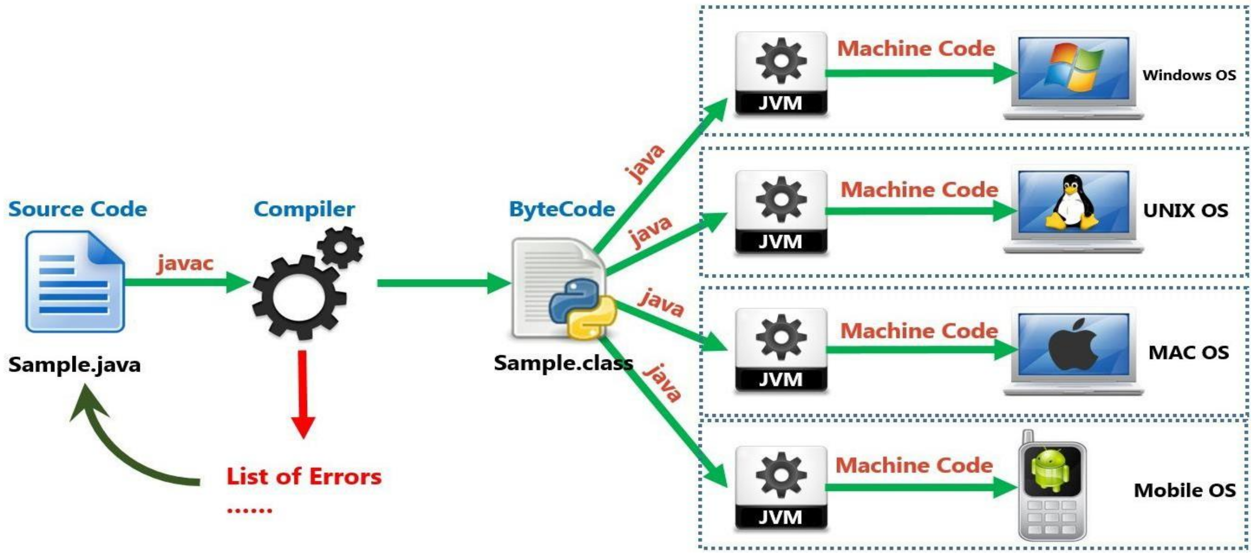
How to Compile and Run the Java Program

To Compile : javac Sample.java [program name]

To Run : java Sample

Output : Hello Java Programming

EXECUTION PROCESS OF JAVA PROGRAM



WHAT IS JVM

Java Virtual Machine is the heart of entire java program execution process. It is responsible for taking the .class file and converting each byte code instruction into the machine language instruction that can be executed by the microprocessor.

CLASSES AND OBJECTS IN JAVA

CLASSES

- In Java, classes and objects are basic concepts of Object Oriented Programming (OOPs) that are used to represent real-world concepts and entities.
- classes usually consist of two things: instance variables and methods.
- The class represents a group of objects having similar properties and behavior.
- For example, the animal type **Dog** is a class while a particular dog named **Tommy** is an object of the **Dog** class.
- It is a user-defined blueprint or prototype from which objects are created. For example, Student is a class while a particular student named Ravi is an object.
- The java class is a template of an object.
- Every class in java forms a new data type.
- Once a class got created, we can generate as many objects as we want.

Class Characteristics

Identity - It is the name given to the class.

State - Represents data values that are associated with an object.

Behavior - Represents actions can be performed by an object.

Properties of Java Classes

1. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
2. Class does not occupy memory.
3. Class is a group of variables of different data types and a group of methods.
4. A Class in Java can contain:
    - Data member
    - Method
    - Constructor
    - Nested Class
    - Interface

Creating a Class

In java, we use the keyword class to create a class. A class in java contains properties as variables and behaviors as methods.

Syntax

```
class className
{
        data members declaration;
        methods definition;
}
```

- The ClassName must begin with an alphabet, and the Upper-case letter is preferred.
- The ClassName must follow all naming rules.

Example

Here is a class called Box that defines three instance variables: width, height, and depth.

```
class Box
{
    double  width;
    double  height;
    double  depth;
    void volume()
    {
        ………………….
    }
}
```

OBJECT

- In java, an object is an instance of a class.
- Objects are the instances of a class that are created to use the attributes and methods of a class.
- All the objects that are created using a single class have the same properties and methods. But the value of properties is different for every object.

Syntax

```
ClassName objectName = new ClassName( );
```

- The objectName must begin with an alphabet, and a Lower-case letter is preferred.
- The objectName must follow all naming rules.

Example

```
Box  mybox = new Box();
```

The new operator dynamically allocates memory for an object.

Example

```
class Box
{
      double  width;
      double height;
      double depth;
}
class BoxDemo
{
    public static void main(String args[])
    {
           Box mybox = new Box();
           double vol;
           mybox.width = 10;
           mybox.height = 20;
           mybox.depth = 15;
           vol = mybox.width * mybox.height * mybox.depth;
           System.out.println("Volume is " + vol);
    }
}
```

METHODS

- A method is a block of statements under a name that gets executes only when it is called.
- Every method is used to perform a specific task. The major advantage of methods is code re-usability (define the code once, and use it many times).
- In a java programming language, a method defined as a behavior of an object. That means, every method in java must belong to a class.
- Every method in java must be declared inside a class.

Every method declaration has the following characteristics.

- **returnType** - Specifies the data type of a return value.
- **name** - Specifies a unique name to identify it.
- **parameters** - The data values it may accept or recieve.
- **{ }** - Defienes the block belongs to the method.

Creating a method

A method is created inside the class

Syntax

```
class ClassName
{
    returnType methodName( parameters )
    {
        // body of method
    }
}
```

Calling a method

- In java, a method call precedes with the object name of the class to which it belongs and a dot operator.
- It may call directly if the method defined with the static modifier.
- Every method call must be made, as to the method name with parentheses (), and it must terminate with a semicolon.

Syntax

```
objectName.methodName(actualArguments );
```

Example

```
//Adding a Method to the Box Class
```

```
Class Box
{
        double width, height, depth;
        void volume()
        {
                System.out.print("Volume is ");
                System.out.println(width * height * depth);
        }
}
class BoxDemo3
{
        public static void main(String args[])
        {
                Box mybox1 = new Box();
                Box mybox2 = new Box();
                mybox1.width = 10;
                mybox1.height = 20;
                mybox1.depth = 15;
                mybox2.width = 3;
                mybox2.height = 6;
                mybox2.depth = 9;
                mybox1.volume();
                mybox2.volume();
         }
  }
```

CONSTRUCTORS

- Constructor in Java is a special member method which will be called automatically by the JVM whenever an object is created for placing user defined values in place of default values.
- In a single word constructor is a special member method which will be called automatically whenever object is created.
- The purpose of constructor is to initialize an object called object initialization. Initialization is a process of assigning user defined values at the time of allocation of memory space.

Syntax

```
ClassName()
{
.......
.......
}
```

Types Of Constructors

Based on creating objects in Java constructor are classified in two types. They are

1. Default or no argument Constructor
2. Parameterized constructor

1. Default Constructor

- A constructor is said to be default constructor if and only if it never take any parameters.
- If any class does not contain at least one user defined constructor then the system will create a default constructor at the time of compilation it is known as system defined default constructor.

Note: System defined default constructor is created by java compiler and does not have any statement in the body part. This constructor will be executed every time whenever an object is created if that class does not contain any user defined constructor.

Example

```
class Test
{
        int a, b;
        Test()
        {
                a=10;
                b=20;
```

```
                        System.out.println("Value of a: "+a);
                        System.out.println("Value of b: "+b);
                    }
        }
        class TestDemo
        {
                public static void main(String args[])
                {
                        Test t1=new Test();
                }
        }
```

2. Parameterized Constructor

If any constructor contain list of variables in its signature is known as paremetrized constructor. A parameterized constructor is one which takes some parameters.

Example

```
class Test
{
        int a, b;
        Test(int n1, int n2)
        {
                a=n1;
                b=n2;
                System.out.println("Value of a = "+a);
                System.out.println("Value of b = "+b);
        }
}
class TestDemo
{
        public static void main(String args[])
        {
                Test t1=new Test(10, 20);
        }
}
```

ACCESS CONTROL(MEMBER ACCESS)

In Java, Access modifiers help to restrict the scope of a class, constructor, variable, method, or data member. It provides security, accessibility, etc to the user depending upon the access modifier used with the element.

Types of Access Modifiers in Java

There are four types of access modifiers available in Java:

1. Default – No keyword required
2. Private
3. Protected
4. Public

1. Default Access Modifier

- When no access modifier is specified for a class, method, or data member – It is said to be having the **default** access modifier by default.
- The default modifier is accessible only within package.
- It cannot be accessed from outside the package.
- It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

//save by A.java

```
package pack;
class A
{
        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();  //Compile Time Error
                obj.msg();        //Compile Time Error
        }
}
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

2. Private

- The private access modifier is accessible only within the class.
- The private access modifier is specified using the keyword **private**.
- The methods or data members declared as private are accessible only **within the class** in which they are declared.
- Any other **class of** the **same package will not be able to access** these members.
- Top-level classes or interfaces can not be declared as private because private means "only visible within the enclosing class".

Example

- In this example, we have created two classes A and Simple.
- A class contains private data member and private method.
- We are accessing these private members from outside the class, so there is a compile-time error.

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

3. Protected

- The protected access modifier is accessible within package and outside the package but through inheritance only.
- The protected access modifier is specified using the keyword **protected**.

Example

- In this example, we have created the two packages pack and mypack.
- The A class of pack package is public, so can be accessed from outside the package.
- But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

//save by A.java

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

//save by B.java

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

4. Public

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

Example

//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();
        }
}
```

Table: Class Member Access

Let's understand the access modifiers in Java by a simple table. Access Modifier	**within class**	**within package**	**outside package by subclass only**	**outside package**
Private	YES	NO	NO	NO
Default	YES	YES	NO	NO
Protected	YES	YES	YES	NO
Public	YES	YES	YES	YES

‘this’ KEYWORD IN JAVA

this is a reference variable that refers to the current object. It is a keyword in java language represents current class object

Why use this keyword in java ?

- The main purpose of **using this keyword** is to differentiate the formal parameter and data members of class, whenever the formal parameter and data members of the class are similar then JVM get ambiguity (no clarity between formal parameter and member of the class).
- To differentiate between formal parameter and data member of the class, the data member of the class must be preceded by "this".

Syntax: this.data member of current class.

Example without using this keyword

```
class Employee
{
        int id;
        String name;
        Employee(int id,String name)
        {
                id = id;
                name = name;
        }
        void show()
        {
                System.out.println(id+"  "+name);
        }
}
class ThisDemo1
{
        public static void main(String args[])
        {
                Employee e1 = new Employee(111,"Harry");
                e1.show();
        }
}
```

Output: 0 null

In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

Example of this keyword in java

```
class Employee
{
        int id;
        String name;
        Employee(int id,String name)
        {
                this.id = id;
                this.name = name;
        }
void show()
{
        System.out.println(id+"  "+name);
}
class ThisDemo2
{
        public static void main(String args[])
        {
                Employee e1 = new Employee(111,"Harry");
                e1.show();
        }
}
```

Output: 111 Harry

GARBAGE COLLECTION IN JAVA

Garbage collection in Java is the process by which Java programs perform automatic memory management.

How Does Garbage Collection in Java works?

- Java garbage collection is an automatic process.
- Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects.
- An unused or unreferenced object is no longer referenced by any part of your program.
- So the memory used by an unreferenced object can be reclaimed.
- The programmer does not need to mark objects to be deleted explicitly.
- The garbage collection implementation lives in the JVM.

Advantage of Garbage Collection

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

How Can an Object be Unreferenced?

There are many ways:

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

1. By nulling a reference:

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

2. By assigning a reference to another:

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

3. By anonymous object:

```
new Employee();
```

OVERLOADING METHODS AND CONSTRUCTORS

OVERLOADING METHODS

- Whenever same method name is exiting multiple times in the same class with different number of parameter or different order of parameters or different types of parameters is known as method overloading.
- Method overloading in Java is also known as Compile-time Polymorphism, Static Polymorphism, or Early binding**.**

Example

```
class Addition
{
        void sum(int a, int b)
        {
        System.out.println(a+b);
        }
        void sum(int a, int b, int c)
        {
        System.out.println(a+b+c);
        }
        void sum(float a, float b)
        {
        System.out.println(a+b);
        }
}
class Methodload
{
     public static void main(String args[])
     {
             Addition obj=new Addition();
             obj.sum(10, 20);
             obj.sum(10, 20, 30);
             obj.sum(10.05f, 15.20f);
     }
}
```

OVERLOADING CONSTRUCTORS

Constructor overloading is a concept of having more than one constructor with different parameters list, so that each constructor performs a different task.

Example

```
public class Person
{
        Person()
        {
                System.out.println("Introduction:");
        }
        Person(String name)
        {
                System.out.println("Name: " +name);
        }
        Person(String scname, int rollNo)
         {
                 System.out.println("School name: "+scname+ ", "+"Roll no:"+rollNo);
         }
        public static void main(String[] args)
        {
                 Person p1 = new Person();
                 Person p2 = new Person("John");
                 Person p3 = new Person("ABC", 12);
        }
}
```

METHOD BINDING

Connecting a method call to the method body is known as binding.

There are two types of binding

1. Static Binding (also known as Early Binding).
2. Dynamic Binding (also known as Late Binding).

Static Binding

When type of the object is determined at compiled time(by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

Example

```
class Dog
{
    private void eat(){System.out.println("dog is eating...");}
    public static void main(String args[])
    {
            Dog d1=new Dog();
            d1.eat();
    }
}
```

Dynamic binding

When type of the object is determined at run-time, it is known as dynamic binding.

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 a=new Dog();
            a.eat();
     }
}
```

In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal. So compiler doesn't know its type, only its base type.

PARAMETER PASSING METHODS

Parameter passing in Java refers to the mechanism of transferring data between methods or functions. Java supports two types of parameters passing techniques

1. Call-by-value
2. Call-by-reference.

1. Call-by-Value

In Call-by-value the copy of the value of the actual parameter is passed to the formal parameter of the method. Any of the modifications made to the formal parameter within the method do not affect the actual parameter.

Example

```
public class CallByValueExample
{
    public static void main(String[] args)
    {
        int num = 10;
        System.out.println("Before calling method:"+num);
        modifyValue(num);
        System.out.println("After calling method:"+num);
    }
    public static void modifyValue(int value)
    {
        value=20;
        System.out.println("Inside method:"+value);
    }
}
```

Output:

```
Before calling method: 10
Inside method: 20
After calling method: 10
```

Call-by-Reference

call by reference" is a method of passing arguments to functions or methods where the memory address (or reference) of the variable is passed rather than the value itself. This means that changes made to the formal parameter within the function affect the actual parameter in the calling environment.

In "call by reference," when a reference to a variable is passed, any modifications made to the parameter inside the function are transmitted back to the caller. This is because the formal parameter receives a reference (or pointer) to the actual data.

Example

```
class CallByReference
{
            int a,b;
            CallByReference(int x,int y)
            {
                  a=x;
                  b=y;
            }
             void changeValue(CallByReference obj)
            {
                  obj.a+=10;
                  obj.b+=20;
            }
}
public class CallByReferenceExample
{
    public static void main(String[] args)
    {
                  CallByReference object=new CallByReference(10, 20);
                  System.out.println("Value of a: "+object.a +" & b: " +object.b);
                  object.changeValue(object);
                  System.out.println("Value of a:"+object.a+ " & b: "+object.b);
    }
}
```

Output:

```
Value of a: 10 & b: 20
Value of a: 20 & b: 40
```

RECURSION IN JAVA

Recursion in java is a process in which a method calls itself continuously. A method in java that calls itself is called recursive method. It makes the code compact but complex to understand.

Syntax:

```
returntype methodname()
{
        methodname();
}
```

Example

```
public class RecursionExample3
{
     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 5 is: "+factorial(5));
    }
}
```

INNER CLASSES

- Inner class means one class which is a member of another class.
- We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

Syntax of Inner class

```
class Outer_class
{
    //code
    class Inner_class
    {
        //code
    }
}
```

Types of Inner classes

There are four types of inner classes.

1. Member Inner class
2. Local inner classes
3. Anonymous inner classes
4. Static nested classes

1. MEMBER INNER CLASS

A non-static class that is created inside a class but outside a method is called member inner class.

Syntax:

```
class Outer
{
    //code
    class Inner
    {
        //code
    }
}
```

Example

```
class TestMemberOuter
{
        private int data=30;
        class Inner
        {
         void msg()
         {
              System.out.println("data is "+data);
        }
}
public static void main(String args[])
{
        TestMemberOuter obj=new TestMemberOuter();
        TestMemberOuter.Inner in=obj.new Inner();
        in.msg();
}
}
```

2. ANONYMOUS INNER CLASS

- In Java, a class can contain another class known as nested class. It's possible to create a nested class without giving any name.
- A nested class that doesn't have any name is known as an anonymous class.
- An anonymous class must be defined inside another class. Hence, it is also known as an anonymous inner class.

Example

```
abstract class Person
{
      abstract void eat();
}
class TestAnonymousInner
{
          public static void main(String args[])
```

```
                {
                        Person p=new Person()
                        {
                            void eat()
                            {
                                System.out.println("nice  fruits");
                            }
                        };
                        p.eat();
                }
        }
```

1. A class is created, but its name is decided by the compiler, which extends the Person class and provides the implementation of the eat() method.
2. An object of the Anonymous class is created that is referred to by 'p,' a reference variable of Person type.

3. LOCAL INNER CLASS

- A class i.e. created inside a method is called local inner class in java.
- If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

Example

```
public class localInner
{
            private int data=30;
            void display()
            {
                class Local
                {
                        void msg()
                        {
                            System.out.println(data);
                        }
                }
                Local l=new Local();
                l.msg();
            }
            public static void main(String args[])
            {
                        localInner obj=new localInner();
                        obj.display();
            }
}
```

4. STATIC NESTED CLASS

- A static class i.e. created inside a class is called static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name.
- It can access static data members of outer class including private.
- Static nested class cannot access non-static (instance) data member or method.

Example

```
class TestOuter
{
        static int data=30;
        static class Inner
        {
              void msg()
              {
               System.out.println("data is "+data);
               }
        }
        public static void main(String args[])
        {
            TestOuter.Inner obj=new TestOuter.Inner();
            obj.msg();
        }
 }
```

In this example, you need to create the instance of static nested class because it has instance method msg(). But you don't need to create the object of Outer class because nested class is static and static properties, methods or classes can be accessed without object.

EXPLORING STRING CLASS

- A string is a sequence of characters surrounded by double quotations. In a java programming language, a string is the object of a built-in class **String**.
- The string created using the **String** class can be extended. It allows us to add more characters after its definition, and also it can be modified.

Example

```
String siteName = "javaprogramming";
siteName = "javaprogramminglanguage";
```

String handling methods

In java programming language, the String class contains various methods that can be used to handle string data values.

The following table depicts all built-in methods of String class in java.

S.No	Method	Description
1	charAt(int)	Finds the character at given index
2	length()	Finds the length of given string
3	compareTo(String)	Compares two strings
4	compareToIgnoreCase(String)	Compares two strings, ignoring case
5	concat(String)	Concatenates the object string with argument string.
6	contains(String)	Checks whether a string contains sub-string
7	contentEquals(String)	Checks whether two strings are same
8	equals(String)	Checks whether two strings are same
9	equalsIgnoreCase(String)	Checks whether two strings are same, ignoring case
10	startsWith(String)	Checks whether a string starts with the specified string
11	isEmpty()	Checks whether a string is empty or not
12	replace(String, String)	Replaces the first string with second string
13	replaceAll(String, String)	Replaces the first string with second string at all occurrences.
14	substring(int, int)	Extracts a sub-string from specified start and end index values
15	toLowerCase()	Converts a string to lower case letters
16	toUpperCase()	Converts a string to upper case letters
17	trim()	Removes whitespace from both ends
18	toString(int)	Converts the value to a String object

Example

```
public class JavaStringExample
{
        public static void main(String[] args)
        {
                String title = "Java Programming";
                String siteName = "String Handling Methods";
                System.out.println("Length of title: " + title.length());
                System.out.println("Char at index 3: " + title.charAt(3));
                System.out.println("Index of 'T': " + title.indexOf('T'));
                System.out.println("Empty: " + title.isEmpty());
                System.out.println("Equals: " + siteName.equals(title));
                System.out.println("Sub-string: " + siteName.substring(9, 14));
                System.out.println("Upper case: " + siteName.toUpperCase());
      }
}
```

UNIT – II

Inheritance, Packages and Interfaces – 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, differences between classes and **interfaces**, defining an interface, implementing interface, applying interfaces, variables in interface and extending interfaces. Exploring java.io.

INHERITANCE IN JAVA

- Inheritance is an important pillar of OOP(Object-Oriented Programming).
- The process of obtaining the data members and methods from one class to another class is known as **inheritance**.

Important Terminologies Used in Java Inheritance

Super Class/Parent Class: The class whose features are inherited is known as a superclass(or a base class or a parent class).

Sub Class/Child Class: The class that inherits the other class is known as a subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

Why Do We Need Java Inheritance?

Code Reusability: The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.

Method Overriding: Method Overriding is achievable only through Inheritance. It is one of the ways by which Java achieves Run Time Polymorphism.

Abstraction: The concept of abstract where we do not have to provide all details is achieved through inheritance. Abstraction only shows the functionality to the user.

How to Use Inheritance in Java?

- The **extends** keyword is used for inheritance in Java.
- Using the extends keyword indicates you are derived from an existing class. In other words, "extends" refers to increased functionality.

Syntax

```
class SubclassName extends SuperclassName
{
        //methods and fields
}
```

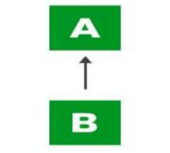
TYPES OF INHERITANCE

Based on number of ways inheriting the feature of base class into derived class we have five types of inheritance they are:

1. Single inheritance
2. Multilevel inheritance
3. Hierarchical inheritance
4. Multiple inheritance
5. Hybrid inheritance

1. Single inheritance

In single inheritance there exists single base class and single derived class.



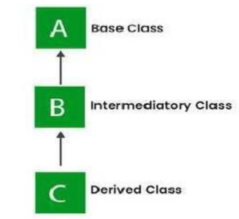
Single Inheritance

Example

```
class Animal
{
      String name;
      void show()
      {
              System.out.println("Animal name is:"+name);
      }
}
class Dog extends Animal
{
      void bark()
      {
              System.out.println("Barking");
      }
}
class TestInheritance
{
      public static void main(String args[])
      {
              Dog d=new Dog();
              d.name="DOG";
              d.show();
              d.bark();
      }
}
```

2. Multilevel inheritances in Java

- When there is a chain of inheritance, it is known as multilevel inheritance.
- In Multilevel inheritances there exists single base class, single derived class and multiple intermediate base classes.



Multilevel Inheritance

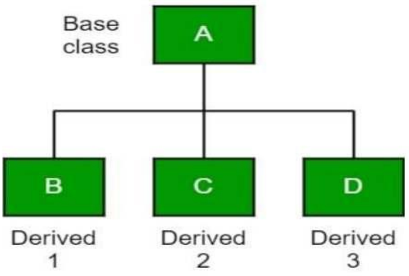
Example

In the example, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

```
class Animal
{
        String name;
        void show()
        {
                System.out.println("Animal Name is"+name);
        }
}
class Dog extends Animal
{
        void bark()
        {
                System.out.println("Mother Dog Barking...");
        }
}
class BabyDog extends Dog
{
        void weep()
        {
                System.out.println("Baby Dog weeping");
        }
}
class TestInheritance2
{
        public static void main(String args[])
        {
                BabyDog d=new BabyDog();
                d.name="MotherDog";
                d.show();
                d.bark();
                d.weep();

        }
}
```

3. Hierarchical Inheritance

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived class B, C and D.

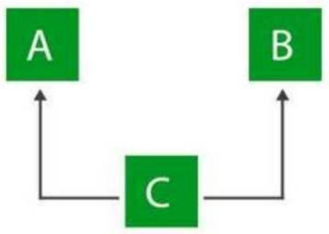


Example

```
class Animal
{
        void eat()
        {
                System.out.println("eating...");
        }
}
class Dog extends Animal
{
        void bark()
        {
                System.out.println("barking...");
        }
}
class Cat extends Animal
{
        void meow()
        {
                System.out.println("meowing...");
        }
}
class TestInheritance3
{
        public static void main(String args[])
        {
                Cat c=new Cat();
                c.meow();
                c.eat();
                //c.bark();//C.T.Error
        }
}
```

4. Multiple inheritance

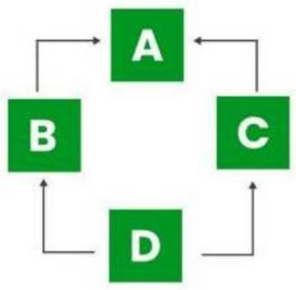
In multiple inheritance there exist multiple classes and single derived class.



The concept of multiple inheritance is not supported in java through concept of classes but it can be supported through the concept of interface.

5. Hybrid inheritance

It is a mix of two or more of the above types of inheritance. Since Java doesn't support multiple inheritances with classes, hybrid inheritance is also not possible with classes. In Java, we can achieve hybrid inheritance only through Interfaces.



Hybrid Inheritance

SUBSTITUTABILITY

- The inheritance concept used for the number of purposes in the java programming language. 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.

FORMS OF INHERITANCE

The following are the different forms of inheritance in java.

- Specialization
- Specification
- Construction
- Extension
- Limitation
- Combination

Specialization

It is the most ideal form of inheritance. The subclass is a special case of the parent class. It holds the principle of substitutability.

Specification

This is another commonly used form of inheritance. In this form of inheritance, 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

This is another form of inheritance where the child class may change the behavior defined by the parent class (overriding). It does not hold the principle of substitutability.

Extension

This is another form of inheritance where the child class may add its new properties. It holds the principle of substitutability.

Limitation

This is another form of inheritance where the subclass restricts the inherited behavior. It does not hold the principle of substitutability.

Combination

This is another form of inheritance 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.
- With inheritance, we will be able to override the methods of the base class so that the meaningful implementation of the base class method can be designed in the derived class. An inheritance leads to less development and maintenance costs.
- In inheritance base class can decide to keep some data private so that it cannot be altered by the derived class.

THE 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.

ACCESS CONTROL(MEMBER ACCESS)

In Java, Access modifiers help to restrict the scope of a class, constructor, variable, method, or data member. It provides security, accessibility, etc to the user depending upon the access modifier used with the element.

Types of Access Modifiers in Java

There are four types of access modifiers available in Java:

1. Default – No keyword required
2. Private
3. Protected
4. Public

1. Default Access Modifier

- When no access modifier is specified for a class, method, or data member – It is said to be having the **default** access modifier by default.
- The default modifier is accessible only within package.
- It cannot be accessed from outside the package.
- It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
//save by A.java
package pack;
class A
{
        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();    //Compile Time Error
                obj.msg();          //Compile Time Error
        }
}
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

2. private

- The private access modifier is accessible only within the class.
- The private access modifier is specified using the keyword **private**.
- The methods or data members declared as private are accessible only **within the class** in which they are declared.
- Any other **class of** the **same package will not be able to access** these members.
- Top-level classes or interfaces can not be declared as private because private means "only visible within the enclosing class".

Example

- In this example, we have created two classes A and Simple.
- A class contains private data member and private method.
- We are accessing these private members from outside the class, so there is a compile-time error.

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

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

3. protected

- The protected access modifier is accessible within package and outside the package but through inheritance only.
- The protected access modifier is specified using the keyword **protected**.

Example

- In this example, we have created the two packages pack and mypack.
- The A class of pack package is public, so can be accessed from outside the package.
- But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

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

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

4. public

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

Example

```
//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();
        }
}
```

Table: class member access

Let's understand the access modifiers in Java by a simple table.

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

SUPER KEYWORD

Super keyword in java is a reference variable that is used to refer parent class features.

Usage of Java super Keyword

1. Super keyword At Variable Level
2. Super keyword At Method Level
3. Super keyword At Constructor Level

- Whenever the derived class is inherits the base class features, there is a possibility that base class features are similar to derived class features and JVM gets an ambiguity.
- In order to differentiate between base class features and derived class features must be preceded by super keyword.

Syntax

super.baseclass features

1. Super Keyword at Variable Level

- Whenever the derived class inherit base class data members there is a possibility that base class data member are similar to derived class data member and JVM gets an ambiguity.
- In order to differentiate between the data member of base class and derived class, in the context of derived class the base class data members must be preceded by super keyword.

Syntax

super.baseclass datamember name

Example

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

2. Super Keyword at Method Level

- The **super keyword** can also be used to invoke or call parent class method.
- It should be use in case of method overriding. In other word **super keyword** use when base class method name and derived class method name have same name.

Example

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

3. Super keyword At Constructor Level

The super keyword can also be used to invoke the parent class constructor.

