

01
TOPIC **OOP principles, Java Buzzwords**

02
TOPIC **Implementing Java program, JVM**

03
TOPIC **Data Types, Variables, Type conversions and Casting**

04
TOPIC **Operators, Control statements, Arrays**

05
TOPIC **Class, Objects, Methods and Constructors**

06
TOPIC **this keyword, static keyword, Overloading Methods and Constructors**

07
TOPIC **Argument passing, Exploring String class, StringBuffer class**

08
TOPIC **String Tokenizer class and Date class.**

Course Outcomes (COs)



- A8601.1 Make **use of various constructs** to **write** a console **application**.
- A8601.2 **Use principles of OOP** to **develop real time applications**.
- A8601.3 **Examine** the applications for **Exception Handling** and **Multithreading**.
- A8601.4 **Implement Collection Framework** **to organize data** efficiently .
- A8601.5 Build **GUI applications** using **AWT** and **Swings**.

What is Java?

- Java is a **programming language** and a **platform**.
- Java is a **high level, robust, secured** and **object-oriented programming language**.

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

- Where it is used?
 - ✓ **Desktop Applications** such as acrobat reader, media player, antivirus etc.
 - ✓ **Web Applications** such as irctc.co.in, cleartrip.com etc.
 - ✓ **Enterprise Applications** such as banking,e-commerce applications.
 - ✓ **Mobile Applications (J2ME)**
 - ✓ **Embedded System**
 - ✓ **Smart Card**
 - ✓ **Robotics**
 - ✓ **Games etc.**

Evolution of Java

- Currently, **Java is used in internet programming, mobile devices, games, e-business solutions** etc.
 - i. **James Gosling**, Mike Sheridan, and Patrick Naughton **initiated** the **Java language** project in **June 1991**. The **small team** of **sun engineers** called **Green Team**.
 - ii. **Originally designed** for **small, embedded systems** in electronic appliances **like set-top boxes**.
 - iii. **Firstly, it was called "Greentalk"** by James Gosling and file extension was **.gt**.
 - iv. **After that**, it was **called Oak** and was developed as a part of the Green project.

Why Oak?

Oak is a symbol of strength and chosen as a **national tree of many countries like U.S.A., France, Germany, Romania** etc.

In 1995, Oak was renamed as "Java" because it was **already a trademark by Oak Technologies**.

Why they choosed java name for java language?



- vii. The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA" etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell and fun to say. According to James Gosling "Java was one of the top choices along with Silk". Since java was so unique, most of the team members preferred java.
- viii. Java is an island of Indonesia where first coffee was produced (called java coffee).
- ix. Notice that Java is just a name not an acronym.
- x. Originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.
- xi. In 1995, Time magazine called Java one of the Ten Best Products of 1995.
- xii. JDK 1.0 released in(January 23, 1996)

java versions

- ✓ There are **many java versions that have been released. Current stable release of Java is Java SE 8.**

1. JDK Alpha and Beta (1995)

2. JDK 1.0 (23rd Jan, 1996)

3. JDK 1.1 (19th Feb, 1997)

4. J2SE 1.2 (8th Dec, 1998)

5. J2SE 1.3 (8th May, 2000)

6. J2SE 1.4 (6th Feb, 2002)

7. J2SE 5.0 (30th Sep, 2004)

8. Java SE 6 (11th Dec, 2006)

9. Java SE 7 (28th July, 2011)

10. Java SE 8 (18th March, 2014)

11. Java SE 9 (September, 21st 2017)

12. Java SE 10(March, 20th 2018)

13. Java SE 11 (18th March, 2014)

14. Java SE 12(March, 19th 2019)

15. Java SE 13(September, 17th 2019)

16. Java SE 14(March, 17th 2020)

17. Java SE 15(Expected in September 2020)

OOPs Concepts / OOPs Principles

- Object means a **real word entity** such as **pen, chair, table etc.**
- **Object Oriented Programming** is a methodology or paradigm **to design a program using classes and objects.** It **simplifies** the **software development** and **maintenance** by **providing some concepts:**
 - i. **Object**
 - ii. **Class**
 - iii. **Inheritance - OOP Principle -2**
 - iv. **Polymorphism - OOP Principle -3**
 - v. **Abstraction**
 - vi. **Encapsulation - - OOP Principle -1**
- i. **Object:** Any entity that **has state(properties)** and **behavior(perform some task)** is known as an object.
It is a real world entity.
Example: **chair, pen, table, keyboard, bike etc.** It can be **physical and logical.**
- ii. **Class:** Collection of objects is called **class.** It is a **logical entity.**

OOPs Concepts / OOPs Principles



iii. Inheritance :

- ✓ When **one object acquires all the properties** and **behaviors of parent object** i.e. known as **inheritance**.
It **provides code reusability**. It is **used** to achieve **runtime polymorphism**.
- ✓ **Property transfer** from **Grand Parent** to **Parent** – to **children**.

iv. Polymorphism:

- ✓ When **one task is performed** by **different ways** i.e. known as **polymorphism**.
- ✓ **one in multiple forms** is known as **polymorphism**.