```
class Animal
{
        Animal()
        {
                System.out.println("animal is created");
        }
}
class Dog extends Animal
{
        Dog()
        {
                super();
                System.out.println("dog is created");
        }
}
class TestSuper3
{
        public static void main(String args[])
        {
                Dog d=new Dog();
        }
}
```

FINAL KEWWORD

- It is used to make a variable as a constant, Restrict method overriding, Restrict inheritance.
- Final keyword is used to make a variable as a constant.
- This is similar to const in other language.

In java language final keyword can be used in following ways:

1. Final Keyword at Variable Level
2. Final Keyword at Method Level
3. Final Keyword at Class Level

1. Final at variable level

- A variable declared with the final keyword cannot be modified by the program after initialization.
- This is useful to universal constants, such as "PI".

Example

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

Output: Compile Time Error

2. Final Keyword at method level

- It makes a method final, meaning that sub classes can not override this method. The compiler checks and gives an error if you try to override the method.
- When we want to restrict overriding, then make a method as a final.

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();
        }
}
```

Output: It gives an error

3. Final Keyword at Class Level

It makes a class final, meaning that the class cannot be inheriting by other classes. When we want to restrict inheritance then make class as a final.

Example

```
final class Bike
{
}
class Honda1 extends Bike
{
        void run()
        {
                System.out.println("running safely with 100kmph");
        }
        public static void main(String args[])
        {
                Honda1 honda= new Honda1();
                honda.run();
        }
}
```

Output: Compile Time Error

POLYMORPHISM

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

Types of Java polymorphism

The Java polymorphism is mainly divided into two types:

1. Compile-time Polymorphism(Method Overloading)
2. Runtime Polymorphism(Method Overriding)

Ad Hoc Polymorphism(Method Overloading)

Whenever same method name is exiting multiple times in the same class with different number of parameter or different order of parameters or different types of parameters is known as **method overloading**.

Example

```
class Addition
{
        void sum(int a, int b)
        {
                System.out.println(a+b);
        }
        void sum(int a, int b, int c)
        {
                System.out.println(a+b+c);
        }
        void sum(float a, float b)
        {
                System.out.println(a+b);
        }
}
class Methodload
{
        public static void main(String args[])
        {
                Addition obj=new Addition();
                obj.sum(10, 20);
                obj.sum(10, 20, 30);
                obj.sum(10.05f, 15.20f);
        }
}
```

Pure Polymorphism(Method Overriding)

- Whenever same method name is existing in both base class and derived class with same types of parameters or same order of parameters is known as **method Overriding**.
- In a java programming language, pure polymorphism carried out with a method overriding concept.

Note: Without Inheritance method overriding is not possible.

Example

```
class Walking
{
    void walk()
    {
            System.out.println("Man walking fastly");
    }
}
class Man extends Walking
{
    void walk()
    {
            System.out.println("Man walking slowly");
            super.walk();
    }
}
class OverridingDemo
{
    public static void main(String args[])
    {
        Man obj = new Man();
        obj.walk();
    }
}
```

Note:

- Whenever we are calling overridden method using derived class object reference the highest priority is given to current class (derived class). We can see in the above example high priority is derived class.
- super. (super dot) can be used to call base class overridden method in the derived class.

ABSTRACT CLASS

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

Abstraction is a process of hiding the implementation details and showing only functionality to the user. There are two ways to achieve abstraction in java.

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

Syntax

```
abstract class className
{
      ......
}
```

ABSTRACT METHOD

- An abstract method contains only declaration or prototype but it never contains body or definition.
- In order to make any undefined method as abstract whose declaration is must be predefined by abstract keyword.

Syntax

```
abstract returntype methodName(List of formal parameter);
```

Example

```
abstract class Shape
{
        abstract void draw();
}
class Rectangle extends Shape
{
        void draw()
        {
                System.out.println("drawing rectangle");
        }
}
class Circle1 extends Shape
{
        void draw()
        {
                System.out.println("drawing circle");
        }
}
class TestAbstraction1
{
         static void main(String args[])
        {
                Shape s=new Circle1();
                s.draw();
        }
}
```

Example2

```
import java.util.*;
abstract class Shape
{
    int length, breadth, radius;
    Scanner input = new Scanner(System.in);
    abstract void printArea();
}
class Rectangle extends Shape
{
    void printArea()
   {
            System.out.println("*** Finding the Area of Rectangle ***");
            System.out.print("Enter length and breadth: ");
            length = input.nextInt();
            breadth = input.nextInt();
            System.out.println("The area of Rectangle is: " + length * breadth);
    }
}
class Triangle extends Shape
{
    void printArea()
   {
            System.out.println("\n*** Finding the Area of Triangle ***");
            System.out.print("Enter Base And Height: ");
            length = input.nextInt();
            breadth = input.nextInt();
            System.out.println("The area of Triangle is: " + (length * breadth) / 2);
    }
}
class Cricle extends Shape
{
    void printArea()
   {
            System.out.println("\n*** Finding the Area of Cricle ***");
            System.out.print("Enter Radius: ");
            radius = input.nextInt();
            System.out.println("The area of Cricle is: " + 3.14f * radius * radius);
    }
}
public class AbstractClassExample
{
    public static void main(String[] args)
   {
            Rectangle rec = new Rectangle();
            rec.printArea();
            Triangle tri = new Triangle();
            tri.printArea();
            Cricle cri = new Cricle();
            cri.printArea();
    }
}
```

OBJECT CLASS

- In java, the Object class is the super most class of any class hierarchy. The Object class in the java programming language is present inside the java.lang package.
- Every class in the java programming language is a subclass of Object class by default.
- The Object class is useful when you want to refer to any object whose type you don't know. Because it is the superclass of all other classes in java, it can refer to any type of object.

Method	**Description**	**Return Value**
getClass()	Returns Class class object	object
hashCode()	returns the hashcode number for object being used.	int
equals(Object obj)	compares the argument object to calling object.	boolean
clone()	Compares two strings, ignoring case	int
concat(String)	Creates copy of invoking object	object
toString()	returns the string representation of invoking object.	String
notify()	wakes up a thread, waiting on invoking object's monitor.	void
notifyAll()	wakes up all the threads, waiting on invoking object's monitor.	void
wait()	causes the current thread to wait, until another thread notifies.	void
wait(long,int)	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies.	void
finalize()	It is invoked by the garbage collector before an object is being garbage collected.	void

PACKAGES IN JAVA

A package is a collection of similar types of classes, interfaces and sub-packages.

Types of packages

Package are classified into two type which are given below.

1. Predefined or built-in package
2. User defined package

1. Predefined or built-in package

These are the packages which are already designed by the Sun Microsystem and supply as a part of java API, every predefined package is collection of predefined classes, interfaces and sub-package.

Following are the list of predefined packages in java

- **java.lang** – This package provides the language basics.
- **java.util** – This packages provides classes and interfaces (API's) related to collection frame work, events, data structure and other utility classes such as date.
- **java.io** – This packages provides classes and interfaces for file operations, and other input and output operations.
- **java.awt** – This packages provides classes and interfaces to create GUI components in Java.
- **java.time** – The main API for dates, times, instants, and durations.

2. User defined package

- If any package is design by the user is known as user defined package.
- User defined package are those which are developed by java programmer and supply as a part of their project to deal with common requirement.

DEFINING A PACKAGE

Rules to create user defined package

- Package statement should be the first statement of any package program.
- Choose an appropriate class name or interface name and whose modifier must be public.
- Any package program can contain only one public class or only one public interface but it can contain any number of normal classes.
- Package program should not contain any main() method.
- Modifier of constructor of the class which is present in the package must be public. (This is not applicable in case of interface because interface have no constructor.)
- The modifier of method of class or interface which is present in the package must be public (This rule is optional in case of interface because interface methods by default public)
- Every package program should be save either with public class name or public Interface name

- If you omit the package statement, the class names are put into the default package, which has no name.

Syntax

```
package packagename;
```

Example

```
package mypack;
```

Compile package programs

For compilation of package program first we save program with public className.java and it compile using below syntax:

Syntax

```
javac -d . className.java
```

Explanation

- In above syntax **"-d"** is a specific tool which tells to java compiler create a separate folder for the given package in given path.
- When we give specific path then it create a new folder at that location and when we use . (dot) then it crate a folder at current working directory.

Note: Any package program can be compile but can not be execute or run. These program can be executed through user defined program which are importing package program.

Example of Package Program

Package program which is save with A.java and compile by javac -d . A.java.

```
package mypack;
public class A
{
        public void show()
        {
                System.out.println("Sum method");
        }
}
```

IMPORTING PACKAGES

- To import the java package into a class, we need to use the java import keyword which is used to access the package and its classes into the java program.
- Use import to access built-in and user-defined packages into your java source file to refer to a class in another package by directly using its name.

syntax:

```
import package.name.ClassName;   // To import a certain class only
import package.name.*            // To import the whole package
```

Example:

```
import java.util.Date;           // imports only Date class
import java.io.*;                // imports everything inside java.io package
```

Example

```
import mypack.A;
public class Hello
{
    public static void main(String args[])
    {
        A a=new A();
        a.show();
        System.out.println("show() class A");
   }
}
```

CLASSPATH

CLASSPATH can be set by any of the following ways:

- CLASSPATH can be set permanently in the environment:
- In Windows, choose control panel
- System
- Advanced
- Environment Variables
- choose "System Variables" (for all the users) or "User Variables" (only the currently login user)
- choose "Edit" (if CLASSPATH already exists) or "New"
- Enter "CLASSPATH" as the variable name
- Enter the required directories and JAR files (separated by semicolons) as the value (e.g., ".;c:\javaproject\classes;d:\tomcat\lib\servlet-api.jar").
- Take note that you need to include the current working directory (denoted by '.') in the CLASSPATH.

To check the current setting of the CLASSPATH, issue the following command:

- > SET CLASSPATH
- CLASSPATH can be set temporarily for that particular CMD shell session by issuing the following command:
- > SET CLASSPATH=.;c:\javaproject\classes;d:\tomcat\lib\servlet-api.jar
- Instead of using the CLASSPATH environment variable, you can also use the command-line option -classpath or -cp of the javac and java commands, for example,

  > java –classpath c:\javaproject\classes com.abc.project1.subproject2.MyClass3

INTERFACES

- **Interface** is similar to class which is collection of public static final variables (constants) and abstract methods.
- The interface is a mechanism to achieve fully abstraction in java. There can be only abstract methods in the interface. It is used to achieve fully abstraction and multiple inheritance in Java.

Why do we use an Interface?

- It is used to achieve total abstraction.
- Since java does not support multiple inheritances in the case of class, by using an interface it can achieve multiple inheritances.
- Any class can extend only 1 class but can any class implement infinite number of interface.
- Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?
- The reason is, abstract classes may contain non-final variables, whereas variables in the interface are final, public and static.

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"
A class can be instantiated i.e., objects of a class can be created.	An Interface cannot be instantiated i.e. objects cannot be created.
Classes do not support multiple inheritance.	The interface supports multiple inheritance.
It can be inherited from another class.	It cannot inherit a class.
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 and methods in a class can be declared using any access specifier(public, private, default, protected).	All variables and methods in an interface are declared as public.
Variables in a class can be static, final, or neither.	All variables are static and final.

DEFINING INTERFACES

The **interface** keyword is used to declare an interface.

Syntax

```
interface interface_name
{
        declare constant fields
        declare methods that abstract
}
```

Example

```
interface A
 {
        public static final int a = 10;
        void display();
}
```

IMPLEMENTING INTERFACES

A class uses the **implements** keyword to implement an interface.

Example

```
interface A
 {
        public static final int a = 10;
        void display();
}
class B implements A
{
        public void display()
        {
                System.out.println("Hello");
        }
}
class InterfaceDemo
{
        public static void main (String[] args)
        {
                B obj= new B();
                obj.display();
                System.out.println(a);
  }
}
```

APPLYING INTERFACES

To understand the power of interfaces, let's look at a more practical example.

Example:

```
interface IntStack
{
      void push(int item);
      int pop();
}
class FixedStack implements IntStack
{
      private int stck[];
      private int top;
      FixedStack(int size)
      {
            stck = new int[size];
            top = -1;
      }
      public void push(int item)
      {
            if(top==stck.length-1)
              System.out.println("Stack is full.");
            else
              stck[++top] = item;
      }
      public int pop()
      {
            if(top ==-1)
            {
                  System.out.println("Stack underflow.");
                  return 0;
            }
            else
            return stck[top--];
      }
}
class InterfaceTest
{
            public static void main(String args[])
            {
                  FixedStack mystack1 = new FixedStack(5);
                  FixedStack mystack2 = new FixedStack(8);
                  for(int i=0; i<5; i++)
                        mystack1.push(i);
                  for(int i=0; i<8; i++)
                        mystack2.push(i);
                  for(int i=0; i<5; i++)
                        System.out.println(mystack1.pop());
                  for(int i=0; i<8; i++)
                        System.out.println(mystack2.pop());
            }
}
```

VARIABLES IN INTERFACE

- Variables can be declared inside of interface declarations. They are implicitly final and static, meaning they cannot be changed by the implementing class.
- You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values.

Example

```
interface SharedConstants
{
        int NO = 0;
        int YES = 1;
        int MAYBE = 2;
        int LATER = 3;
        int NEVER = 4;
}
class Question implements SharedConstants
{
    BufferedReader br=new BufferedReader(new InputStreamreader(System.in));
    int ask()
    {
            System.out.println("would u like to have a cup of coffee?)
            String ans=br.readLine();
            if (ans= ="no")
            return NO;
            else if (ans=="yes")
            return YES;
            else if (ans=="notnow")
            return LATER;
            else
            return NEVER;
     }
}
class AskMe
{
       public static void main(String args[])
       {
            Question q = new Question();
            System.out.println(q.ask());
       }
}
```

EXTENDING INTERFACES

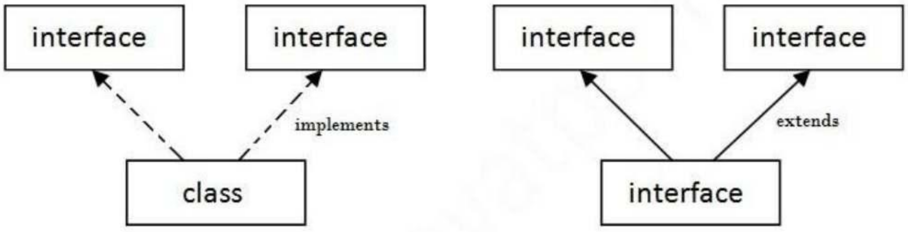
- One interface can inherit another by use of the keyword **extends**.
- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

Example

```
  interface A
  {
        void meth1();
        void meth2();
  }
  interface B extends A
{
        void meth3();
}
class MyClass implements B
{
        public void meth1()
        {
                System.out.println("Implement  meth1().");
        }
        public void meth2()
        {
                System.out.println("Implement  meth2().");
        }
        public void meth3()
        {
                 System.out.println("Implement  meth3().");
        }
 }
  class InterfaceDemo
  {
        public static void main(String args[])
        {
                MyClass ob = new MyClass();
                ob.meth1();
                ob.meth2();
                ob.meth3();
       }
   }
```

MULTIPLE INHERITANCE IN JAVA BY INTERFACE

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



Multiple Inheritance in Java

Example

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

STREAM BASED I/O (JAVA.IO)

- **Java I/O** (Input and Output) is used *to process the* input *and* produce the output.
- Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.
- We can perform **file handling in Java** by Java I/O API.

STREAM

In Java, streams are the sequence of data that are read from the source and written to the destination.

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

1. System.in: This is the **standard input stream** that is used to read characters from the keyboard or any other standard input device.

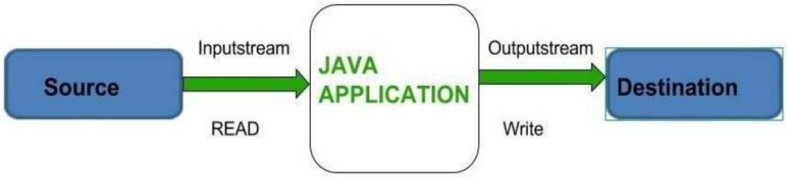
2. System.out: This is the **standard output stream** that is used to produce the result of a program on an output device like the computer screen.

3. System.err: This is the **standard error stream** that is used to output all the error data that a program might throw, on a computer screen or any standard output device.

TYPES OF STREAMS

Depending on the type of operations, streams can be divided into two primary classes:

- **InPutStream** – The InputStream is used to read data from a source.
- **OutPutStream** – The OutputStream is used for writing data to a destination.



Depending upon the data a stream can be classified into:

1. Byte Stream
2. Character Stream

1. BYTE STREAM

Java byte streams are used to perform input and output of 8-bit bytes.

Byte Stream Classes

All byte stream classes are derived from base abstract classes called **InputStream** and **OutputStream**.

InputStream Class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

Subclasses of InputStream

In order to use the functionality of InputStream, we can use its subclasses. Some of them are:

Stream class	Description
BufferedInputStream	Used for Buffered Input Stream.
DataInputStream	Contains method for reading java standard datatype
FileInputStream	Input stream that reads from a file

Methods of InputStream

The InputStream class provides different methods that are implemented by its subclasses. Here are some of the commonly used methods:

- **read() -** reads one byte of data from the input stream
- **read(byte[] array) -** reads bytes from the stream and stores in the specified array
- **available() -** returns the number of bytes available in the input stream
- **mark() -** marks the position in the input stream up to which data has been read
- **reset() -** returns the control to the point in the stream where the mark was set
- **close() -** closes the input stream

OutputStream class

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes.

Subclasses of OutputStream

In order to use the functionality of OutputStream, we can use its subclasses. Some of them are:

Stream class	Description
BufferedOutputStream	Used for Buffered Output Stream.
DataOutputStream	An output stream that contain method for writing java standard data type
FileOutputStream	Output stream that write to a file.
PrintStream	Output Stream that contain print() and println() method

Methods of OutputStream

The OutputStream class provides different methods that are implemented by its subclasses. Here are some of the methods:

- **write() -** writes the specified byte to the output stream
- **write(byte[] array) -** writes the bytes from the specified array to the output stream
- **flush() -** forces to write all data present in output stream to the destination
- **close() -** closes the output stream

2. CHARACTER STREAM

Character stream is used to read and write a single character of data.

Character Stream Classes

All the character stream classes are derived from base abstract classes **Reader** and **Writer.**

Reader Class

The Reader class of the java.io package is an abstract super class that represents a stream of characters.

Sub classes of Reader Class

In order to use the functionality of Reader, we can use its subclasses. Some of them are:

Stream class	Description
BufferedReader	Handles buffered input stream.
FileReader	Input stream that reads from file.
InputStreamReader	Input stream that translate byte to character

Methods of Reader

The Reader class provides different methods that are implemented by its subclasses. Here are some of the commonly used methods:

- **ready() -** checks if the reader is ready to be read
- **read(char[] array) -** reads the characters from the stream and stores in the specified array
- **read(char[] array, int start, int length) -** reads the number of characters equal to length from the stream and stores in the specified array starting from the start
- **mark() -** marks the position in the stream up to which data has been read
- **reset() -** returns the control to the point in the stream where the mark is set
- **skip() -** discards the specified number of characters from the stream

Writer Class

- The Writer class of the java.io package is an abstract super class that represents a stream of characters.
- Since Writer is an abstract class, it is not useful by itself. However, its subclasses can be used to write data.

Subclasses of Writer

Stream class	Description
BufferedWriter	Handles buffered output stream.
FileWriter	Output stream that writes to file.
PrintWriter	Output Stream that contain print() and println() method.

Methods of Writer

The Writer class provides different methods that are implemented by its subclasses. Here are some of the methods:

- **write(char[] array) -** writes the characters from the specified array to the output stream
- **write(String data) -** writes the specified string to the writer
- **append(char c) -** inserts the specified character to the current writer
- **flush() -** forces to write all the data present in the writer to the corresponding destination
- **close() -** closes the writer

READING CONSOLE INPUT

There are times when it is important for you to get input from users for execution of programs. To do this you need Java Reading Console Input Methods.

Java Reading Console Input Methods

1. Using BufferedReader Class
2. Using Scanner Class
3. Using Console Class

1. Using BufferedReader Class

- Reading input data using the BufferedReader class is the traditional technique. This way of the reading method is used by wrapping the System.in (standard input stream) in an InputStreamReader which is wrapped in a BufferedReader, we can read input from the console.
- The BufferedReader class has defined in the java.io package.
- We can use read() method in BufferedReader to read a character.

int read() throws IOException

Reading Console Input Characters Example:

```
import java.io.*;
class ReadingConsoleInputTest
{
        public static void main(String args[])
        {
            char ch;
            BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("Enter characters, char 'x' to exit.");
            do
            {
                    ch = (char) br.read();
                    System.out.println(ch);
            } while(ch != 'x');
        }
}
```

How to read a string input in java?

readLine() method is used to read the string in the BufferedReader.

Program to take String input from Keyboard in Java

```
import java.io.*;
class MyInput
{
 public static void main(String[] args)
 {
  String text;
  InputStreamReader isr = new InputStreamReader(System.in);
  BufferedReader br = new BufferedReader(isr);
  text = br.readLine();   //Reading String
  System.out.println(text);
 }
}
```

2. Using the Scanner Class

Scanner is one of the predefined class which is used for reading the data dynamically from the keyboard.

Import Scanner Class in Java

java.util.Scanner

Constructor of Scanner Class

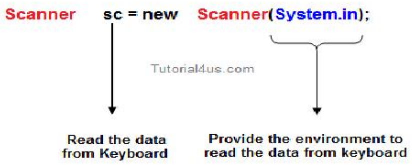
Scanner(InputStream)

This constructor create an object of Scanner class by talking an object of InputStream class. An object of InputStream class is called **in** which is created as a static data member in the System class.

Syntax of Scanner Class in Java

```
Scanner  sc=new  Scanner(System.in);
```

Here the object **'in'** is use the control of keyboard


Scanner sc = new Scanner(System.in);
Tutorial4us.com
Read the data from Keyboard
Provide the environment to read the data from keyboard

Instance methods of Scanner Class

S.No	Method	Description
1	public byte nextByte()	Used for read byte value
2	public short nextShort()	Used for read short value
3	public int nextInt()	Used for read integer value
4	public long nextLong()	Used for read numeric value
5	public float nextLong()	Used for read numeric value
6	public double nextDouble()	Used for read double value
7	public char nextChar()	Used for read character
8	public boolean nextBoolean()	Used for read boolean value
9	public String nextLine()	Used for reading any kind of data in the form of String data.

Example of Scanner Class in Java

```
import java.util.Scanner
public class ScannerDemo
{
      public static void main(String args[])
      {
          Scanner  s=new  Scanner(System.in);
          System.out.println("Enter  first  no= ");
          int num1=s.nextInt();
          System.out.println("Enter second no= ");
          int num2=s.nextInt();
          System.out.println("Sum of no is= "+(num1+num2));
      }
}
```

3. Using the Console Class

- This is another way of reading user input from the console in Java.
- The Java Console class is be used to get input from console. It provides methods to read texts and passwords.
- If you read password using Console class, it will not be displayed to the user.
- The Console class is defined in the java.io class which needs to be imported before using the console class.

Example

```
import java.io.*;
class consoleEg
{
     public static void main(String args[])
     {
          String name;
          System.out.println ("Enter your name: ");
          Console c = System.console();
          name = c.readLine();
          System.out.println ("Your name is: " + name);
     }

}
```

WRITING CONSOLE OUTPUT

- print and println methods in System.out are mostly used for console output.
- These methods are defined by the class PrintStream which is the type of object referenced by System.out.
- System.out is the byte stream.
- PrintStream is the output derived from OutputStream. write method is also defined in PrintStream for console output.

void write(int byteval)

//Java code to Write a character in Console Output

```
import java.io.*;
class WriteCharacterTest
{
        public static void main(String args[])
        {
                int byteval;
                byteval = 'J';
                System.out.write(byteval);
                System.out.write('\n');
        }
}
```

w

UNIT - III

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.

CONCEPTS OF EXCEPTION HANDLING

EXCEPTION

- An **Exception** is a run time error, which occurs during the execution of a program, that distrupts the normal flow of the program's instructions.
- It 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

- Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc.
- Errors are usually beyond the control of the programmer, and we should not try to handle errors.

EXCEPTION HANDLING IN JAVA

- The **Exception Handling in Java** is one of the powerful *feature to handle the runtime errors* so that normal flow of the application can be maintained.
- **Exception Handling is used to** convert system error message into user friendly error message.

Let's take a scenario:

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5; //Exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

- Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed.
- If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in Java.

UNCAUGHT EXCEPTIONS(WITH OUT USING TRY & CATCH)

Example without Exception Handling

```
class ExceptionDemo
{
    public static void main(String[] args)
    {
        int a=30, b=0;
        int c=a/b;
        System.out.println("Denominator should  not be zero");
    }
}
```

Output: Exception in thread "main" java.lang.ArithmeticException: / by zero at ExceptionDemo.main(ExceptionDemo.java:7)

Explanation:

- Abnormally terminate program and give a message like below,

  Exception in thread "main" java.lang.ArithmeticException: / by zero
  at ExceptionDemo.main(ExceptionDemo.java:7)

- This error message is not understandable by user so we convert this error message into user friendly error message, like "denominator should not be zero".

BENEFITS OF EXCEPTIONHANDLING

1. Provision to Complete Program Execution
2. Easy Identification of Program Code and Error-Handling Code
3. Propagation of Errors
4. Meaningful Error Reporting
5. Identifying Error Types

TERMINATION OR RESUMPTIVE MODELS

In java, there are two exception models. Java programming language has two models of exception handling. The exception models that java supports are as follows.

- **Termination Model**
- **Resumptive Model**

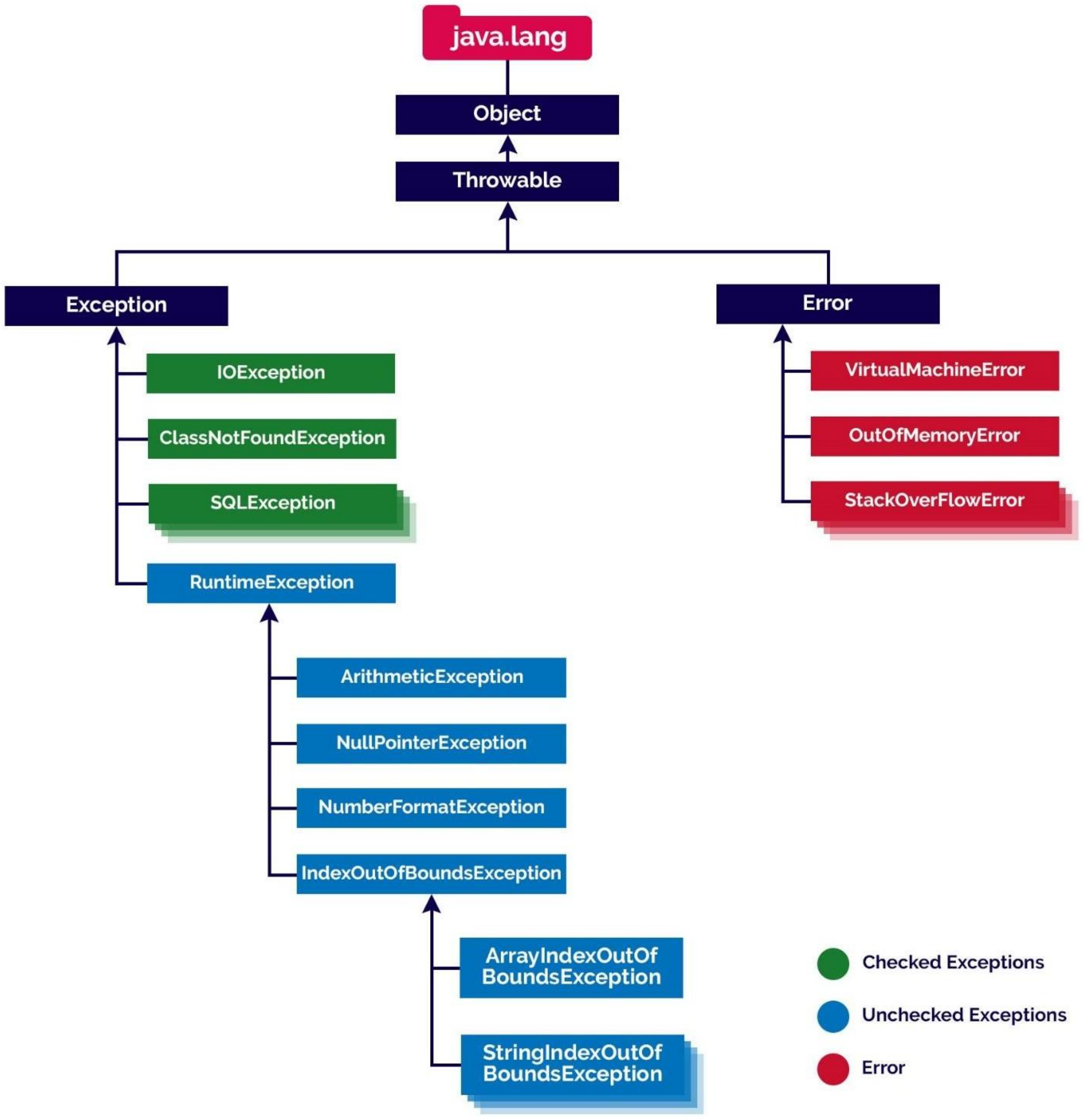
Termination Model

- In the termination model, when a method encounters an exception, further processing in that method is terminated and control is transferred to the nearest catch block that can handle the type of exception encountered.
- In other words we can say that in termination model the error is so critical there is no way to get back to where the exception occurred.

Resumptive Model

- The alternative of termination model is resumptive model.
- In resumptive model, the exception handler is expected to do something to stable the situation, and then the faulting method is retried.
- In resumptive model we hope to continue the execution after the exception is handled.
- In resumptive model we may use a method call that want resumption like behavior.
- We may also place the try block in a while loop that keeps re-entering the try block until the result is satisfactory.

EXCEPTION HIERARCHY



HOW TO HANDLE THE EXCEPTION

Use Five keywords for Handling the Exception

1. try
2. catch
3. throw
4. throws
5. finally

1. try block

- The try block contains set of statements where an exception can occur.
- In other words try block always contains problematic statements.
- A try block is always followed by a catch block, which handles the exception that occurs in associated try block. A try block must be followed by catch blocks or finally block or both.

Syntax

```
try
{
   //statements that may cause an exception
}
```

2. catch block

- A catch block is where we handle the exceptions, this block must follow the try block.
- A single try block can have multiple catch blocks associated with it. We can catch different exceptions in different catch blocks.
- When an exception occurs in try block, the corresponding catch block that handles that particular exception executes.
- For example if an arithmetic exception occurs in try block then the statements enclosed in catch block for arithmetic exception executes.

Syntax of try-catch in java

```
try
{
     // statements causes problem at run time
}
catch(type of exception-1 object-1)
{
    // statements provides user friendly error message
}
catch(type of exception-2 object-2)
{
    // statements provides user friendly error message
}
```

Example1: ArithmeticException

```
class ExceptionDemo
{
        public static void main(String[] args)
        {
            int a=30, b=0;
            try
            {
              int c=a/b;
            }
            catch (ArithmeticException e)
            {
              System.out.println("Denominator should not be zero");
            }
        }
 }
```

Output: Denominator should not be zero

Example2: NullPointerException

```
class NullPointer_Demo
{
  public static void main(String args[])
  {
    try
    {
            String a = null; //null value
            System.out.println(a.charAt(0));
    }
    catch(NullPointerException  e)
    {
            System.out.println("NullPointerException..");
    }
  }
}
```

Output: NullPointerException..

Example3: FileNotFoundException

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
class File_notFound_Demo
{
    public static void main(String args[])
    {
      try
      {          // Following file does not exist
                 File file = new File("E://file.txt");
                 FileReader fr = new FileReader(file);
      }
      catch(FileNotFoundException e)
      {
                 System.out.println("File does not exist");
      }
    }
}
```

Output: File does not exist

MULTIPLE CATCH BLOCKS

We can write multiple catch blocks for generating multiple user friendly error messages to make our application strong.

Example

```
class ExceptionDemo
{
        public static void main(String[] args)
        {
                int a=30, b=0;
                try
                {
                        int c=a/b;
                        System.out.println("Result: "+c);
                }
                catch(NullPointerException e)
                {
                        System.out.println("Enter valid number");
                }
                catch(ArithmeticException e)
                {
                        System.out.println("Denominator not be zero");
                }

        }
}
```

NESTED TRY STATEMENTS

The try block within a try block is known as nested try block in java.

Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax

```
try
{
        statement 1;
        statement 2;
        try
        {
                statement 1;
                statement 2;
        }
        catch(Exception e)
        {
              ...........
        }
}
catch(Exception e)
{
      ..........
}
```

Example

```
class NestedTry
{
        public static void main(String args[])
        {
                try
                {
                    try
                    {
                            int a[]=new int[5];
                            a[5]=4;
                    }
                    catch(ArrayIndexOutOfBoundsException  e)
                    {
                            System.out.println(e);
                    }
                }
            catch(Exception e)
            {
                    System.out.println("handeled");
            }
            System.out.println("normal flow..");
        }
}
```

3. throw

- The throw keyword in Java is used to explicitly throw an exception from a method or any block of code.
- We can throw either checked or unchecked exception.
- The throw keyword is mainly used to throw custom exceptions.

Syntax

```
throw Instance
```

Example

```
throw new ArithmeticException("/ by zero");
```

Example

```
class ThrowExcep
{
        static void fun()
        {
                try
                {
                     throw new NullPointerException("demo");
                }
                catch(NullPointerException e)
                {
                      System.out.println("Caught inside fun().");
                      throw e; // rethrowing the exception
                }
        }
        public static void main(String args[])
        {
                    try
                    {
                        fun();
                    }
                    catch(NullPointerException e)
                    {
                        System.out.println("Caught in main.");
                    }
        }
}
```

Output:

```
Caught inside fun().
Caught in main.
```

4. throws

throws is a keyword in java language which is used to throw the exception which is raised in the called method to it's calling method throws keyword always followed by method signature.

Syntax

```
returnType methodName(parameter) throws Exception_class....
{
          .....
}
```

Example

```
class ThrowsExecp
{
       static void fun() throws IllegalAccessException
       {
         System.out.println("Inside fun(). ");
         throw new IllegalAccessException("demo");
       }
       public static void main(String args[])
       {
                try
                {
                    fun();
                }
                catch(IllegalAccessException e)
                {
                    System.out.println("caught in main.");
                }
       }
}
```

Output:

```
Inside fun().
caught in main.
```

5. finally Block

- **Java finally block** is a block that is used *to execute important code* such as closing connection, stream etc.
- Java finally block is always executed whether exception is handled or not.
- Java finally block follows try or catch block.

Example

```
class TestFinallyBlock
{
        public static void main(String args[])
        {
            try
            {
                int data=25/0;
                System.out.println(data);
            }
            catch(NullPointerException e)
            {
                System.out.println(e);
            }
            finally
            {
                System.out.println("finally block is always executed");
            }
            System.out.println("rest of the code...");
        }
}
```

Output:

```
finally block is always executed
Exception in thread main java.lang.ArithmeticException:/ by zero
```

RE-THROWING EXCEPTIONS

- Sometimes we may need to rethrow an exception in Java.
- If a catch block cannot handle the particular exception it has caught, we can rethrow the exception.
- The rethrow expression causes the **originally thrown object to be rethrown.**
- Because the exception has already been caught at the scope in which the rethrow expression occurs, it is rethrown out to the next enclosing try block. Therefore, it cannot be handled by catch blocks at the scope in which the rethrow expression occurred.
- Any catch blocks for the enclosing try block have an opportunity to catch the exception.

Example

```
class RethrowExcep
{
        static void fun()
        {
                try
                {
                      throw new NullPointerException("demo");
                }
                catch(NullPointerException e)
                {
                      System.out.println("Caught inside fun().");
                      throw e; // rethrowing the exception
                }
        }
        public static void main(String args[])
        {
                try
                {
                   fun();
                }
                catch(NullPointerException e)
                {
                   System.out.println("Caught in main.");
                }
        }
}
```

Output:

```
Caught inside fun().
Caught in main.
```

CREATING OWN EXCEPTION(CUSTOM EXCEPTION IN JAVA)

If any exception is design by the user known as user defined or Custom Exception. Custom Exception is created by user.

Rules to design user defined Exception

1. Create a package with valid user defined name.
2. Create any user defined class.
3. Make that user defined class as derived class of Exception or RuntimeException class.
4. Declare parametrized constructor with string variable.
5. call super class constructor by passing string variable within the derived class constructor.
6. Save the program with public class name.java

Example

```
package nage;
public class InvalidAgeException extends Exception
{
    public InvalidAgeException (String s)
    {
        super(s);
    }
}
```

A Class that uses above InvalidAgeException:

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

Output:

```
Exception occured: InvalidAgeException:not valid
rest of the code...
```

BUILT IN EXCEPTIONS

The Java programming language has several built-in exception class that support exception handling. Every exception class is suitable to explain certain error situations at run time.

All the built-in exception classes in Java were defined a package java.lang.

EXCEPTION TYPES IN JAVA

In java, exceptions are mainly categorized into two types, and they are as follows.

- **Checked Exceptions**
- **Unchecked Exceptions**

Checked Exceptions

- The checked exception is an exception that is checked by the compiler during the compilation process to confirm whether the exception is handled by the programmer or not. If it is not handled, the compiler displays a compilation error using built-in classes.
- The checked exceptions are generally caused by faults outside of the code itself like missing resources, networking errors, and problems with threads come to mind.
- The following are a few built-in classes used to handle checked exceptions in java.
- In the exception class hierarchy, the checked exception classes are the direct children of the Exception class.

List of checked exceptions in Java

S. No.	Exception Class with Description
1	**ClassNotFoundException** It is thrown when the Java Virtual Machine (JVM) tries to load a particular class and the specified class cannot be found in the classpath.
2	**CloneNotSupportedException** Used to indicate that the clone method in class Object has been called to clone an object, but that the object's class does not implement the Cloneable interface.
3	**IllegalAccessException** It is thrown when one attempts to access a method or member that visibility qualifiers do not allow.
4	**InstantiationException** It is thrown when an application tries to create an instance of a class using the newInstance method in class, but the specified class object cannot be instantiated because it is an interface or is an abstract class.
5	**InterruptedException**

S. No.	Exception Class with Description
	It is thrown when a thread that is sleeping, waiting, or is occupied is interrupted.
6	**NoSuchFieldException** It indicates that the class doesn't have a field of a specified name.
7	**NoSuchMethodException** It is thrown when some JAR file has a different version at runtime that it had at compile time, a NoSuchMethodException occurs during reflection when we try to access a method that does not exist.

Unchecked Exceptions

- The unchecked exception is an exception that occurs at the time of program execution. The unchecked exceptions are not caught by the compiler at the time of compilation.
- The unchecked exceptions are generally caused due to bugs such as logic errors, improper use of resources, etc.
- In the exception class hierarchy, the unchecked exception classes are the children of RuntimeException class, which is a child class of Exception class.

List of unchecked exceptions in Java

S. No.	Exception Class with Description
1	**ArithmeticException** It handles the arithmetic exceptions like division by zero
2	**ArrayIndexOutOfBoundsException** It handles the situations like an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
3	**ArrayStoreException** It handles the situations like when an attempt has been made to store the wrong type of object into an array of objects
5	**ClassCastException** It handles the situation when we try to improperly cast a class from one type to another.
6	**IllegalArgumentException** This exception is thrown in order to indicate that a method has been passed an illegal or inappropriate argument.
7	**IllegalMonitorStateException** This indicates that the calling thread has attempted to wait on an object's monitor, or has attempted to notify other threads that wait on an object's monitor, without owning the specified monitor.

S. No.	Exception Class with Description
8	**IllegalStateException** It signals that a method has been invoked at an illegal or inappropriate time.
9	**IllegalThreadStateException** It is thrown by the Java runtime environment, when the programmer is trying to modify the state of the thread when it is illegal.
10	**IndexOutOfBoundsException** It is thrown when attempting to access an invalid index within a collection, such as an array , vector , string , and so forth.
11	**NegativeArraySizeException** It is thrown if an applet tries to create an array with negative size.
12	**NullPointerException** it is thrown when program attempts to use an object reference that has the null value.
13	**NumberFormatException** It is thrown when we try to convert a string into a numeric value such as float or integer, but the format of the input string is not appropriate or illegal.
14	**SecurityException** It is thrown by the Java Card Virtual Machine to indicate a security violation.
15	**StringIndexOutOfBounds** It is thrown by the methods of the String class, in order to indicate that an index is either negative, or greater than the size of the string itself.
16	**UnsupportedOperationException** It is thrown to indicate that the requested operation is not supported.

STRING HANDLING

- A string is a sequence of characters surrounded by double quotations. In a java programming language, a string is the object of a built-in class **String**.
- The string created using the **String** class can be extended. It allows us to add more characters after its definition, and also it can be modified.

Example

```
String siteName = "javaprogramming";
siteName = "javaprogramminglanguage";
```

String handling methods

In java programming language, the String class contains various methods that can be used to handle string data values.

The following table depicts all built-in methods of String class in java.

S.No	Method	Description
1	charAt(int)	Finds the character at given index
2	length()	Finds the length of given string
3	compareTo(String)	Compares two strings
4	compareToIgnoreCase(String)	Compares two strings, ignoring case
5	concat(String)	Concatenates the object string with argument string.
6	contains(String)	Checks whether a string contains sub-string
7	contentEquals(String)	Checks whether two strings are same
8	equals(String)	Checks whether two strings are same
9	equalsIgnoreCase(String)	Checks whether two strings are same, ignoring case
10	startsWith(String)	Checks whether a string starts with the specified string
11	isEmpty()	Checks whether a string is empty or not
12	replace(String, String)	Replaces the first string with second string
13	replaceAll(String, String)	Replaces the first string with second string at all occurrences.
14	substring(int, int)	Extracts a sub-string from specified start and end index values
15	toLowerCase()	Converts a string to lower case letters
16	toUpperCase()	Converts a string to upper case letters
17	trim()	Removes whitespace from both ends
18	toString(int)	Converts the value to a String object

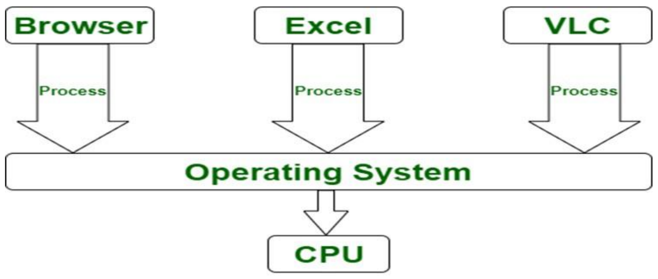
Example

```
public class JavaStringExample
{
        public static void main(String[] args)
        {
                String title = "Java Programming";
                String siteName = "String Handling Methods";
                System.out.println("Length of title: " + title.length());
                System.out.println("Char at index 3: " + title.charAt(3));
                System.out.println("Index of 'T': " + title.indexOf('T'));
                System.out.println("Empty: " + title.isEmpty());
                System.out.println("Equals: " + siteName.equals(title));
                System.out.println("Sub-string: " + siteName.substring(9, 14));
                System.out.println("Upper case: " + siteName.toUpperCase());
      }
}
```

MULTI-TASKING AND MULTI-THREADING

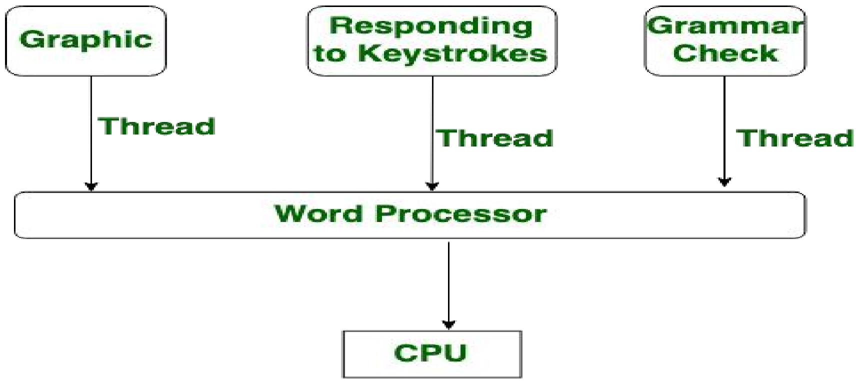
Introduction

- Multi-tasking and multi-threading are two techniques used in operating systems to manage multiple processes and tasks.
- **Multi-tasking** is the ability of an operating system to run multiple processes or tasks concurrently, sharing the same processor and other resources.
- In multi-tasking, the operating system divides the CPU time between multiple tasks, allowing them to execute simultaneously.
- Each task is assigned a time slice, or a portion of CPU time, during which it can execute its code.
- Multi-tasking is essential for increasing system efficiency, improving user productivity, and achieving optimal resource utilization.



Multitasking

- **Multi-threading** is a technique in which an operating system divides a single process into multiple threads, each of which can execute concurrently.
- Threads share the same memory space and resources of the parent process, allowing them to communicate and synchronize data easily.
- Multi-threading is useful for improving application performance by allowing different parts of the application to execute simultaneously.



Multithreading

DIFFERENCE BETWEEN MULTI-TASKING AND MULTI-THREADING

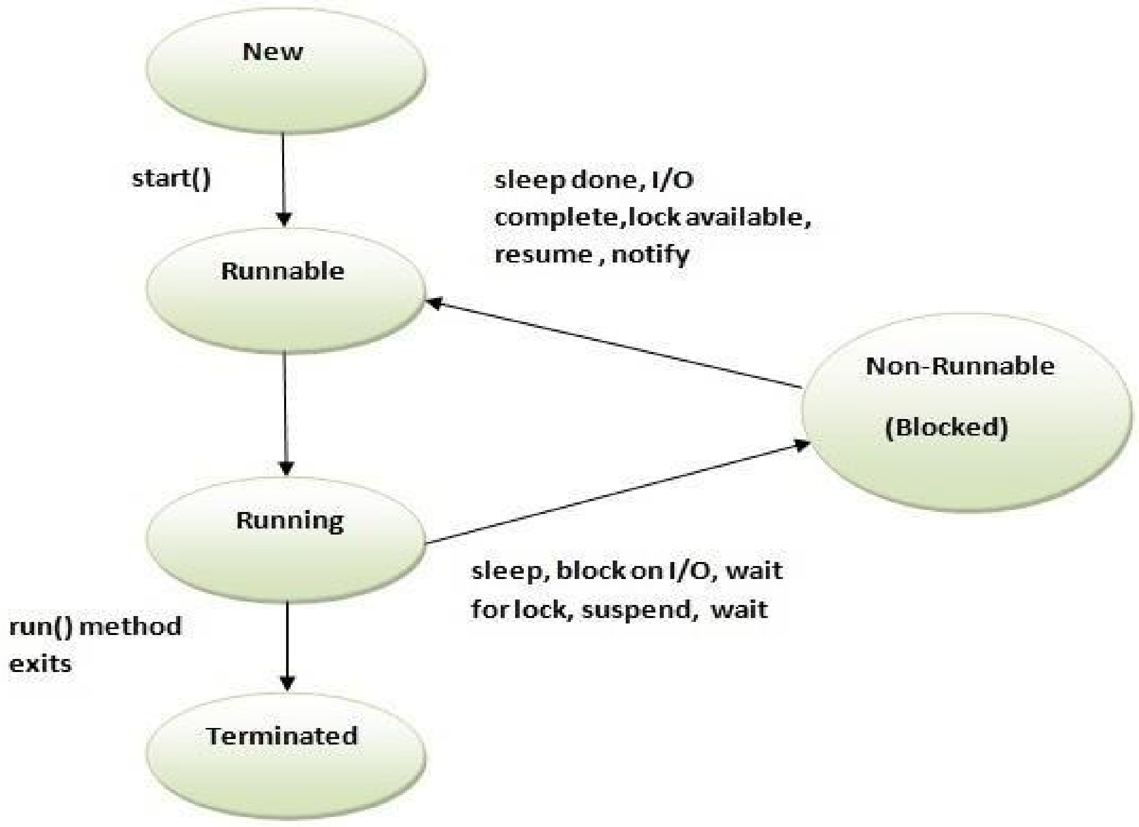
S.NO	Multitasking	Multithreading
1.	In multitasking, users are allowed to perform many tasks by CPU.	While in multithreading, many threads are created from a process through which computer power is increased.
2.	Multitasking involves often CPU switching between the tasks.	While in multithreading also, CPU switching is often involved between the threads.
3.	In multitasking, the processes share separate memory.	While in multithreading, processes are allocated the same memory.
4.	The multitasking component involves multiprocessing.	While the multithreading component does not involve multiprocessing.
5.	In multitasking, the CPU is provided in order to execute many tasks at a time.	While in multithreading also, a CPU is provided in order to execute many threads from a process at a time.
6.	In multitasking, processes don't share the same resources, each process is allocated separate resources.	While in multithreading, each process shares the same resources.
7.	Multitasking is slow compared to multithreading.	While multithreading is faster.
8.	In multitasking, termination of a process takes more time.	While in multithreading, termination of thread takes less time.
9.	Isolation and memory protection exist in multitasking.	Isolation and memory protection does not exist in multithreading.
10.	It helps in developing efficient programs.	It helps in developing efficient operating systems.
11.	Involves running multiple independent processes or tasks	Involves dividing a single process into multiple threads that can execute concurrently
12.	Multiple processes or tasks run simultaneously, sharing the same processor and resources	Multiple threads within a single process share the same memory space and resources
13.	Each process or task has its own memory space and resources	Threads share the same memory space and resources of the parent process
14.	Used to manage multiple processes and improve system efficiency	Used to manage multiple processes and improve system efficiency
15.	Examples: running multiple applications on a computer, running multiple servers on a network	Examples: splitting a video encoding task into multiple threads, implementing a responsive user interface in an application

JAVA THREAD MODEL (LIFE CYCLE OF A THREAD)

- In java, a thread goes through different states throughout its execution.
- These stages are called thread life cycle states or phases.
- A thread can be in one of the five states in the thread.
- The life cycle of the thread is controlled by JVM.

The thread states are as follows:

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


New
start()
sleep done, I/O complete,lock available, resume , notify
Runnable
Non-Runnable (Blocked)
Running
sleep, block on I/O, wait for lock, suspend, wait
run() method exits
Terminated

1. New

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

Example

```
Thread t1 = new Thread();
```

2. Runnable

- 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( );
```

3. 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.

4. Non-Runnable (Blocked)

- This is the state when the thread is still alive, but is currently not eligible to run.
- 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.
- When a thread is in the blocked or waiting state, it may move to Runnable state due to reasons like sleep time completed, waiting time completed, notify( ) or notifyAll( ) method called, resume( ) method called, etc.

Example

```
Thread.sleep(1000);
wait(1000);
wait();
suspend();
notify();
notifyAll();
resume();
```

5. 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.

CREATING THREADS

There are two ways to create a thread:

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

1. By extending Thread class

The java contains a built-in class Thread inside the java.lang package. The Thread class contains all the methods that are related to the threads.

To create a thread using Thread class, follow the step given below.

Step-1: Create a class as a child of Thread class. That means, create a class that extends Thread class.

Step-2: Override the run( ) method with the code that is to be executed by the thread. The run( ) method must be public while overriding.

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: By extending Thread class

```
class SampleThread extends Thread
{
        public void run()
        {
                System.out.println("Thread is under Running...");
                for(int i= 1; i<=10; i++)
                {
                        System.out.println("i = " + i);
                }
        }
}
public class My_Thread_Test
{

        public static void main(String[] args)
        {
                SampleThread t1 = new SampleThread();
                System.out.println("Thread about to start...");
                t1.start();
        }
}
```

Output:

```
Thread about to start...
Thread is under Running...
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
i = 10
```

2. By implementing Runnable interface

- The java contains a built-in interface Runnable inside the java.lang package.
- The Runnable interface implemented by the Thread class that contains all the methods that are related to the threads.

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 that is to be executed by the thread. The run( ) method must be public while overriding.
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: By implementing the Runnable interface

```
class SampleThread implements Runnable
{

        public void run()
        {
                System.out.println("Thread is under Running...");
                for(int i= 1; i<=10; i++)
                {
                        System.out.println("i = " + i);
                }
        }
}
public class My_Thread_Test
{
        public static void main(String[] args)
        {
                SampleThread threadObject = new SampleThread();
                Thread thread = new Thread(threadObject);
                System.out.println("Thread about to start...");
                thread.start();
        }
}
```

Output:

```
Thread about to start...
Thread is under Running...
    i = 1
    i = 2
    i = 3
    i = 4
    i = 5
    i = 6
    i = 7
    i = 8
    i = 9
    i = 10
```

CONSTRUCTORS OF THREAD CLASS

1. Thread()
2. Thread(String name)
3. Thread(Runnable r)
4. Thread(Runnable r,String name)

METHODS OF THREAD CLASS

1. **public void run():** is used to defines actual task of the thread.
2. **public void start():**It moves the thread from Ready state to Running state by calling run( ) method.
3. **public void sleep(long milliseconds):** Moves the thread to blocked state till the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long milliseconds):** waits for a thread to die for the specified milliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void suspend():** is used to suspend the thread(deprecated).

SLEEP() & JOIN()

```
class SampleThread extends Thread
{
		public void run()
		{
			System.out.println("Thread is under Running...");
			for(int i= 1; i<=10; i++)
			{
				try
				{
					Thread.sleep(1000);
				}
				catch(Exception e)
				{
					System.out.println(e);
				}
```

```
                    System.out.println("i = " + i);
                }
            }
}
public class My_Thread_Test
{
            public static void main(String[] args)
            {
                SampleThread t1 = new SampleThread();
                SampleThread t2 = new SampleThread();
                SampleThread t3 = new SampleThread();
                System.out.println("Thread about to start...");
                t1.start();
                    try
                    {
                        t1.join();
                    }
                    catch(Exception e)
                    {
                        System.out.println(e);
                    }


                t2.start();
                t3.start();
            }
}
```

THREAD PRIORITIES

- In a java programming language, every thread has a property called priority.
- Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling).
- The thread with more priority allocates the processor first.
- But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

Three constants defined in Thread class:

1. MIN_PRIORITY
2. NORM_PRIORITY
3. MAX_PRIORITY

- Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.
- The java programming language Thread class provides two methods setPriority(int), and getPriority( ) to handle thread priorities.

setPriority( ) method

The setPriority( ) method of Thread class 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

The getPriority( ) method of Thread class used to access the priority of a thread.

It does not takes any argument and returns name of the thread as String.

Example

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

Example1

```
class SampleThread extends Thread
{
     public void run()
     {
              System.out.println("Inside SampleThread");
              System.out.println("CurrentThread: " + Thread.currentThread().getName());
     }
}
public class My_Thread_Test
{
             public static void main(String[] args)
             {
                        SampleThread threadObject1 = new SampleThread();
                        SampleThread threadObject2 = new SampleThread();
                        threadObject1.setName("first");
                        threadObject2.setName("second");
                        threadObject1.setPriority(4);
                        threadObject2.setPriority(Thread.MAX_PRIORITY);
                        threadObject1.start();
                        threadObject2.start();
             }
}
```

Output:

```
Inside SampleThread
Inside SampleThread
CurrentThread: second
CurrentThread: first
```

Example2

```
class MultiThread extends Thread
{
     public void run()
    {
            System.out.println("running thread name is:"+Thread.currentThread().getName());
            System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
    }
    public static void main(String args[])
    {
            MultiThread m1=new MultiThread ();
            MultiThread m2=new MultiThread ();
            m1.setPriority(Thread.MIN_PRIORITY);
            m2.setPriority(Thread.MAX_PRIORITY);
            m1.start();
            m2.start();
    }
}
```

Output

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

SYNCHRONIZING THREADS

SYNCHRONIZATION

- The java programming language supports multithreading.
- The problem of shared resources occurs when two or more threads get execute at the same time.
- In such a situation, we need some way to ensure that the shared resource will be accessed by only one thread at a time, and this is performed by using the concept called synchronization.
- **The synchronization is the process of allowing only one thread to access a shared resource at a time.**

UNDERSTANDING THE PROBLEM WITHOUT SYNCHRONIZATION

In this example, there is no synchronization, so output is inconsistent.

Example:

```
class Table
{
        void printTable(int n)
        {
                //method not synchronized
                 for(int i=1;i<=5;i++)
                 {
                        System.out.println(n*i);
                        try
                        {
                              Thread.sleep(400);
                        }
                       catch(Exception e)
                       {
                              System.out.println(e);
                       }
                 }
           }
  }
  class MyThread1 extends Thread
  {
              Table t;
              MyThread1(Table t)
              {
                  this.t=t;
              }
              public void run()
              {
                  t.printTable(5);
              }
     }
     class MyThread2 extends Thread
```

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

Output:

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

Thread Synchronization

In java, the synchronization is achieved using the following concepts.

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

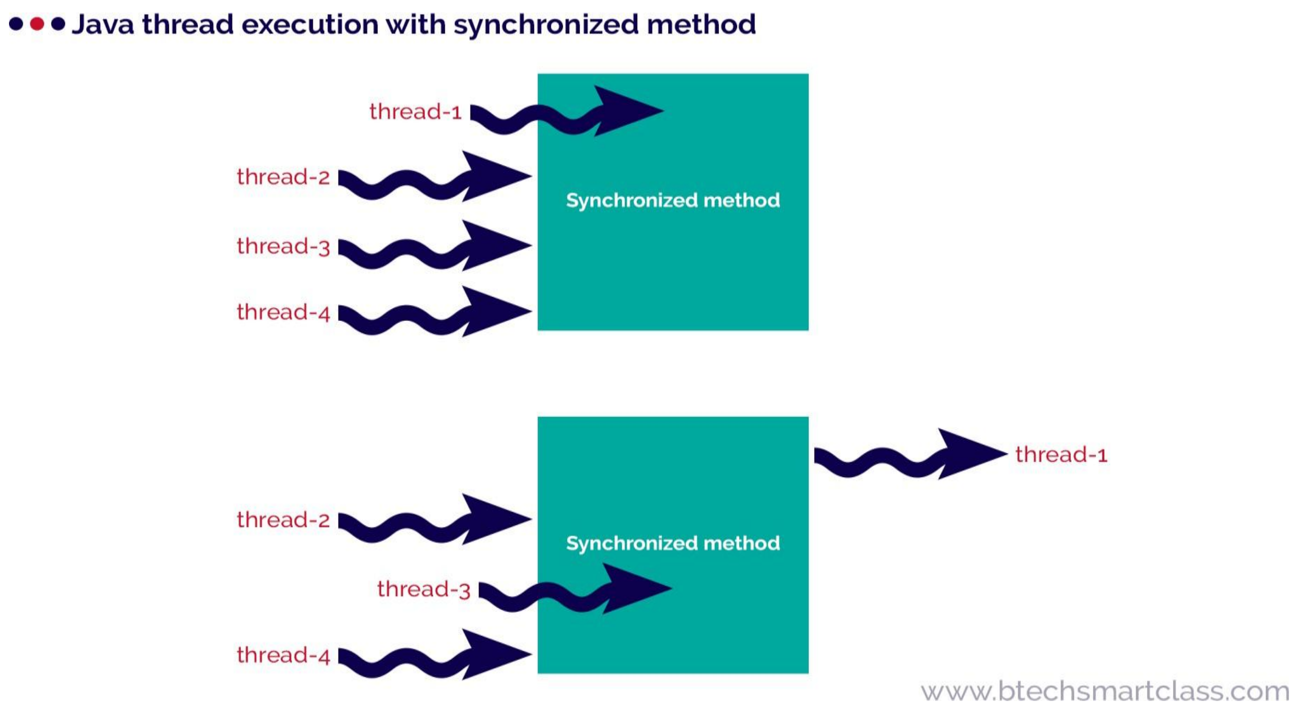
Mutual Exclusive

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

1. by synchronized method
2. by synchronized block

Java synchronized method

- If you declare any method as synchronized, it is known as synchronized method.
- When a method created using a synchronized keyword, it allows only one object to access it at a time.
- When an object calls a synchronized method, it put a lock on that method so that other objects or thread that are trying to call the same method must wait, until the lock is released.
- Once the lock is released on the shared resource, one of the threads among the waiting threads will be allocated to the shared resource.



- In the above image, initially the thread-1 is accessing the synchronized method and other threads (thread-2, thread-3, and thread-4) are waiting for the resource (synchronized method).
- When thread-1 completes it task, then one of the threads that are waiting is allocated with the synchronized method, in the above it is thread-3.

Example:

```
class Table
{
    synchronized void printTable(int n)
    {
            for(int i = 1; i <= 10; i++)
                    System.out.println(n + " * " + i + " = " + i*n);
    }
}
class MyThread1 extends Thread
{
    Table table = new Table();
    int number;
    MyThread1(Table table, int number)
    {
            this.table = table;
            this.number = number;
    }
    public void run()
    {
            table.printTable(number);
    }
}
class MyThread2 extends Thread
{
    Table table = new Table();
    int number;
    MyThread2(Table table, int number)
    {
            this.table = table;
            this.number = number;
    }
    public void run()
    {
            table.printTable(number);
    }
}
class ThreadSynchronizationExample
{
    public static void main(String[] args)
    {
            Table table = new Table();
```

```
                                        MyThread1 thread1 = new MyThread1(table, 5);
                                        MyThread2 thread2 = new MyThread2(table, 10);
                                        thread1.start();
                                        thread2.start();
                            }
                    }
```

Output:

```
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
10 * 1 = 10
10 * 2 = 20
10 * 3 = 30
10 * 4 = 40
10 * 5 = 50
10 * 6 = 60
10 * 7 = 70
10 * 8 = 80
10 * 9 = 90
10 * 10 = 100
```

Synchronized Block in Java

- The synchronized block is used when we want to synchronize only a specific sequence of lines in a method.
- For example, let's consider a method with 20 lines of code where we want to synchronize only a sequence of 5 lines code, we use the synchronized block.
- If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.
- Scope of synchronized block is smaller than the method.

Syntax to use synchronized block

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

Example of synchronized block

```
class Table
{
        void printTable(int n)
        {
                synchronized(this)
                { //synchronized block
                        for(int i=1;i<=5;i++)
                        {
                                System.out.println(n*i);
                                try
                                {
                                    Thread.sleep(400);
                                }
                               catch(Exception e)
                               {
                                    System.out.println(e);
                               }
                       }
                }
        }//end of the method
}
class MyThread1 extends Thread
{
    Table t;
    MyThread1(Table t)
    {
        this.t=t;
    }
    public void run()
    {
        t.printTable(5);
    }

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

Output:

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

INTERTHREAD COMMUNICATION

- Inter thread communication is the concept where two or more threads communicate to solve the problem of **polling**.
- In java, polling is the situation to check some condition repeatedly, to take appropriate action, once the condition is true.
- That means, in inter-thread communication, a thread waits until a condition becomes true such that other threads can execute its task.
- The inter-thread communication allows the synchronized threads to communicate with each other.

Java provides the following methods to achieve inter thread communication.

Method	**Description**
void wait( )	It makes the current thread to pause its execution until other thread in the same monitor calls notify( )
void notify( )	It wakes up the thread that called wait( ) on the same object.
void notifyAll()	It wakes up all the threads that called wait( ) on the same object.

Let's look at an example problem of producer and consumer.

- The producer produces the item and the consumer consumes the same.
- But here, the consumer can not consume until the producer produces the item, and producer can not produce until the consumer consumes the item that already been produced.
- So here, the consumer has to wait until the producer produces the item, and the producer also needs to wait until the consumer consumes the same.
- Here we use the inter-thread communication to implement the producer and consumer problem.

Example

```
class ItemQueue
{
        int item;
        boolean valueSet = false;
        synchronized int getItem()
        {
                 while (!valueSet)
                 try
                 {
                      wait();
                 }
                catch (InterruptedException e)
                {
                      System.out.println("InterruptedException  caught");
                }
                 System.out.println("Consumed:" + item);
                 valueSet = false;
                 try
                 {
                        Thread.sleep(1000);
                 }
                catch (InterruptedException e)
                {
                        System.out.println("InterruptedException  caught");
                }
                 notify();
                 return item;
        }
        synchronized void putItem(int item)
        {
                 while (valueSet)
                 try
                 {
                       wait();
                 }
                catch (InterruptedException e)
                {
                       System.out.println("InterruptedException  caught");
                 }
                 this.item = item;
                 valueSet = true;
                 System.out.println("Produced: " + item);
                 try
                 {
                        Thread.sleep(1000);
                 }
                catch (InterruptedException e)
                {
                    System.out.println("InterruptedException  caught");
                }
                notify();
        }
}
class Producer implements Runnable
{
        ItemQueue itemQueue;
        Producer(ItemQueue itemQueue)
        {
                this.itemQueue = itemQueue;
                new Thread(this, "Producer").start();
        }
        public void run()
        {
                int i = 0;
                while(true)
                {
                        itemQueue.putItem(i++);
                }
        }
}
class Consumer implements Runnable
{
        ItemQueue itemQueue;
        Consumer(ItemQueue itemQueue)
        {
                this.itemQueue = itemQueue;
                new Thread(this, "Consumer").start();
        }
        public void run()
        {
                while(true)
                {
                        itemQueue.getItem();
                }
        }
}
class ProducerConsumer
{
        public static void main(String args[])
        {
                ItemQueue itemQueue = new ItemQueue();
                new Producer(itemQueue);
                new Consumer(itemQueue);
        }
}
```

THREADGROUP IN JAVA

- 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.
- ava thread group is implemented by *java.lang.ThreadGroup* class.
- A ThreadGroup represents a set of threads. A thread group can also include the other thread group. The thread group creates a tree in which every thread group except the initial thread group has a parent.
- A thread is allowed to access information about its own thread group, but it cannot access the information about its thread group's parent thread group or any other thread groups.

ThreadGroup Example

```
public class ThreadGroupDemo implements Runnable
{
    public void run()
    {
            System.out.println(Thread.currentThread().getName());
    }
    public static void main(String[] args)
    {
            ThreadGroupDemo r = new ThreadGroupDemo();
            ThreadGroup tg1 = new ThreadGroup("Parent ThreadGroup");
            Thread t1 = new Thread(tg1, r,"one");
             t1.start();
            Thread t2 = new Thread(tg1, r,"two");
             t2.start();
             Thread t3 = new Thread(tg1, r,"three");
             t3.start();
             System.out.println("Thread Group Name: "+tg1.getName());
    }
 }
```

Output

```
one
two
three
Thread Group Name: Parent ThreadGroup
```

DAEMON THREAD IN JAVA

- In Java, daemon threads are low-priority threads that run in the background to perform tasks such as garbage collection or provide services to user threads.
- The life of a daemon thread depends on the mercy of user threads, meaning that when all user threads finish their execution, the Java Virtual Machine (JVM) automatically terminates the daemon thread.
- To put it simply, daemon threads serve user threads by handling background tasks and have no role other than supporting the main execution.

Methods for Java Daemon thread by Thread class

The java.lang.Thread class provides two methods for java daemon thread.

No.	Method	Description
1	public void setDaemon(boolean status)	is used to mark the current thread as daemon thread or user thread.
2	public boolean isDaemon()	is used to check that current is daemon.

Example

```
public class TestDaemonThread1 extends Thread
{
    public void run()
    {
            if(Thread.currentThread().isDaemon())
            {
                    System.out.println("daemon thread work");
            }
            else
            {
                    System.out.println("user thread work");
            }
    }
    public static void main(String[] args)
    {
            TestDaemonThread1 t1=new TestDaemonThread1();    //creating thread
            TestDaemonThread1 t2=new TestDaemonThread1();
            TestDaemonThread1 t3=new TestDaemonThread1();
            t1.setDaemon(true);                          //now t1 is daemon thread
            t1.start();                                  //starting threads
            t2.start();
            t3.start();
    }
}
```

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) , directions (NORTH, SOUTH, EAST, and WEST), season (SPRING, SUMMER, WINTER, and AUTUMN or FALL), colors (RED, YELLOW, BLUE, GREEN, WHITE, and BLACK) etc.
- According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.
- Java Enums can be thought of as classes which have a fixed set of constants (a variable that does not change).
- The Java enum constants are static and final implicitly.
- Enums are used to create our own data type like classes.
- The **enum** data type (also known as Enumerated Data Type) is used to define an enum in Java.
- Java Enum internally inherits the *Enum class*, so it cannot inherit any other class, but it can implement many interfaces.
- We can have fields, constructors, methods, and main methods in Java enum.

Example

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

AUTOBOXING

- The automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing and opposite operation is known as unboxing.
- So java programmer doesn't need to write the conversion code.

Advantage

No need of conversion between primitives and Wrappers manually so less coding is required.

Example

```
     class BoxingExample1
     {
                public static void main(String args[])
                {
                        int a=50;
                        Integer a2=new Integer(a);  //Boxing
                        Integer a3=5; //Boxing
                        System.out.println(a2+" "+a3);
                }
    }
```

Output: 50 5

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.
- Annotations in Java are used to provide additional information, so it is an alternative option for XML and Java marker interfaces.

Example

```
@Override
@SuppressWarnings
@Deprecated
```

@Override

- @Override annotation assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.
- Sometimes, we does the silly mistake such as spelling mistakes etc. So, it is better to mark @Override annotation that provides assurity that method is overridden.

Example

```
class Animal
{
	void eatSomething()
	{
		System.out.println("eating  something");}
	}
	class Dog extends Animal
	{
		@Override
		void eatsomething()
		{
			System.out.println("eating foods");
		}	//should be eatSomething
	}

class TestAnnotation1
{
	public static void main(String args[])
	{
		Animal a=new Dog();
		a.eatSomething();
	}
}
```

Output: Comple Time Error

@Deprecated

@Deprecated annotation marks that this method is deprecated so compiler prints warning. It informs user that it may be removed in the future versions. So, it is better not to use such methods.

Example

```
class A
{
        void m()
        {
                System.out.println("hello m");
        }
        @Deprecated
        void n()
        {
                System.out.println("hello n");
        }
}
class TestAnnotation3
{
        public static void main(String args[])
        {
                A a=new A();
                a.n();
        }

}
```

Output

At Compile Time:

```
Note: Test.java uses or overrides a deprecated API.

Note: Recompile with -Xlint:deprecation for details.
```

At Runtime:

```
hello n
```

JAVA GENERICS

- Java Generics allows us to create a single class, interface, and method that can be used with different types of data (objects).
- This helps us to reuse our code.
- **Note**: **Generics** does not work with primitive types (int, float, char, etc).

UNIT - IV

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.

EVENT HANDLING

- In general we can not perform any operation on dummy GUI screen even any button click or select any item.
- To perform some operation on these dummy GUI screen you need some predefined classes and interfaces.
- All these type of classes and interfaces are available in **java.awt.event** package.
- Changing the state of an object is known as an **event**.
- The process of handling the request in GUI screen is known as **event handling** (event represent an action). It will be changes component to component.

Note: In event handling mechanism event represent an action class and Listener represent an interface. Listener interface always contains abstract methods so here you need to write your own logic.

EVENTS

- The Events are the objects that define state change in a source.
- An event can be generated as a reaction of a user while interacting with GUI elements.
- Some of the event generation activities are moving the mouse pointer, clicking on a button, pressing the keyboard key, selecting an item from the list, and so on.
- We can also consider many other user operations as events.

EVENT SOURCES

- A source is an object that causes and generates an event.
- It generates an event when the internal state of the object is changed.
- The sources are allowed to generate several different types of events.
- A source must register a listener to receive notifications for a specific event.
- Each event contains its registration method.

Syntax

```
public void addTypeListener (TypeListener e1)
```

- From the above syntax, the Type is the name of the event, and e1 is a reference to the event listener.
- For example, for a keyboard event listener, the method will be called as addKeyListener().
- For the mouse event listener, the method will be called as addMouseMotionListener().
- When an event is triggered using the respected source, all the events will be notified to registered listeners and receive the event object.
- This process is known as event multicasting.

EVENT LISTENERS

- It is also known as event handler.
- Listener is responsible for generating response to an event.
- From java implementation point of view the listener is also an object.
- Listener waits until it receives an event.
- Once the event is received, the listener process the event and then returns.

EVENT CLASSES AND LISTENER INTERFACES

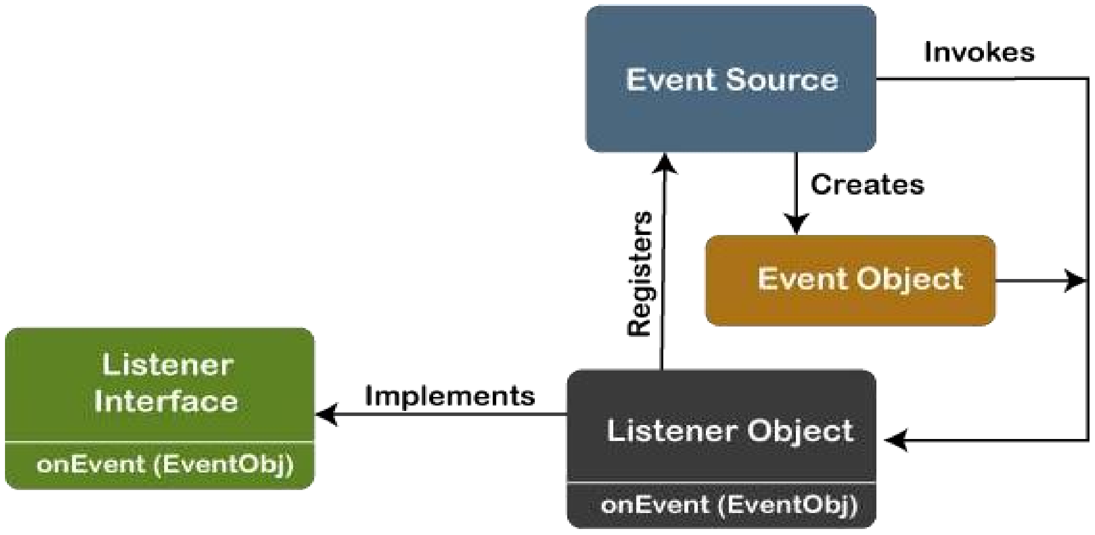
Event Classes	Description	Listener Interface
ActionEvent	generated when button is pressed, menu-item is selected, list-item is double clicked	ActionListener
MouseEvent	generated when mouse is dragged, moved,clicked,pressed or released and also when it enters or exit a component	MouseListener
KeyEvent	generated when input is received from keyboard	KeyListener
ItemEvent	generated when check-box or list item is clicked	ItemListener
TextEvent	generated when value of textarea or textfield is changed	TextListener
MouseWheelEvent	generated when mouse wheel is moved	MouseWheelListener
WindowEvent	generated when window is activated, deactivated, deiconified, iconified, opened or closed	WindowListener
ComponentEvent	generated when component is hidden,moved,resized or set visible	ComponentEventListener

ContainerEvent	generated when component is added or removed from container	ContainerListener
AdjustmentEvent	generated when scroll bar is manipulated	AdjustmentListener
FocusEvent	generated when component gains or loses keyboard focus	FocusListener

DELEGATION EVENT MODEL IN JAVA

- The Delegation Event model is defined to handle events in GUI programming languages.
- The GUI stands for Graphical User Interface, where a user graphically/visually interacts with the system.
- The GUI programming is inherently event-driven; whenever a user initiates an activity such as a mouse activity, clicks, scrolling, etc., each is known as an event that is mapped to a code to respond to functionality to the user. This is known as event handling.

The below image demonstrates the event processing.



- In this model, a source generates an event and forwards it to one or more listeners.
- The listener waits until it receives an event. Once it receives the event, it is processed by the listener and returns it.

REGISTRATION METHODS

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

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)
{
}
```

STEPS TO PERFORM EVENT HANDLING

Following steps are required to perform event handling:

- Implement the Listener interface and overrides its methods
- Register the component with the Listener
- The User clicks the button and the event is generated.
- Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
- Event object is forwarded to the method of registered listener class.
- The method is now get executed and returns.

Syntax to Handle the Event

```
class className implements XXXListener
{
    .......
    .......
}
addcomponentobject.addXXXListener(this);
.......
// override abstract method of given interface and write proper logic
public void methodName(XXXEvent e)
{
    .......
    .......
}
     .......
}
```

EVENT HANDLING FOR MOUSE

For handling event for mouse you need MouseEvent class and MouseListener interface.

GUI Component	Event class	Listener Interface
Mouse	MouseEvent	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.

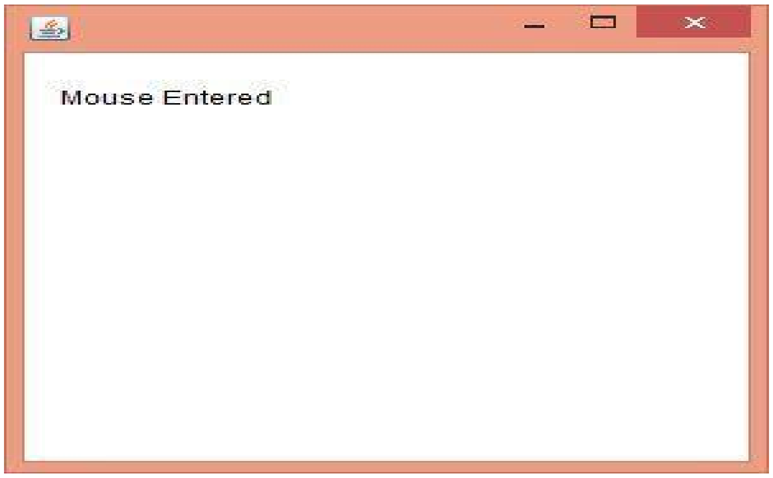
Methods of MouseListener interface

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

1. public abstract void mouseClicked(MouseEvent e);
2. public abstract void mouseEntered(MouseEvent e);
3. public abstract void mouseExited(MouseEvent e);
4. public abstract void mousePressed(MouseEvent e);
5. 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();
        }
}
```
Output:



```
Mouse Entered
```

EVENT HANDLING FOR KEYBOARD

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. It has three methods.

Methods of KeyListener interface

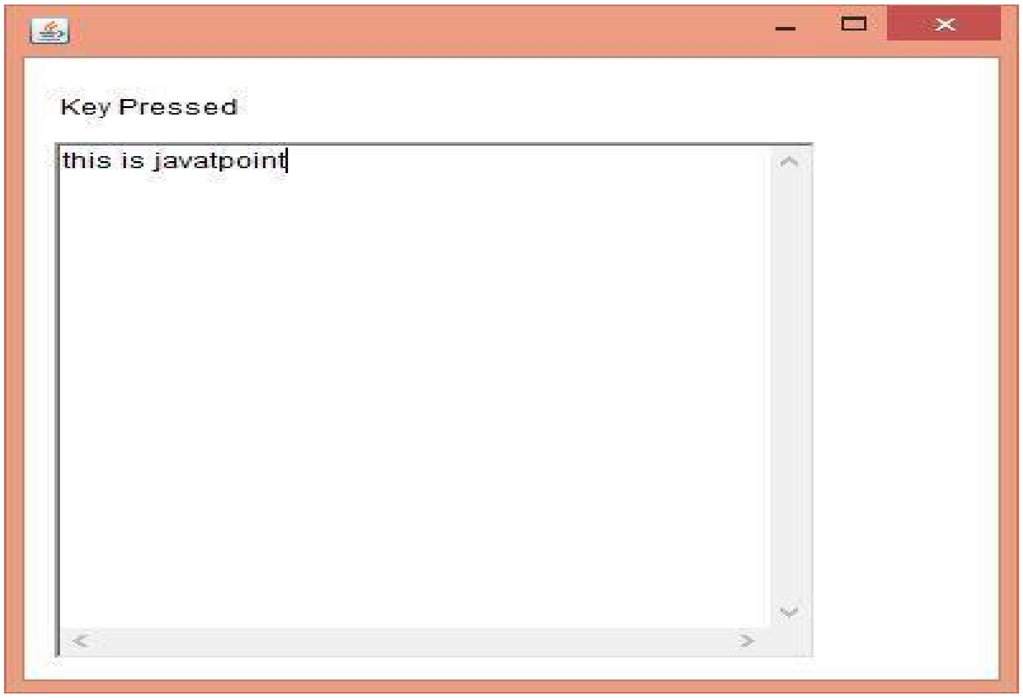
1. public abstract void keyPressed(KeyEvent e);
2. public abstract void keyReleased(KeyEvent e);
3. public abstract void keyTyped(KeyEvent e);

EXAMPLE

```
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();
   }
}
```

Output:



```
Key Pressed
this is javatpoint
```

ADAPTER CLASSES

- In a program, when a listener has many abstract methods to override, it becomes complex for the programmer to override all of them.
- For example, for closing a frame, we must override seven abstract methods of WindowListener, but we need only one method of them.
- For reducing complexity, Java provides a class known as "adapters" or adapter class.
- Adapters are abstract classes, that are already being overriden.

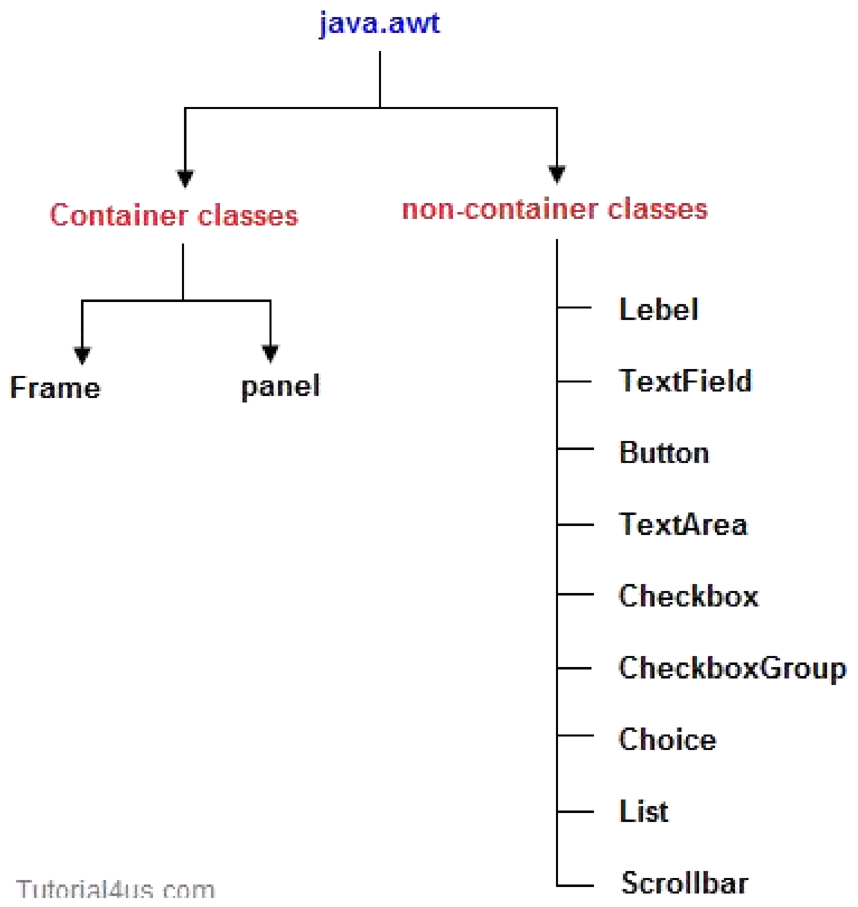
Adapter class	**Listener interface**
WindowAdapter	WindowListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
FocusAdapter	FocusListener
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
HierarchyBoundsAdapter	HierarchyBoundsListener

Java WindowAdapter Example

```
import java.awt.*;
import java.awt.event.*;
public class AdapterExample
{
          Frame f;
          AdapterExample()
          {
                f=new Frame("Window Adapter");
                f.addWindowListener(new WindowAdapter()
                {
                  public void windowClosing(WindowEvent e)
                  {
                    f.dispose();
                  }
                });
                f.setSize(400,400);
                f.setVisible(true);
          }
        public static void main(String[] args)
        {
             new AdapterExample();
        }
}
```

JAVA AWT HIERARCHY

The hierarchy of Java AWT classes are given below.



Components

All the elements like the button, text fields, scroll bars, etc. are called components. In Java AWT, there are classes for each component as shown in above diagram. In order to place every component in a particular position on a screen, we need to add them to a container.

Container

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

AWT FRAME

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

```
Frame f=new Frame();
```

Methods

1. setTitle()

It is used to display user defined message on title bar.

```
Frame f=new Frame();
f.setTitle("myframe");
```

2. setBackground()

It is used to set background or image of frame.

```
Frame f=new Frame();
f.setBackground(Color.red);
```

3. setForground()

It is used to set the foreground text color.

```
Frame f=new Frame();
f.setForground(Color.red);
```

4. setSize()

It is used to set the width and height for frame.

```
Frame f=new Frame();
f.setSize(400,300);
```

5. setVisible()

It is used to make the frame as visible to end user.

```
Frame f=new Frame();
f.setVisible(true);
```

Note: You can write setVisible(true) or setVisible(false), if it is true then it visible otherwise not visible.

7.add()

It is used to add non-container components (Button, List) to the frame.

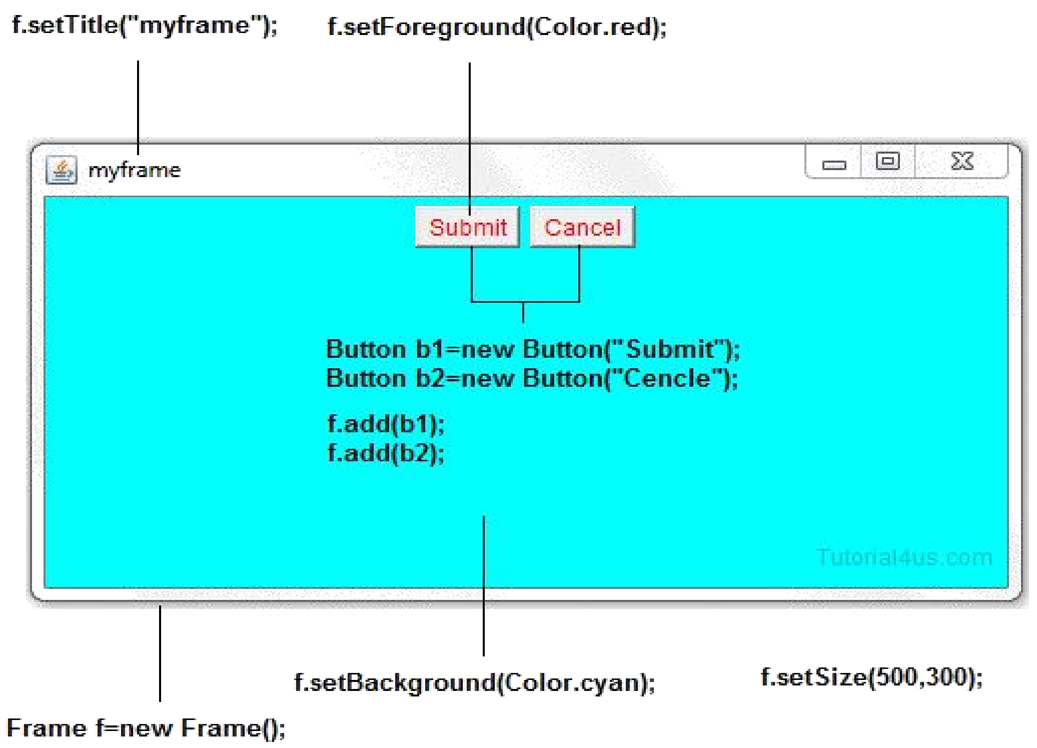
```
Frame f=new Frame();
Button b=new Button("Click");
f.add(b);
```

Explanation: In above code we add button on frame using f.add(b), here b is the object of Button class..

Example

```
import java.awt.*;
class FrameDemo
{
        public static void main(String[] args)
        {
                Frame f=new Frame();
                f.setTitle("myframe");
                f.setBackground(Color.cyan);
                f.setForeground(Color.red);
                f.setLayout(new FlowLayout());
                Button b1=new Button("Submit");
                Button b2=new Button("Cancel");
                f.add(b1);f.add(b2);
                f.setSize(500,300);
                f.setVisible(true);
        }
  }
```

Output


f.setTitle("myframe");
f.setForeground(Color.red);
myframe
Submit
Cancel
Button b1=new Button("Submit");
Button b2=new Button("Cencle");
f.add(b1);
f.add(b2);
Tutorial4us.com
f.setBackground(Color.cyan);
f.setSize(500,300);
Frame f=new Frame();

AWT PANEL

It is a predefined class used to provide a logical container to hold various GUI component. Panel always should exist as a part of frame.

Note: Frame is always visible to end user where as panel is not visible to end user.

Panel is a derived class of container class so you can use all the methods which is used in frame.

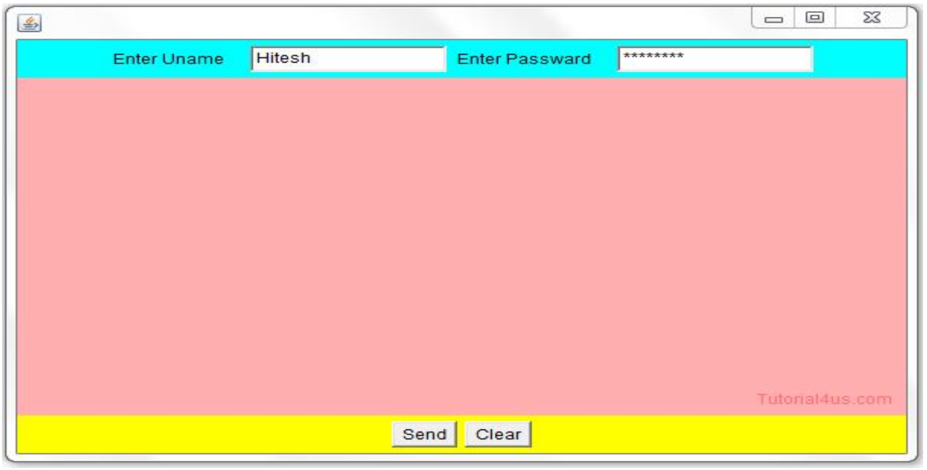
Syntax

```
Panel p=new Panel();
```

Example

```
import java.awt.*;
class PanelFrame
{
        PanelFrame()
        {
                Frame f=new Frame();
                f.setSize(600,400);
                f.setBackground(Color.pink);
                f.setLayout(new BorderLayout());
                Panel p1=new Panel();
                p1.setBackground(Color.cyan);
                Label l1 =new Label("Enter Uname");
                TextField tf1=new TextField(15);
                Label l2=new Label("Enter Passward");
                TextField tf2=new TextField(15);
                p1.add(l1);
                p1.add(tf1);
                p1.add(l2);
                p1.add(tf2);
                f.add("North",p1);
                Panel p2=new Panel();
                p2.setBackground(Color.yellow);
                Button b1=new Button("Send");
                Button b2=new Button("Clear");
                p2.add(b1);
                p2.add(b2);
                f.add("South",p2);
                f.setVisible(true);}
                public static void main(String[] args)
                {
                        PanelFrame pf=new PanelFrame();
                }
}
```

Output:



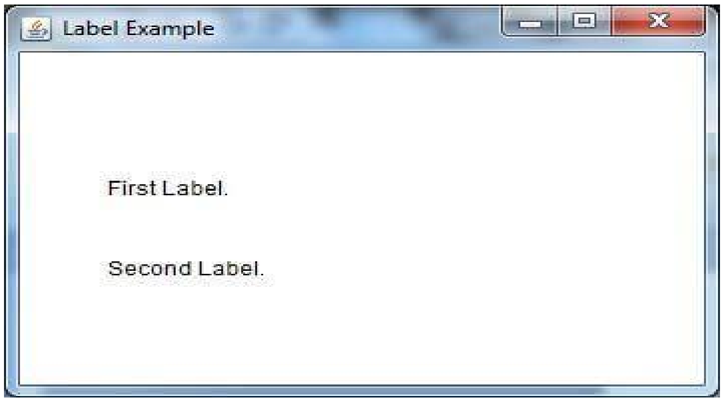
AWT Label

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. The text can be changed by a programmer but a user cannot edit it directly.

Example

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

Output



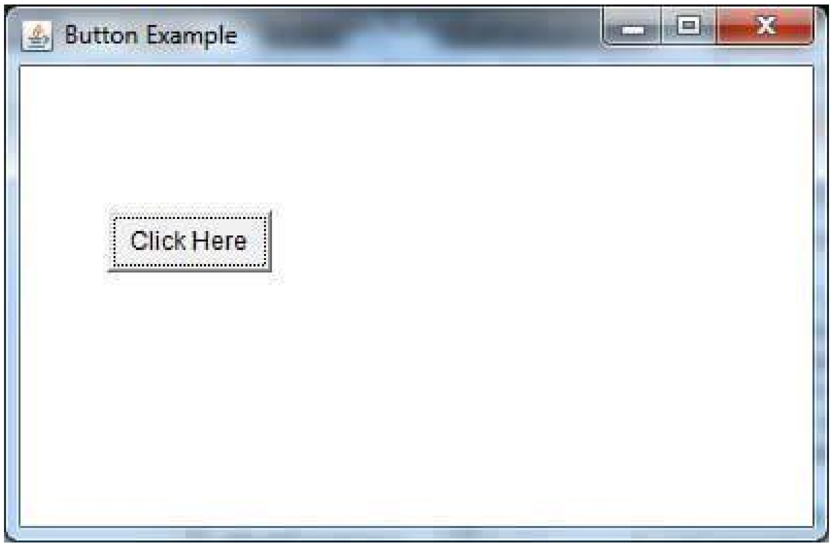
Java AWT Button

A button is basically a control component with a label that generates an event when pushed. The Button class is used to create a labeled button that has platform independent implementation. The application result in some action when the button is pushed.

Example

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

Output:



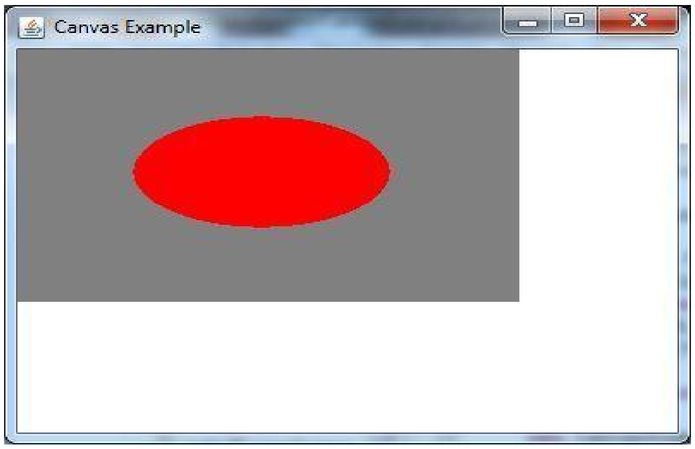
Java AWT Canvas

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.

Example

```
import java.awt.*;
public class CanvasExample
{
     public CanvasExample()
    {
                Frame f = new Frame("Canvas Example");
                f.add(new MyCanvas());
                f.setLayout(null);
                f.setSize(400, 400);
                f.setVisible(true);
  }
  public static void main(String args[])
  {
                 new CanvasExample();
  }
}
class MyCanvas extends Canvas
{
       public MyCanvas()
       {
                setBackground (Color.GRAY);
                 setSize(300, 200);
       }
     public void paint(Graphics g)
    {
                g.setColor(Color.red);
                g.fillOval(75, 75, 150, 75);
    }
}
```

Output:



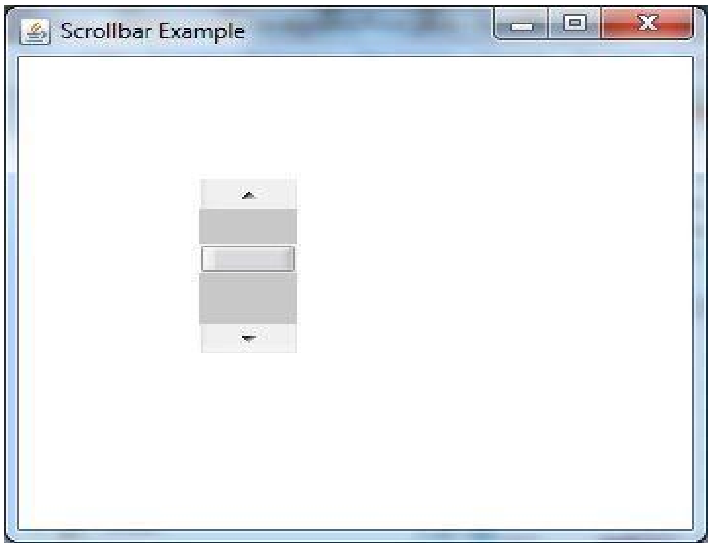
Java AWT Scrollbar

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.

Example

```
import java.awt.*;
public class ScrollbarExample1
{
    ScrollbarExample1()
    {
            Frame f = new Frame("Scrollbar Example");
            Scrollbar s = new Scrollbar();
            s.setBounds (100, 100, 50, 100);
            f.add(s);
            f.setSize(400, 400);
            f.setLayout(null);
            f.setVisible(true);
    }
    public static void main(String args[])
    {
        new ScrollbarExample1();
    }
}
```

Output:



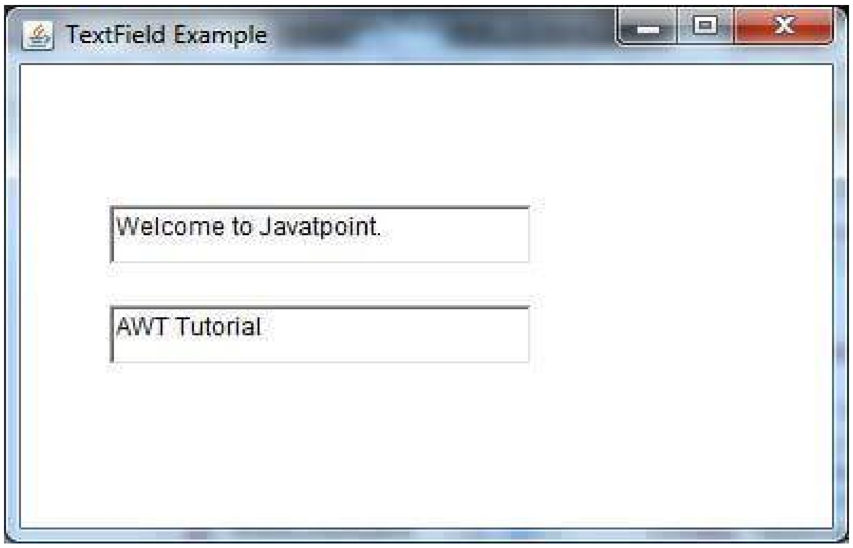
Java AWT TextField

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.

Exmple:

```
import java.awt.*;
public class TextFieldExample1
{
    public static void main(String args[])
    {
            Frame f = new Frame("TextField Example");
            TextField t1, t2;
            t1 = new TextField("Welcome to Javatpoint.");
            t1.setBounds(50, 100, 200, 30);
            t2 = new TextField("AWT Tutorial");
            t2.setBounds(50, 150, 200, 30);
            f.add(t1);
            f.add(t2);
            f.setSize(400,400);
            f.setLayout(null);
            f.setVisible(true);
    }
}
```

Output:



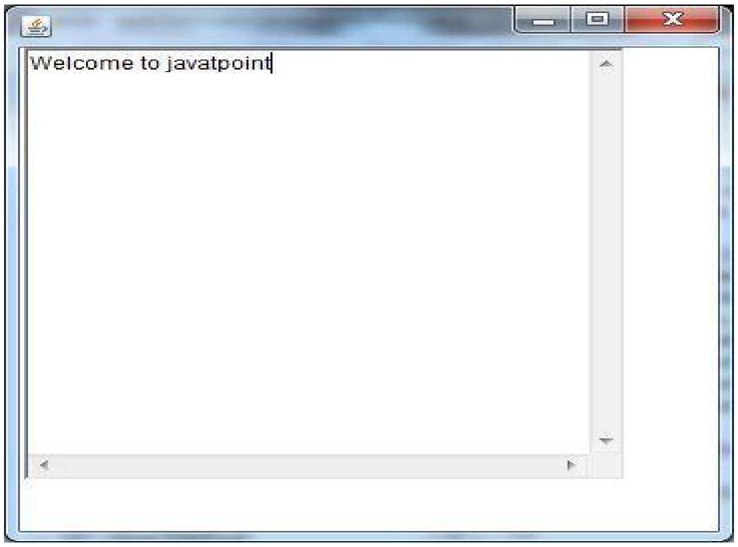
Java AWT TextArea

The object of a TextArea class is a multiline region that displays text. It allows the editing of multiple line text. It inherits TextComponent class.

Example

```
import java.awt.*;
public class TextAreaExample
{
        TextAreaExample()
        {
                Frame f = new Frame();
                TextArea area = new TextArea("Welcome to javatpoint");
                area.setBounds(10, 30, 300, 300);
                f.add(area);
                f.setSize(400, 400);
                f.setLayout(null);
                f.setVisible(true);
        }
        public static void main(String args[])
        {
                new TextAreaExample();
        }
}
```

Output



Welcome to javatpoint

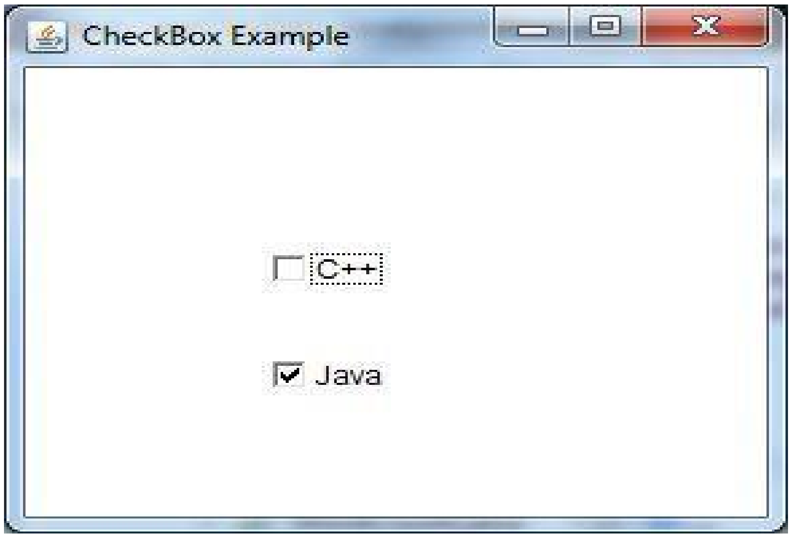
Java AWT Checkbox

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".

Example

```
import java.awt.*;
public class CheckboxExample1
{
        CheckboxExample1()
        {
                Frame f = new Frame("Checkbox Example");
                Checkbox checkbox1 = new Checkbox("C++");
                checkbox1.setBounds(100, 100,  50, 50);
                Checkbox checkbox2 = new Checkbox("Java", true);
                checkbox2.setBounds(100, 150,  50, 50);
                f.add(checkbox1);
                f.add(checkbox2);
                f.setSize(400,400);
                f.setLayout(null);
                f.setVisible(true);
        }
        public static void main (String args[])
        {
                new CheckboxExample1();
        }
}
```

Output



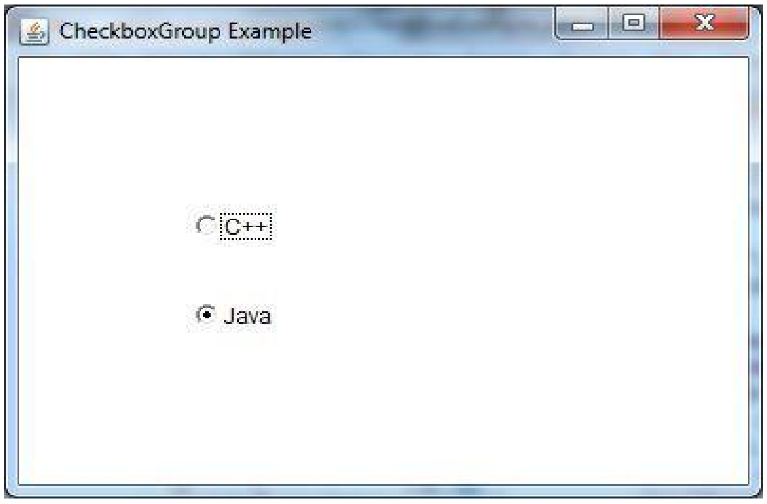
Java AWT CheckboxGroup

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.

Example

```
import java.awt.*;
public class CheckboxGroupExample
{
     CheckboxGroupExample()
    {
            Frame f= new Frame("CheckboxGroup Example");
            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);
            f.add(checkBox1);
            f.add(checkBox2);
            f.setSize(400,400);
            f.setLayout(null);
            f.setVisible(true);
    }
    public static void main(String args[])
    {
            new CheckboxGroupExample();
    }
}
```

Output



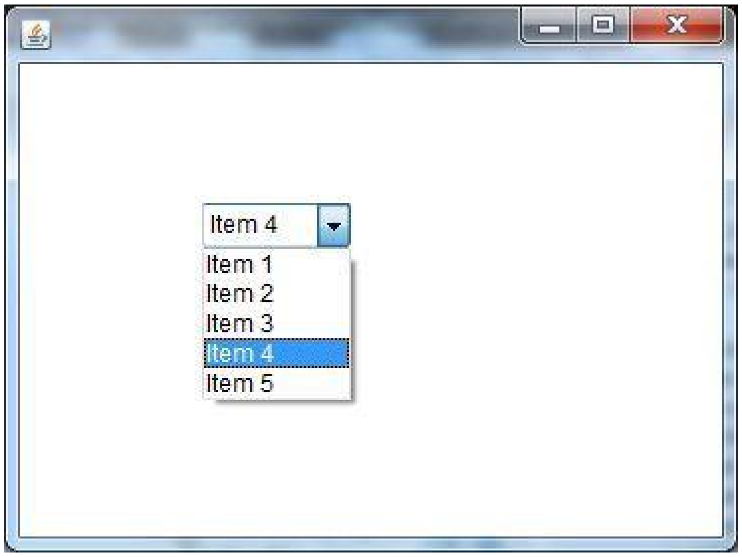
Java AWT Choice

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.

Example

```
import java.awt.*;
public class ChoiceExample1
{
        ChoiceExample1()
        {
                Frame f = new Frame();
                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");
                f.add(c);
                f.setSize(400, 400);
                f.setLayout(null);
                f.setVisible(true);
        }
        public static void main(String args[])
        {
                new ChoiceExample1();
        }
}
```

Output



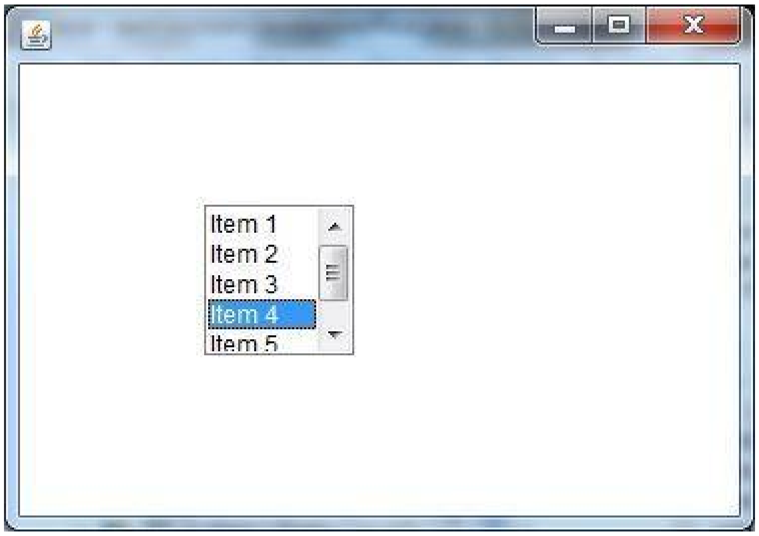
Java AWT List

The object of List class represents a list of text items. With the help of the List class, user can choose either one item or multiple items. It inherits the Component class.

Example

```
import java.awt.*;
public class ListExample1
{
        ListExample1()
        {
                Frame f = new Frame();
                List l1 = new List(5);
                l1.setBounds(100, 100, 75, 75);
                l1.add("Item 1");
                l1.add("Item 2");
                l1.add("Item 3");
                l1.add("Item 4");
                l1.add("Item 5");
                f.add(l1);
                f.setSize(400, 400);
                f.setLayout(null);
                f.setVisible(true);
        }
        public static void main(String args[])
        {
                new ListExample1();
        }
}
```

Output



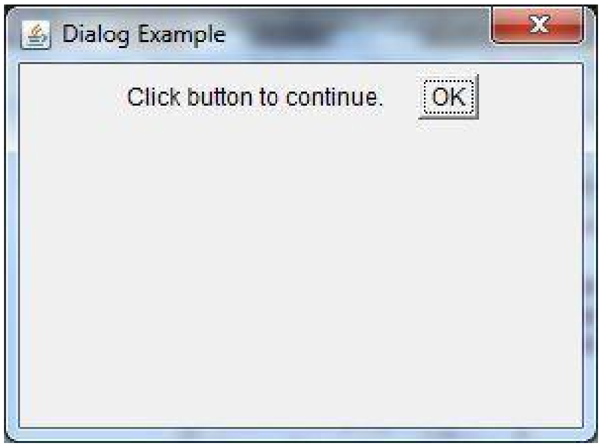
Java AWT Dialog

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.

Example

```
import java.awt.*;
import java.awt.event.*;
public class DialogExample
{
    private static Dialog d;
    DialogExample()
    {
            Frame f= new Frame();
            d = new Dialog(f , "Dialog Example", true);
            d.setLayout( new FlowLayout() );
            Button b = new Button ("OK");
            b.addActionListener ( new ActionListener()
            {
                    public void actionPerformed( ActionEvent e )
                    {
                            DialogExample.d.setVisible(false);
                    }
            });
            d.add( new Label ("Click button to continue."));
            d.add(b);
            d.setSize(300,300);
            d.setVisible(true);
    }
    public static void main(String args[])
    {
            new DialogExample();
    }
}
```

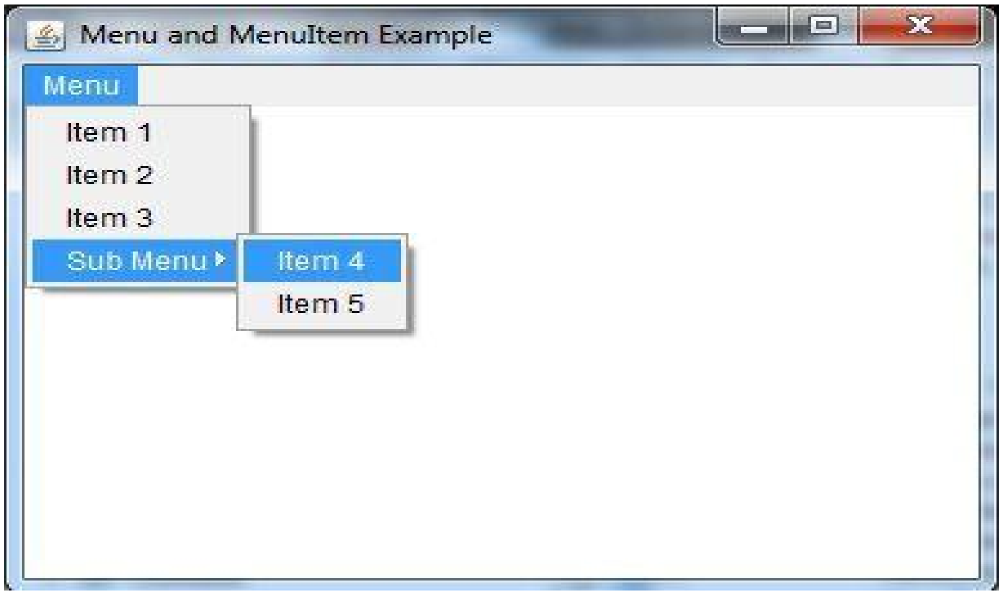
Output



Java AWT Menubar

```
import java.awt.*;
class MenuExample
{
    MenuExample()
    {
            Frame f= new Frame("Menu and MenuItem Example");
            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");
            MenuItem i3=new MenuItem("Item 3");
            MenuItem i4=new MenuItem("Item 4");
            MenuItem i5=new MenuItem("Item 5");
            menu.add(i1);
            menu.add(i2);
            menu.add(i3);
            submenu.add(i4);
            submenu.add(i5);
            menu.add(submenu);
            mb.add(menu);
            f.setMenuBar(mb);
            f.setSize(400,400);
            f.setLayout(null);
            f.setVisible(true);
    }
    public static void main(String args[])
    {
            new MenuExample();
    }
}
```

Output



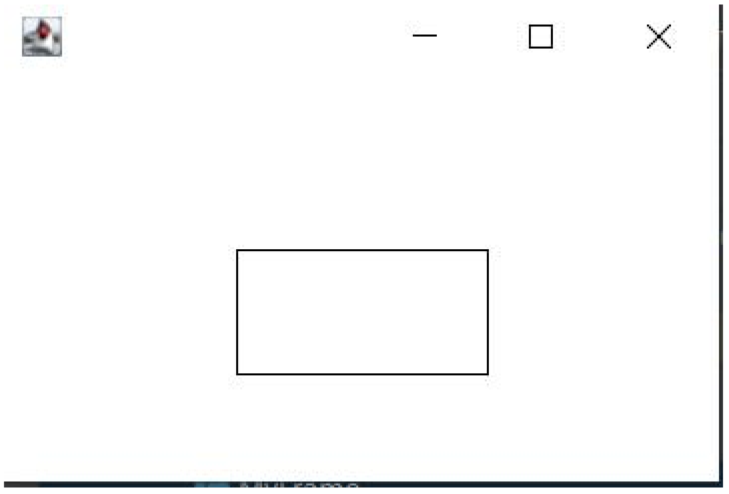
Java AWT Graphics

Graphics is an abstract class provided by Java AWT which is used to draw or paint on the components. It consists of various fields which hold information like components to be painted, font, color, XOR mode, etc., and methods that allow drawing various shapes on the GUI components.

Example

```
import java.awt.*;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
public class MyFrame extends Frame
{
        public MyFrame()
        {
                setVisible(true);
                setSize(300, 200);
                addWindowListener(new WindowAdapter()
                {
                        @Override
                        public void windowClosing(WindowEvent e)
                        {
                                System.exit(0);
                        }
                });
   }
   public void paint(Graphics g)
   {
         g.drawRect(100, 100, 100, 50);
   }
    public static void main(String[] args)
   {
        new MyFrame();
   }
}
```

Output



JAVA LAYOUT MANAGERS

- The Layout Managers are used to arrange components in a particular manner.
- Layout Manager is an interface that is implemented by all the classes of layout managers.

There are following classes that represents the layout managers:

1. BorderLayout
2. FlowLayout
3. GridLayout
4. CardLayout
5. GridBagLayout

BORDERLAYOUT

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

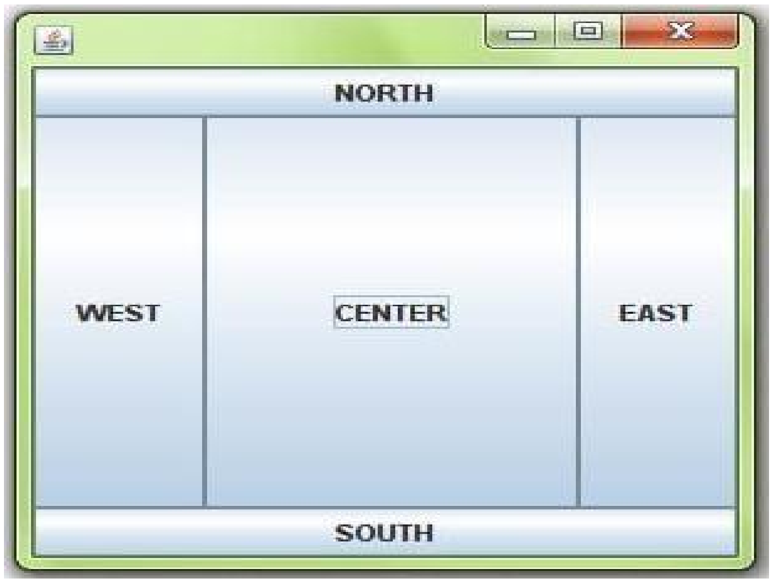
The BorderLayout provides five constants for each region:

- **public static final int NORTH**
- **public static final int SOUTH**
- **public static final int EAST**
- **public static final int WEST**
- **public static final int CENTER**

EXAMPLE

```
import java.awt.*;

import javax.swing.*;

public class Border
{
        Border()
        {
                JFrame f=new JFrame();
                JButton b1=new JButton("NORTH");

                JButton b2=new JButton("SOUTH");

                JButton b3=new JButton("EAST");

                JButton b4=new JButton("WEST");

                JButton b5=new JButton("CENTER");

                f.add(b1,BorderLayout.NORTH);

                f.add(b2,BorderLayout.SOUTH);

                f.add(b3,BorderLayout.EAST);
                f.add(b4,BorderLayout.WEST);
                f.add(b5,BorderLayout.CENTER);
                f.setSize(300,300);
                f.setVisible(true);
        }
        public static void main(String[] args)
        {
            new Border();
        }
}
```

Output:



FLOWLAYOUT

- This layout is used to arrange the GUI components in a sequential flow (that means one after another in horizontal way)
- You can also set flow layout of components like flow from left, flow from right.

FlowLayout Left

```
Frame f=new Frame();
f.setLayout(new FlowLayout(FlowLayout.LEFT));
```

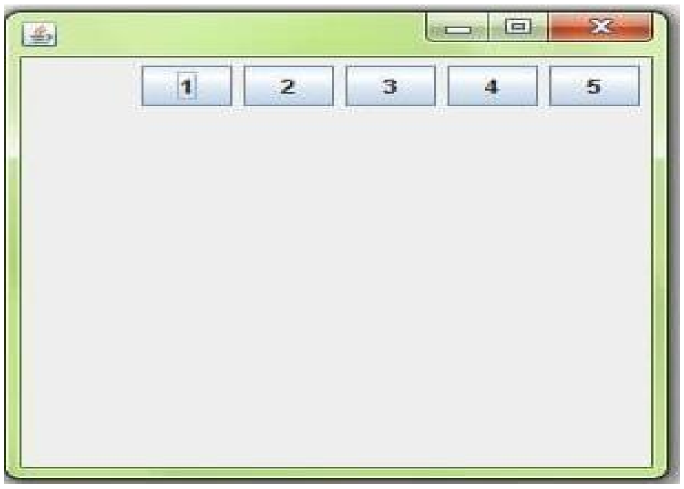
FlowLayout Right

```
Frame f=new Frame();
f.setLayout(new FlowLayout(FlowLayout.RIGHT))
```

EXAMPLE

```
import java.awt.*;

import javax.swing.*;

public class MyFlowLayout
{
          MyFlowLayout()
          {
             JFrame f=new JFrame();

             JButton b1=new JButton("1");

             JButton b2=new JButton("2");

             JButton b3=new JButton("3");

             JButton b4=new JButton("4");

             JButton b5=new JButton("5");

             f.add(b1);

            f.add(b2);

            f.add(b3);

            f.add(b4);

            f.add(b5);

            f.setLayout(new FlowLayout(FlowLayout.RIGHT));

            f.setSize(300,300);

            f.setVisible(true);
          }
          public static void main(String[] args)
          {
             new MyFlowLayout();
          }
}
```

Output:



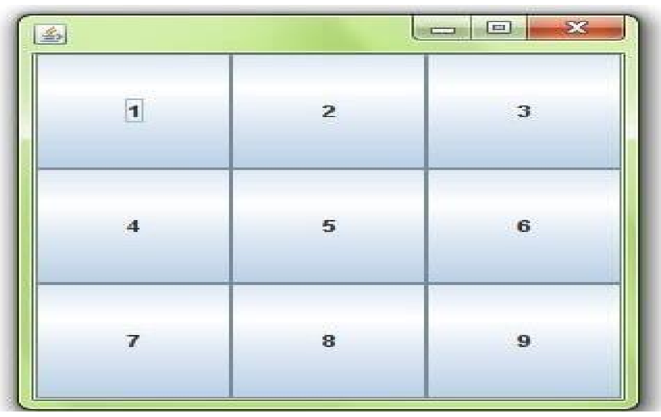
GRIDLAYOUT

This layout is used to arrange the GUI components in the table format.

EXAMPLE

```
import java.awt.*;
import javax.swing.*;
public class MyGridLayout
{
    MyGridLayout()
    {
        JFrame f=new JFrame();
        JButton b1=new JButton("1");
        JButton b2=new JButton("2");
        JButton b3=new JButton("3");
        JButton b4=new JButton("4");
        JButton b5=new JButton("5");
        JButton b6=new JButton("6");
        JButton b7=new JButton("7");
        JButton b8=new JButton("8");
        JButton b9=new JButton("9");
        f.add(b1);
         f.add(b2);
         f.add(b3);
         f.add(b4);
         f.add(b5);
         f.add(b6);
         f.add(b7);
         f.add(b8);
         f.add(b9);
         f.setLayout(new GridLayout(3,3));
        f.setSize(300,300);
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
        new MyGridLayout();
    }
}
```

Output



CARDLAYOUT

The 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.

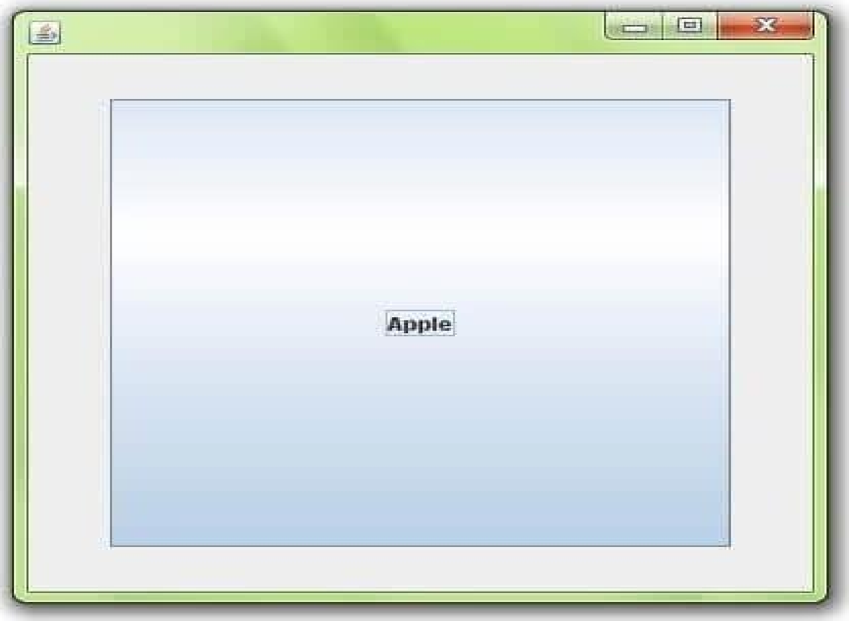
Commonly used methods of CardLayout class

1. public void next(Container parent): is used to flip to the next card of the given container.
2. public void previous(Container parent): is used to flip to the previous card of the given container.
3. public void first(Container parent): is used to flip to the first card of the given container.
4. public void last(Container parent): is used to flip to the last card of the given container.
5. public void show(Container parent, String name): is used to flip to the specified card with the given name.

EXAMPLE

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class CardLayoutExample extends JFrame implements ActionListener
{
        CardLayout card;
        JButton b1,b2,b3;
        Container c;
        CardLayoutExample()
        {
            c=getContentPane();
            card=new CardLayout(40,30);
           //create CardLayout object with 40 hor space and 30 ver space
            c.setLayout(card);
              b1=new JButton("Apple");
             b2=new JButton("Boy");
             b3=new JButton("Cat");
             b1.addActionListener(this);
             b2.addActionListener(this);
             b3.addActionListener(this);
             c.add("a",b1);
            c.add("b",b2);
            c.add("c",b3);
        }
        public void actionPerformed(ActionEvent e)
        {
                card.next(c);
        }
        public static void main(String[] args)
        {
                CardLayoutExample cl=new CardLayoutExample();
                cl.setSize(400,400);
                cl.setVisible(true);
                cl.setDefaultCloseOperation(EXIT_ON_CLOSE);
        }
}
```

Output:



GridBagLayout

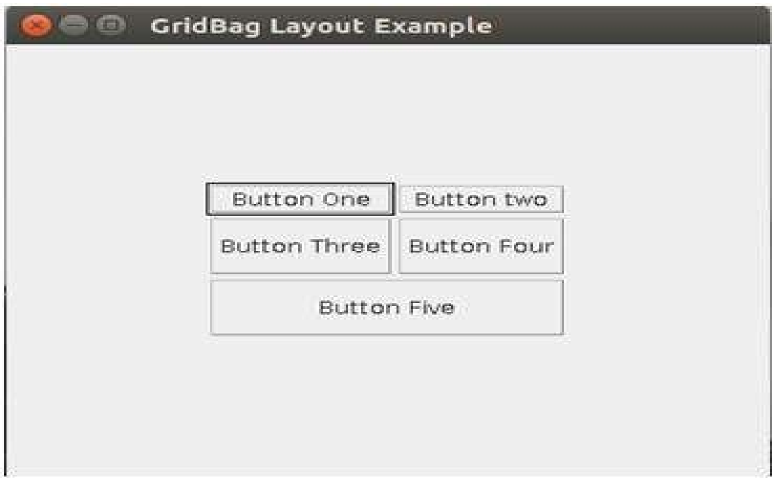
- The Java GridBagLayout class is used to align components vertically, horizontally or along their baseline.
- The components may not be of same size. Each GridBagLayout object maintains a dynamic, rectangular grid of cells.
- Each component occupies one or more cells known as its display area. Each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

Example

```
import java.awt.Button;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.*;
public class GridBagLayoutExample extends JFrame
{
    public static void main(String[] args)
   {
          GridBagLayoutExample a = new GridBagLayoutExample();
   }
   public GridBagLayoutExample()
   {
          GridBagLayoutgrid = new GridBagLayout();
          GridBagConstraints gbc = new GridBagConstraints();
          setLayout(grid);
          setTitle("GridBag Layout Example");
          GridBagLayout layout = new GridBagLayout();
          this.setLayout(layout);
           gbc.fill = GridBagConstraints.HORIZONTAL;
           gbc.gridx = 0;
           gbc.gridy = 0;
```

```
        this.add(new Button("Button One"), gbc);
        gbc.gridx = 1;
        gbc.gridy = 0;
        this.add(new Button("Button two"), gbc);
        gbc.fill = GridBagConstraints.HORIZONTAL;
        gbc.ipady = 20;
        gbc.gridx = 0;
        gbc.gridy = 1;
        this.add(new Button("Button Three"), gbc);
        gbc.gridx = 1;
        gbc.gridy = 1;
        this.add(new Button("Button Four"), gbc);
        gbc.gridx = 0;
        gbc.gridy = 2;
        gbc.fill = GridBagConstraints.HORIZONTAL;
        gbc.gridwidth = 2;
        this.add(new Button("Button Five"), gbc);
        setSize(300, 300);
        setPreferredSize(getSize());   setVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
  }
```

Output:



UNIT – V

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.

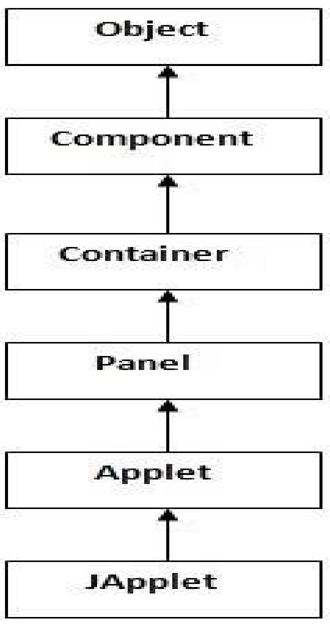
JAVA APPLET

- 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.
- Applets are used to make the website more dynamic and entertaining.
- An applet is embedded in an HTML page using the APPLET or OBJECT tag and hosted on a web server.

Important points

- All applets are sub-classes of java.applet.Applet class.
- Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer. JDK provides a standard applet viewer tool called applet viewer.
- In general, execution of an applet does not begin at main() method.
- Output of an applet window is not performed by System.out.println(). Rather it is handled with various AWT methods, such as drawString().

HIERARCHY OF APPLET



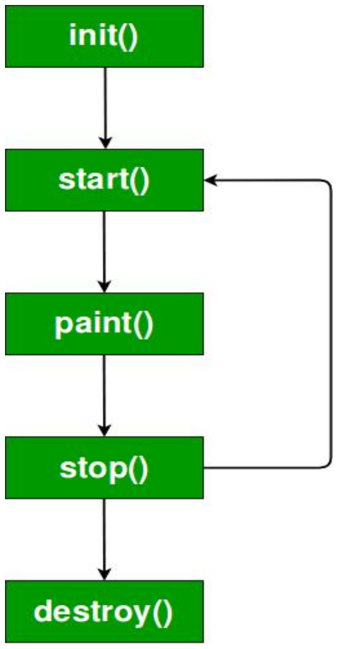
As displayed in the above diagram, Applet class extends Panel. Panel class extends Container which is the subclass of Component.

DIFFERENCES BETWEEN APPLETS AND APPLICATIONS

Parameters	Java Application	Java Applet
Definition	Applications are just like a Java program that can be executed independently without using the web browser.	Applets are small Java programs that are designed to be included with the HTML web document. They require a Java-enabled web browser for execution.
main() method	The application program requires a main() method for its execution.	The applet does not require the main() method for its execution instead init() method is required.
Compilation	The “javac” command is used to compile application programs, which are then executed using the “java” command.	Applet programs are compiled with the “javac” command and run using either the “appletviewer” command or the web browser.
File access	Java application programs have full access to the local file system and network.	Applets don’t have local disk and network access.
Access level	Applications can access all kinds of resources available on the system.	Applets can only access browser-specific services. They don’t have access to the local system.
Installation	First and foremost, the installation of a Java application on the local computer is required.	The Java applet does not need to be installed beforehand.
Execution	Applications can execute the programs from the local system.	Applets cannot execute programs from the local machine.
Program	An application program is needed to perform some tasks directly for the user.	An applet program is needed to perform small tasks or part of them.
Run	It cannot run on its own; it needs JRE to execute.	It cannot start on its own, but it can be executed using a Java-enabled web browser.
Connection with servers	Connectivity with other servers is possible.	It is unable to connect to other servers.
Read and Write Operation	It supports the reading and writing of files on the local computer.	It does not support the reading and writing of files on the local computer.
Security	Application can access the system’s data and resources without any security limitations.	Executed in a more restricted environment with tighter security. They can only use services that are exclusive to their browser.
Restrictions	Java applications are self-contained and require no additional security because they are trusted.	Applet programs cannot run on their own, necessitating the maximum level of security.

LIFE CYCLE OF AN APPLET

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



Lifecycle methods for Applet

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

java.applet.Applet class

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

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

java.awt.Component class

The Component class provides 1 life cycle method of applet.

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

CREATING APPLETS

How To Run an Applet?

There are two ways to run an applet

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

1. By html file

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

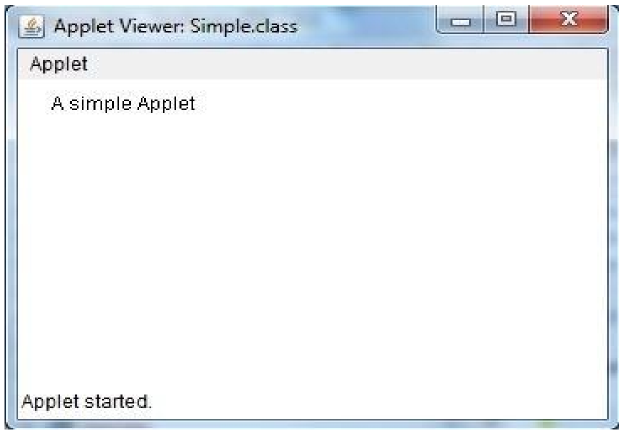
Example

```
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet
{
        public void paint(Graphics g)
        {
                g.drawString("A simple Applet",20,20);
        }
 }
```

myapplet.html

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

Output



```
A simple Applet
```

```
Applet started.
```

2. By appletViewer tool

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

Example

```
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
```

PARAMETER IN APPLET

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

Syntax:

```
public String getParameter(String parameterName)
```

Example of using parameter in Applet

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

myapplet.html

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

SWING INTRODUCTION

- Java Swing is used to create window-based applications. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.
- Unlike AWT, Java Swing provides platform-independent and lightweight components.
- The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

LIMITATIONS OF AWT

- The buttons of AWT does not support pictures.
- It is heavyweight in nature.
- Two very important components trees and tables are not present.
- Extensibility is not possible as it is platform dependent

MVC ARCHITECTURE

The MVC design pattern consists of three modules model, view and controller.

Model

- The model represents the state (data) and business logic of the application.
- For example-in case of a check box, the model contains a field which indicates whether the box is checked or unchecked.

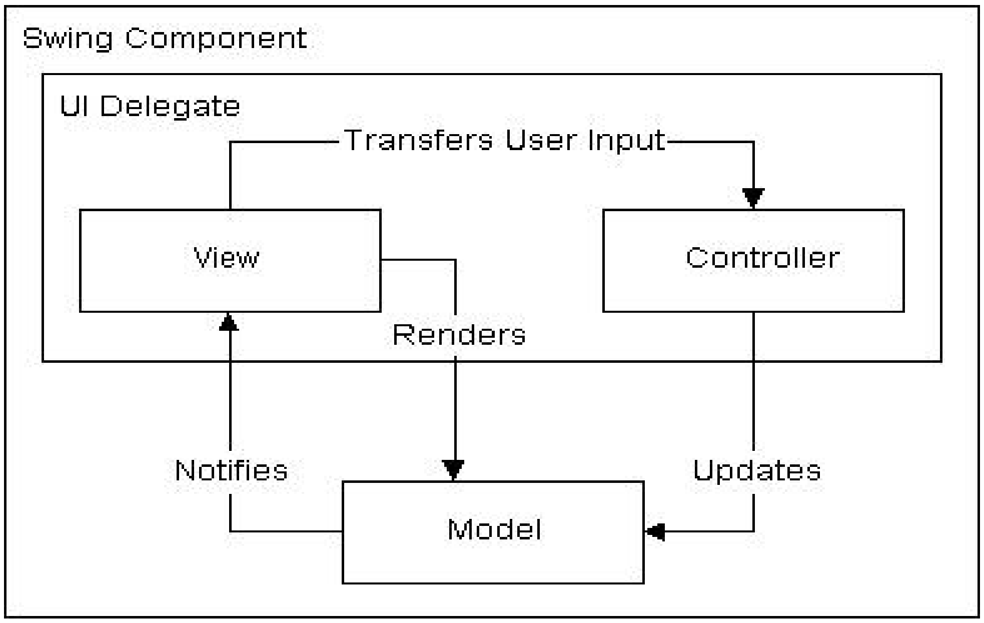
View

- The view module is responsible to display data i.e. it represents the presentation.
- The view determines how a component has displayed on the screen, including any aspects of view that are affected by the current state of the model.

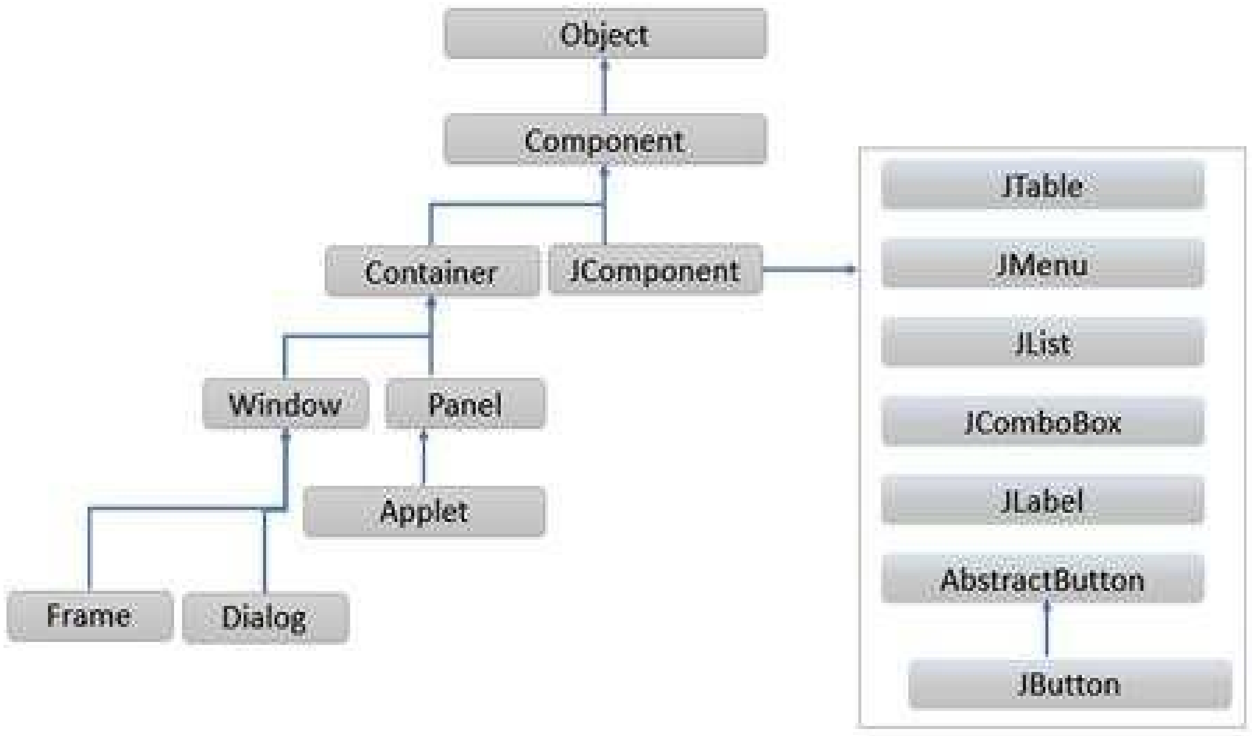
Controller

- The controller determines how the component will react to the user.
- The controller module acts as an interface between view and model.

- It intercepts all the requests i.e. receives input and commands to Model / View to change accordingly.



HIERARCHY OF JAVA SWING CLASSES



COMPONENT CLASS

- A *component* is an object having a graphical representation that can be displayed on the screen and that can interact with the user.
- Examples of components are the buttons, checkboxes, and scrollbars of a typical graphical user interface.

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

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

CONTAINER

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

WINDOW

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

JPANEL

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

JDIALOG

- The JDialog 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 Dialog class.
- Unlike JFrame, it doesn't have maximize and minimize buttons.

JFrame

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

There are two ways to create a frame:

1. By creating the object of Frame class (association)
2. By extending Frame class (inheritance)

1. By creating the object of Frame class (association)

```
import javax.swing.*;
public class FirstSwingExample
{
        public static void main(String[] args)
        {
            JFrame f=new JFrame();
            JButton b=new JButton("click");
            f.add(b);
            f.setSize(400,500);
            f.setVisible(true);
        }
 }
```

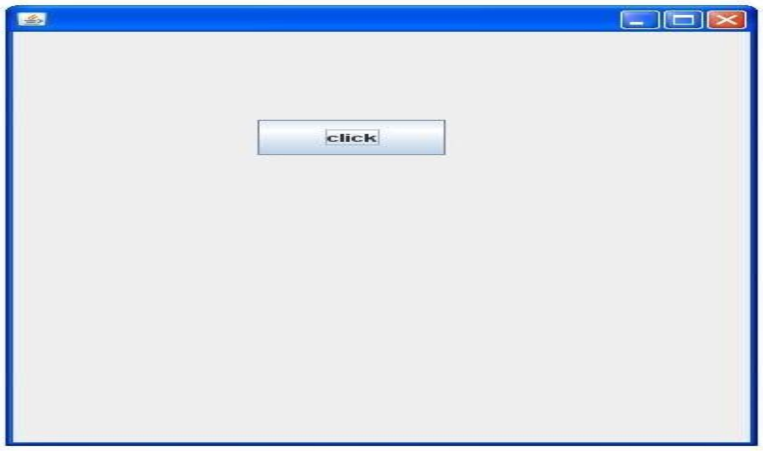
2. By extending Frame class (inheritance)

We can also inherit the JFrame class, so there is no need to create the instance of JFrame class explicitly.

EXAMPLE

```
import javax.swing.*;
public class Simple2 extends JFrame
{
            Simple2()
            {
                 JButton b=new JButton("click");
                 add(b);
                 setSize(400,500);
                 setVisible(true);
            }
            public static void main(String[] args)
            {
                  new Simple2();
            }
  }
```

Output



SWING COMPONENTS

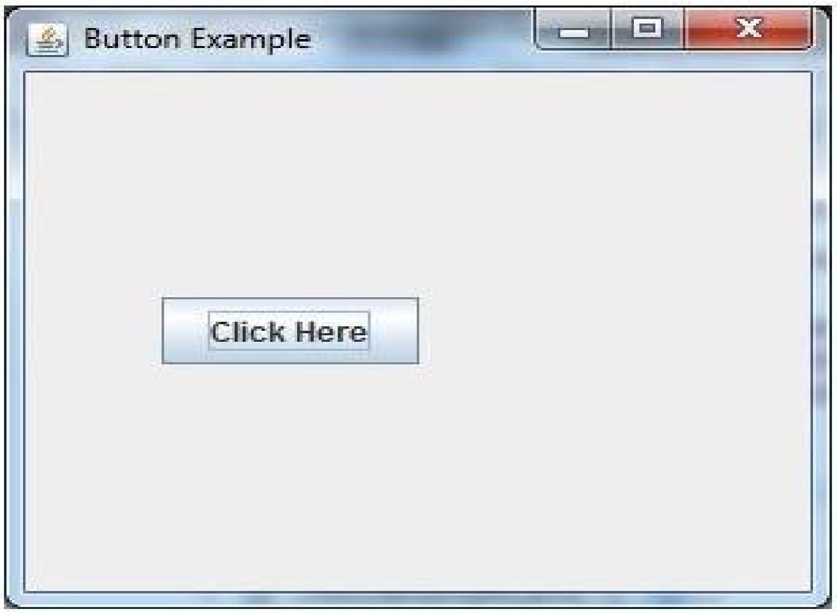
JButton

- 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.

Example

```
import javax.swing.*;
public class ButtonExample
{
        public static void main(String[] args)
        {
                JFrame f=new JFrame("Button Example");
                JButton b=new JButton("Click Here");
                f.add(b);
                f.setSize(400,400);
                f.setVisible(true);
        }
}
```

Output:



JLabel

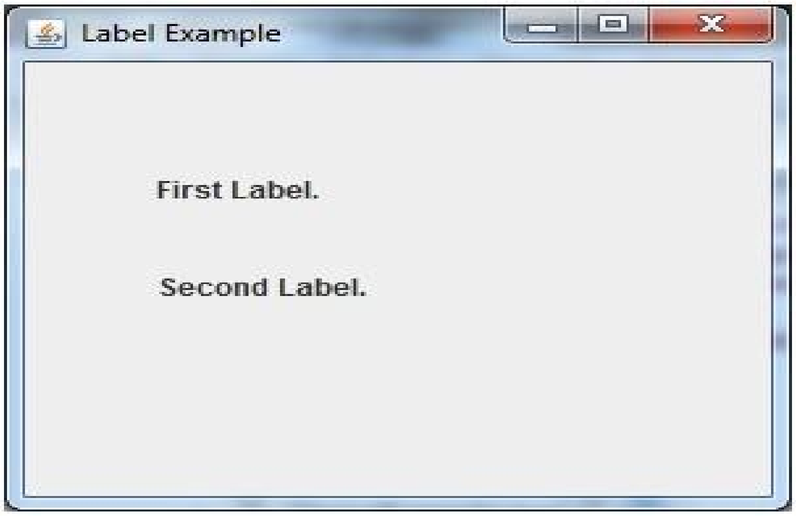
- The object of JLabel class is a component for placing text in a container.
- It is used to display a single line of read only text.
- The text can be changed by an application but a user cannot edit it directly.
- It inherits JComponent class.

Example

```
import javax.swing.*;
class LabelExample
{
    public static void main(String args[])
    {
        JFrame f= new JFrame("Label Example");
        JLabel  l1=new JLabel("First Label.");
        t1.setBounds(50,100, 200,30);
        JLabel l2=new JLabel("Second Label.");
        t1.setBounds(50,100, 200,30);
        f.add(l1);
        f.add(l2);
        f.setSize(300,300);
```

```
                    f.setLayout(null);
                    f.setVisible(true);
          }
     }
```

Output



JTextField

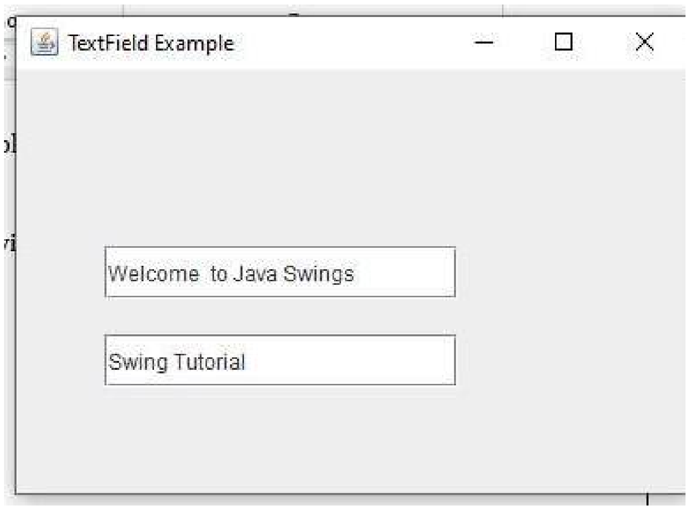
- The object of a JTextField class is a text component that allows the editing of a single line text.
- It inherits JTextComponent class.

Example

```
import javax.swing.*;
class TextFieldExample
{
     public static void main(String args[])
     {
          JFrame f= new JFrame("TextField Example");
          JTextField t1,t2;
          t1=new JTextField("Welcome  to Java Swings");
          t1.setBounds(50,100, 200,30);
          t2=new JTextField("Swing Tutorial");
          t2.setBounds(50,150, 200,30);
          f.add(t1);
          f.add(t2);
          f.setSize(400,400);
          f.setLayout(null);
          f.setVisible(true);
```

```
        }
}
```

Output:



Welcome to Java Swings

Swing Tutorial

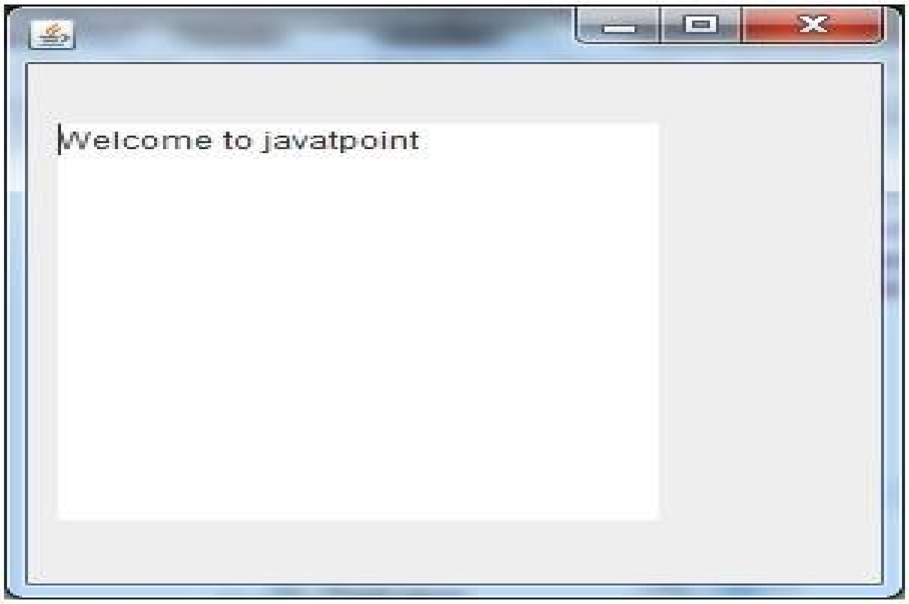
JTextArea

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

Example

```
import javax.swing.*;
public class TextAreaExample
{
        TextAreaExample()
        {
                JFrame f= new JFrame();
                JTextArea area=new JTextArea("Welcome to javatpoint");
                f.add(area);
                f.setSize(300,300);
                f.setVisible(true);
        }
      public static void main(String args[])
      {
                new TextAreaExample();
      }
}
```

Output



```
Welcome to javatpoint
```

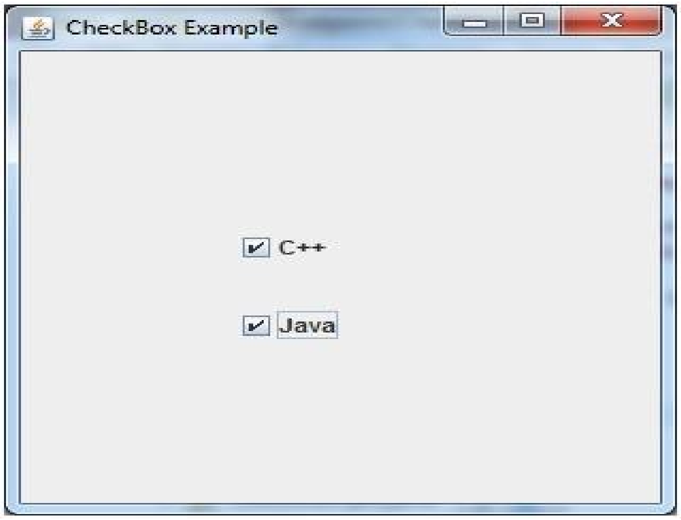
JCheckBox

- 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.

Example

```
import javax.swing.*;
public class CheckBoxExample
{
        CheckBoxExample()
        {
                JFrame f= new JFrame("CheckBox Example");
                JCheckBox checkBox1 = new JCheckBox("C++");
                checkBox1.setBounds(100,100, 50,50);
                JCheckBox checkBox2 = new JCheckBox("Java", true);
                checkBox2.setBounds(100,150, 50,50);
                f.add(checkBox1);
                f.add(checkBox2);
                f.setSize(400,400);
                f.setLayout(null);
                f.setVisible(true);
        }
        public static void main(String args[])
        {
                new CheckBoxExample();
        }
}
```

Output



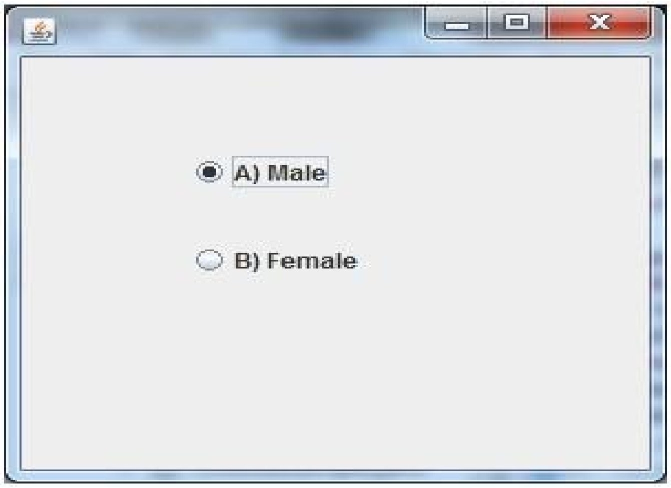
JRadioButton

- The JRadioButton class is used to create a radio button. It is used to choose one option from multiple options. It is widely used in exam systems or quiz.
- It should be added in ButtonGroup to select one radio button only.

Example

```
import javax.swing.*;
public class RadioButtonExample
{
    JFrame f;
    RadioButtonExample()
    {
        f=new JFrame();
        JRadioButton r1=new JRadioButton("A) Male");
        JRadioButton r2=new JRadioButton("B) Female");
        r1.setBounds(75,50,100,30);
        ButtonGroup bg=new ButtonGroup();
        bg.add(r1);
        bg.add(r2);
        f.add(r1);
        f.add(r2);
        f.setSize(300,300);
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
        new RadioButtonExample();
    }
}
```

Output



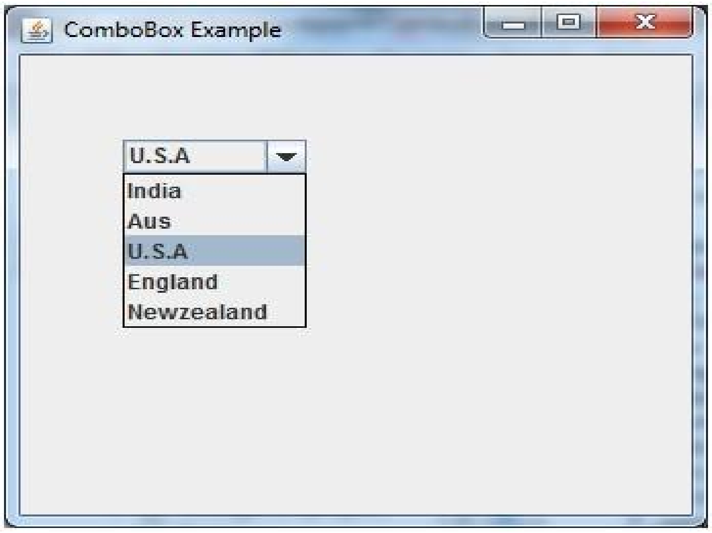
JComboBox

- 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.

Example

```
import javax.swing.*;
public class ComboBoxExample
{
    JFrame f;
    ComboBoxExample()
    {
        f=new JFrame("ComboBox Example");
        String country[]={"India","Aus","U.S.A","England","Newzealand"};
        JComboBox cb=new JComboBox(country);
        f.add(cb);
        f.setSize(400,500);
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
        new ComboBoxExample();
    }
}
```

Output



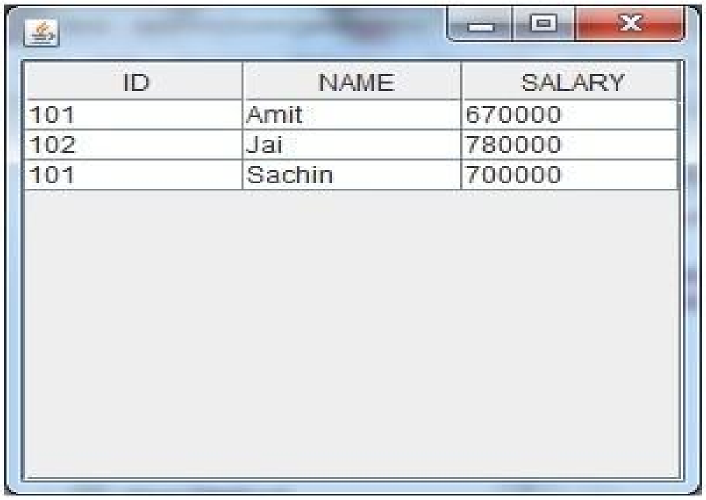
JTable

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

Example

```
import javax.swing.*;
public class TableExample
{
    JFrame f;
    TableExample()
    {
        f=new JFrame();
                                                                                String data[]
    []={ {"101","Amit","670000"}, {"102","Jai","780000"}, {"101","Sachin","700000"}};     String col
    umn[]={"ID","NAME","SALARY"};
        JTable jt=new JTable(data,column);
        JScrollPane sp=new JScrollPane(jt);
        f.add(sp);
        f.setSize(300,400);
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
        new TableExample();
    }
}
```

Output



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

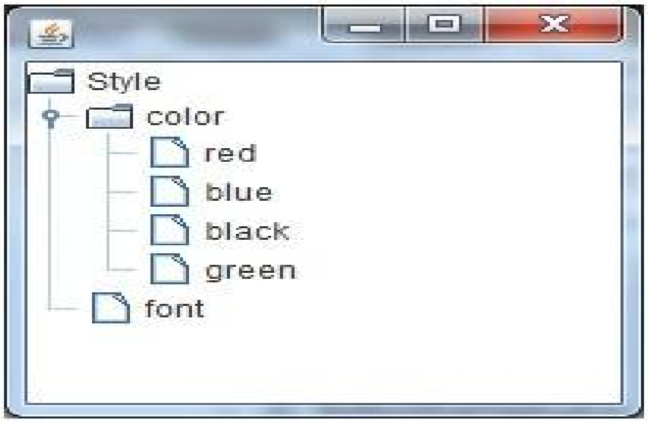
JTree

- 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.

Example

```
import javax.swing.*;
import javax.swing.tree.DefaultMutableTreeNode;
public class TreeExample
{
    JFrame f;
    TreeExample()
    {
        f=new JFrame();
        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);
        JTree jt=new JTree(style);
        f.add(jt);
        f.setSize(200,200);
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
      new TreeExample();
    }
}
```

Output:



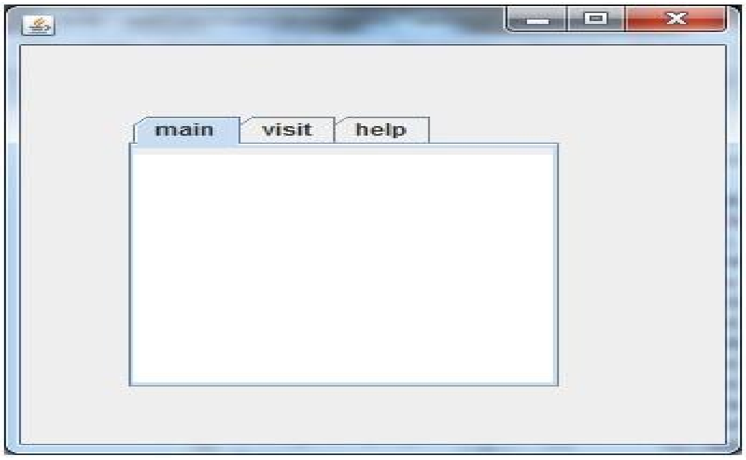
JTabbedPane

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.

Example

```
import javax.swing.*;
public class TabbedPaneExample
{
    JFrame f;
    TabbedPaneExample()
    {
        f=new JFrame();
        JTextArea ta=new JTextArea(200,200);
        JPanel p1=new JPanel();
        p1.add(ta);
        JPanel p2=new JPanel();
        JPanel p3=new JPanel();
        JTabbedPane tp=new JTabbedPane();
        tp.setBounds(50,50,200,200);
        tp.add("main",p1);
        tp.add("visit",p2);
        tp.add("help",p3);
        f.add(tp);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
      new TabbedPaneExample();
    }
}
```

Output:



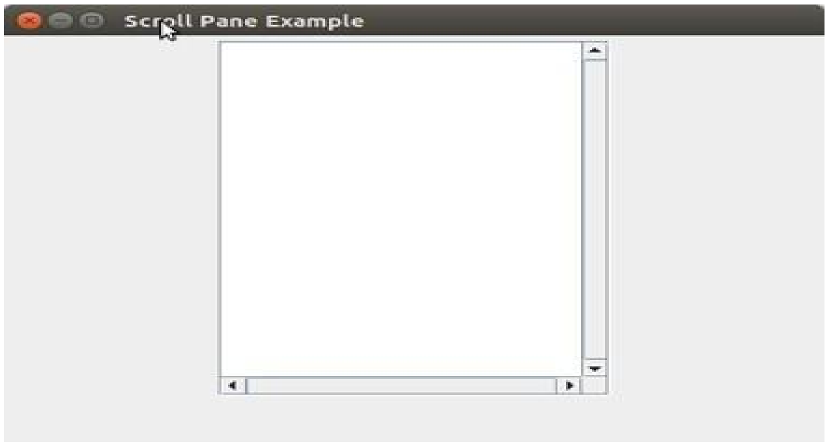
JScrollPane

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.

Example

```
import java.awt.FlowLayout;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JtextArea;
public class JScrollPaneExample
{
   private static final long serialVersionUID = 1L;
   private static void createAndShowGUI()
   {
        final JFrame frame = new JFrame("Scroll Pane Example");
        frame.setSize(500, 500);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().setLayout(new FlowLayout());
        JTextArea textArea = new JTextArea(20, 20);
        JScrollPane scrollableTextArea = new JScrollPane(textArea);
    scrollableTextArea.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
    scrollableTextArea.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
    frame.getContentPane().add(scrollableTextArea);
  }
  public static void main(String[] args)
  {
    javax.swing.SwingUtilities.invokeLater(new Runnable()
        {
          public void run()
          {
             createAndShowGUI();
          }
        });
     }
  }
```

Output:



6.MID Question paper

II-II CSM A&B
Subject: Object-Oriented Programming through Java
Faculty: B.Ranjith Kumar/A.Raju/A.Chiranjeevi
SET-1

1. Write a java program on multiple inheritance and explain different forms of inheritance?
2. Differentiate type conversion with type casting with an example each with program.
3. What is constructor? Write a java program on constructor?
4. Explain the concept of exception handling with program?
5. Write a java program on A)Control statements small note B)class path
6. a. Write a program on super keyword?Explain about it?
   b. Differentiate over loading with overriding ?Explain with program?

SET-2

1. Explain the concepts of oops? Write program on creation of object for class?
2. Write a java program on multiple inheritance ?What are the different forms of inheritance?
3. Discuss why to use final key word with a suitable program?
4. Write a program on paramterized constructor? Explain about it?
5. Explain the concept of exception handling? Write a program on unchecked exception.
6. Explain about A)member access rules B)this keyword with program

SET-3

1. Why do we need type casting .explain with an example with program
2. Write a program on abstract class and interface?
3. What are the different kinds of data types supported in java? Write a program on Primitive Data Type?
4. Explain about the java buzz words?
5. What is package? Write a program to create a package?
6. Write a program using try catch and finally blocks?

JAVA OBJECTIVES(II-CSM A&B/CSO)

UNIT-1

1) Which of the following is the correct way to declare a variable in Java?

   [A] int 1x = 10;   [B] int x = 10;

   [C] x = 10;   [D] none

2) What are the declare of array correct ways?

   [A] int[] arr = new int[5];

   [B]int arr[] = new int[10];

   [C]int arr[]= {1,2,3,4};

   [D]All

3) Consider the following object declaration statement: [   ]

   Scanner = new Scanner(System.in);

   What does System.in stand for in the above declaration?

   [A]Any file storing data

   [B] Refers to the standard input stream, which is typically the keyboard by default

   [C] Reference to scanner as an input device

   [D] It is a mouse as an input device

4) What is the correct signature of the main method in Java? [   ]

   [A]public void main(String args[])   [B]public static void main(String args[])

   [C]void main(String args[])   [D]public static void main()

5) What will be the output of the following Java program?

```
class java {
    public static void main(String args[]) {
        int a=10; {
         int a=20;
```

```
            System.out.print(a); }}}
```

[A] Compilation error [B] Runtime error [c] 10 [D] 20

6) What will be the output of the following Java program?

```
class Animal {
 public static void main(String args[]) {
 String obj = "My Bits College";
 System.out.println(obj.charAt(4));
 } }
```

[A] i [B] B [C] t [D] l

[A] javac [B] java [C] javad [D] .javadoc

7) Guess the Name of Logic?

```
  class abc {
 int m(int n) {
 if (n==1) return 1;
 else
  int res=n*m(n-1);
  return res; } }
```

[A] Control Statements [B]Recursion [C] Conditional Statements [D] None

8) Which control statement is used to execute a block of code repeatedly until a certain condition is met?

[A] if B) while C) switch D) break

9)The extension name of a Java source code file is? [ ]

[A] .java [B] .obj [C] .class [D] .exe

10)Which member can never be accessed by inherited classes? [ ]

[A] Private member function b) Public member function
c) Protected member function d) All can be accessed

1) The output of the Java compiler is known as ____________

2) ____________variables cannot be changed after initialization.

3) ____________keyword is used to reference to the current object.
4) A method declared with ____________ modifier cannot be overridden.
5) Every object has a ________,__________and __________
6) Languages like Simula, Smalltalk, C++ and Java adopt ___________ approach in programming
7) ________________converting a smaller type to a larger type size

8) ____________ is used to access members of the base class.

9) In java A class can extend at most ________________number of classes at a time.

10) Method same name different signature is called as _____________

UNIT II

1. What is the output of the following code if the package name is com.example? [ ]

```
package com.example;

public class Test {

 public static void main(String[] args) {

    System.out.println("Hello from com.example package");

 }

}
```

[A] Hello from com.example package [B] Error: Package not found
[C] com.example.Test [D] Hello from Test class

2. ………………..Array the elements are stored in sequential manner [ ]
[A] Two-dimensional array [B] Single-dimensional array
[C] Package [D] Interface

3. Which of the following can be declared as final in Java?

A) Methods only B) Variables only

C) Methods, variables, and classes D) None of the above

4 .Which of the following packages is imported by default in every Java program?

A) java.io.*; B) java.lang.*; C) java.util.*; D) None of the above

5. Write the fill in the blocks?

```
class Bits {
      int x;
    String name;
     Bits(int x,String rno) {
     ____.x=x;
     ____.name=name; } }
class abc {
     public static void main(String args[]){
     Bits obj = new Bits(2,"Anu");
     System.out.println(obj.x + " " + obj.name);
} }
```

a) super,super b) this,super c) super,this d) this,this

6. …………………….. package in java contents set of classes for implementing graphical user interface, which includes classes for windows, buttons, lists, menus and so on. [ ]

A) java.util
B) java.awt
C) java.net
D) java.lang

7. State whether the following statements are True or False. [ ]

i) When present, package must be the first statement is package name in the file.
ii) When we implement an interface method, it should be declared as public.
A) True, False B) False, True C) True, True D) False, False

8. What will be LINE 1 the output of the following Java program?

```
class A {
    void dis() { System.out.println("Parent"); } }
 class B extends A {
    void dis() { System.out.println("Child"); } }
 class Main {
 public static void main(String args[]) {
     A obj=new B(); //LINE 1
     B obj=new A(); //LINE 2
     obj.dis();  } }
```

[A]Parent   [B] Child   [C] Compilation Error   [D]None

9. A package is a collection of   [   ]

[A] Classes
[B] interfaces
[C] editing tools
[D] classes and interfaces

10. A class that contains normal methods and abstract methods is called as   [   ]

A) Interface

B) Abstract Class

C) Class

D) Data Type

11) ____________keyword is used when we used class with interface

12) The Date class includes within ………………….. Package.

13) A class can be declared as …………………….. if you do not want the class to be sub-classed.

14) The ……………………... keyword is used to derive a class from a super-class.

15) What is an Interface?_________________

16) _______________ keyword is used by a class to use an interface defined previously.

17) Arrays in Java ___________________

18) ________________is the default value of Boolean data type in java.

19) ________________ package contain classes and interfaces used for input & output operations of a program

20) Pure abstract class is known as_____________

UNIT III

1. What will be the output of the following Java program?

```
class exception_handling
  {        public static void main(String args[])
     {
        try  {
           int a, b;
           b = 0;
           a = 5 / b;
           System.out.print("A");
        }
        catch(ArithmeticException e)
        {        System.out.print("B");        }
        finally{   System.out.print("C");   }        }
  }
```

[A]A   [B]B   [C]AC   [D] BC

3. Exception generated in try block is caught in........... block.   **[   ]**

[A]Catch [B] throw [C] throws [D] finally

4. Which of the following blocks execute compulsorily whether exception is caught or not. [ ]

[A] Finally [B] throw [C]throws [D] catch

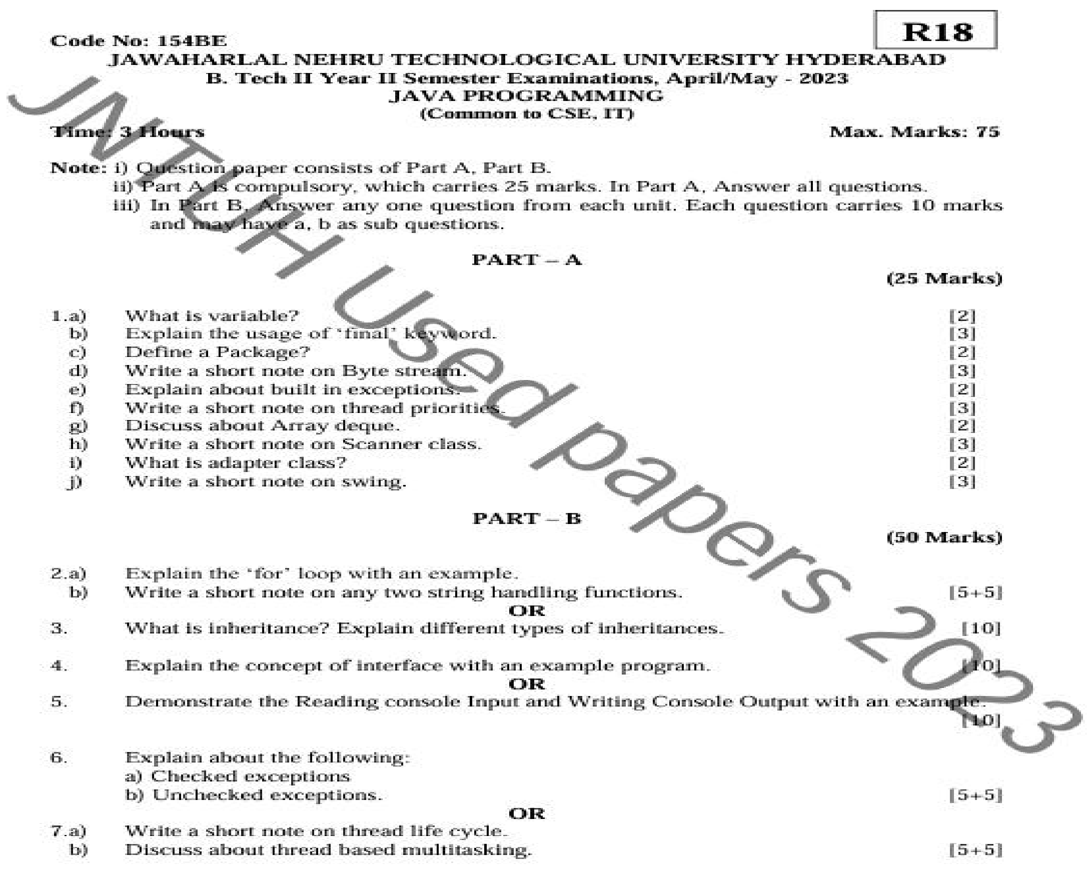
5. ______ is a superclass of all exception classes. [ ]

[A] Exception [B] Throwable

[C] RuntimeException [D] IOException

1. checked exception is occur at_______
2. When an array element is accessed beyond the array size, ____ exception is thrown
3. _______________ method is used to print the description of the exception?
4. Final classes are created so the methods implemented by that class cannot be _________
5. Exceptions can be handled using _________

7.Previous Year Papers



Code No: 154BE **R18**

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD
B. Tech II Year II Semester Examinations, April/May - 2023
JAVA PROGRAMMING
(Common to CSE, IT)

Time: 3 Hours **Max. Marks: 75**

Note: i) Question paper consists of Part A, Part B.
ii) Part A is compulsory, which carries 25 marks. In Part A, Answer all questions.
iii) In Part B, Answer any one question from each unit. Each question carries 10 marks and may have a, b as sub questions.

PART – A

(25 Marks)

1.a)	What is variable?	[2]
b)	Explain the usage of 'final' keyword.	[3]
c)	Define a Package?	[2]
d)	Write a short note on Byte stream.	[3]
e)	Explain about built in exceptions.	[2]
f)	Write a short note on thread priorities.	[3]
g)	Discuss about Array deque.	[2]
h)	Write a short note on Scanner class.	[3]
i)	What is adapter class?	[2]
j)	Write a short note on swing.	[3]

PART – B

(50 Marks)

2.a) Explain the 'for' loop with an example.
b) Write a short note on any two string handling functions. [5+5]

OR

3. What is inheritance? Explain different types of inheritances. [10]

4. Explain the concept of interface with an example program. [10]

OR

5. Demonstrate the Reading console Input and Writing Console Output with an example. [10]

6. Explain about the following:
a) Checked exceptions
b) Unchecked exceptions. [5+5]

OR

7.a) Write a short note on thread life cycle.
b) Discuss about thread based multitasking. [5+5]

8. Briefly explain about the following:
a) Linked List
b) Tree set. [5+5]

OR

9. Write a short note on:
a) Priority Queue
b) Hashtable. [5+5]

10. Briefly explain about the following:
a) Card Layout
b) JScroll Pane. [5+5]

OR

11.a) Explain any two swing controls.
b) Write a Java program to demonstrate the handling Mouse events. [5+5]

R18

Code No: 154BE

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD
B. Tech II Year II Semester Examinations, August/September - 2021
JAVA PROGRAMMING
(Common to CSE, IT)

Time: 3 Hours **Max. Marks: 75**

Answer any five questions
All questions carry equal marks
- - -

1.a) Analyze the characteristics of object oriented programming concepts?
b) With suitable program segments examine the usage of 'super' keyword. [8+7]

2.a) Does Java support multi way selection statement? Justify your answer.
b) Generate different forms of inheritance with suitable program segments and real world example classes. [7+8]

3.a) Demonstrate about Reading console Input and Writing Console Output.
b) Explain nested interface with example. [7+8]

4.a) What is java package? What is CLASSPATH? Show how to create and access a java package with an example.
b) Create an interface with at least one method and implement that interface. [7+8]

5.a) What is meant by re-throwing exception? Demonstrate with a suitable scenario for this.
b) Write a program that creates a thread that forces pre-emptive scheduling for lower priority threads. [7+8]

6.a) Summarize the differences between thread-based multitasking and process-based multitasking.
b) Write a program to illustrate user defined exception that checks the internal and external marks if the internal marks are greater than 40 it raise the exception "internal marks are exceed", if the external marks are greater than 60 exception is raised and display the message the "external marks are exceed." [7+8]

7.a) Develop a program to read a file content and extract words using String Tokenized class. Display the file if it contains the user query term/search key.
b) Judge the purpose of Stack class. [8+7]

8.a) Design a user interface to collect data from the student for admission application using swing components.
b) What is an adapter class? Demonstrate its role in event handling. [8+7]

8.Unit Wise Important Questions

UNIT-1

1. Explain the concept of classes, objects and methods in OOP?
2. List and explain the Java buzzwords in detail?
3. Define constructor? Explain different types with suitable example program?
4. Define data type? Explain various data types in Java?
5. Explain about control statements and operators in Java?
6. Explain about type conversion and casting with example?
7. Discuss about inner classes in Java with suitable example?
8. Discuss about following with suitable examples?
   a)this b)super c)final

UNIT-2

1. What is inheritance? Explain different types of inheritances?
2. Explain about Exploring java.io?
3. Explain how Packages are created and accessed?
4. Demonstrate with an example method overriding?
5. Explain the concept of implement the interface with an example program?
6. What is abstract class ?Explain with suitable program

UNIT-3

1. What is an exception? Explain the exception in handling java?
2. Summarize the differences between thread-based multitasking and process-based multitasking?
3. Develop a program that includes a try block and a catch clause which processes the arithmetic exception generated by division-by-zero error.
4. Discuss about following with suitable examples? a)Enumerations b)autoboxin
5. Write a short note on thread life cycle
6. Write about inter thread communication
7. What is mean by generic ? Explain with suitable program?
   Write a short note on any two string handling functions?

UNIT-4

1. What is an adapter class? Demonstrate its role in event handling?
2. With the help of a neat diagram, explain the AWT class architecture?
3. Explain about Delegation event model?
4. Discuss the following user interface components a)canvas b)scrollbars, text components a)check box b)checkbox groups c)choices
5. Write short note on scroll pane, dialogs, menu bar?
6. Discuss about Layout manager with suitable examples?

UNIT-5

1.What is Applet ?Explain about Applet lifecycle with diagram?
2.Write the differences between applet program and application program.
3.Write the limitations of AWT?
With the help of a neat diagram, explain the Swing architecture.
4.Explain MVC architecture with suitable diagram?
5.Explain the swing components JApplet, JFrame and Jcomponent with suitable program?