Example: When we are in **class – student**,

When we are in **Market – Customer**,

When we are in **Home – Son / Daughter**.

- ✓ In java, we use **method overloading** and **method overriding** to achieve **polymorphism**.

OOPs Concepts / OOPs Principles

v. Abstraction:

- ✓ Hiding internal details and showing functionality is known as abstraction.

Example: phone call, we don't know the internal processing.

- ✓ In java, we use abstract class and interface to achieve abstraction.
- ✓ Exposing only the required essentials characteristics & behavior with respect to the context.
- ✓ Shows important things to the user and hide the internals details.

Example: ATM , BIKE

vi. Encapsulation:

- ✓ Binding (or wrapping) code and data together into a single unit is known as encapsulation.

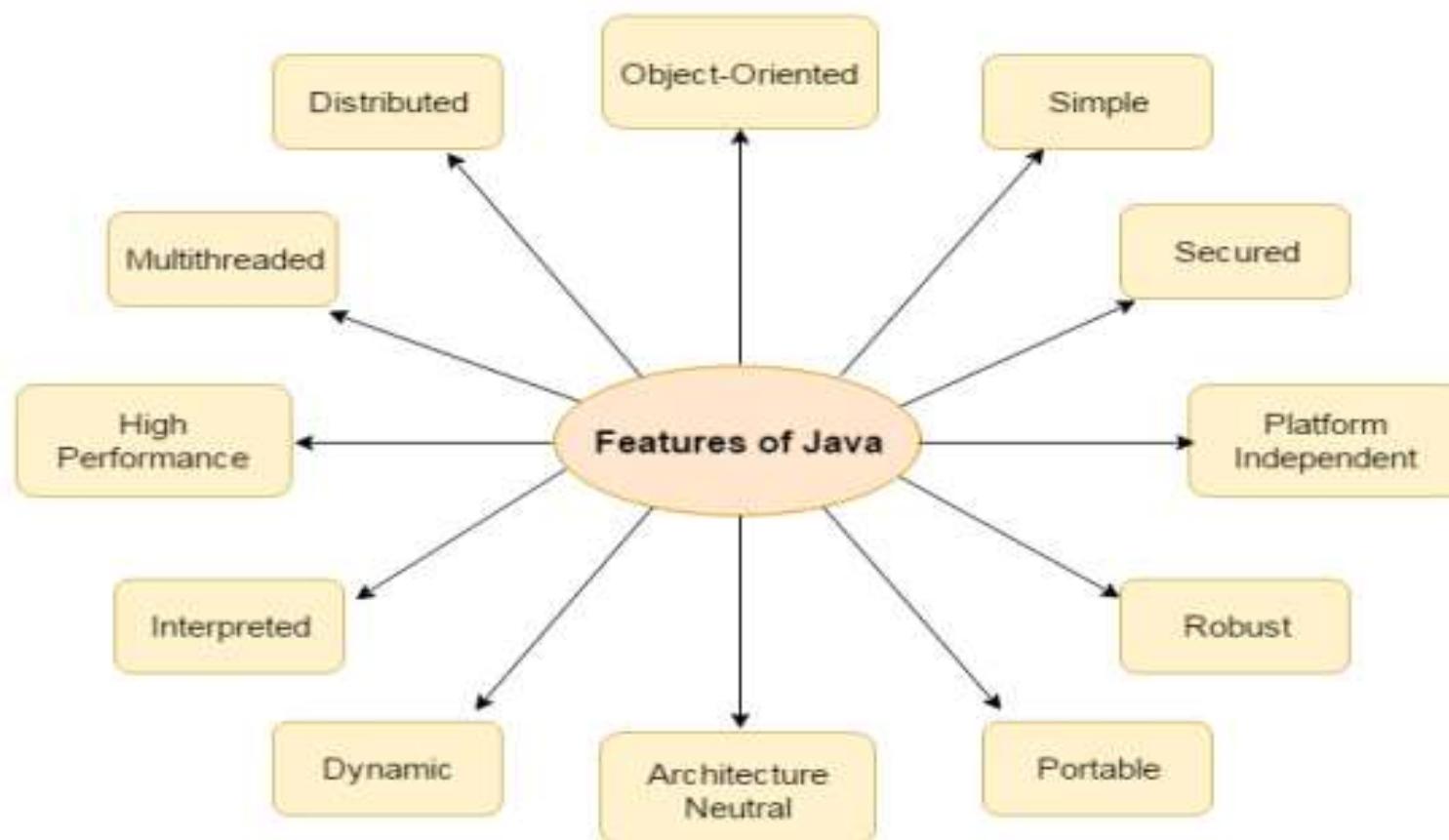
Example: Capsule, it is wrapped with different medicines,

Power Steering Car (Internally lot of Components tightly Coupled together)

- ✓ A java class is the example of encapsulation.
- ✓ Java bean is the fully encapsulated class because all the data members are private here.

Features of Java / Java Buzz Words

- There are many features of Java. They are also known as Java buzzwords. The Java Features given below are simple and easy to understand.



Features of Java / Java Buzz Words

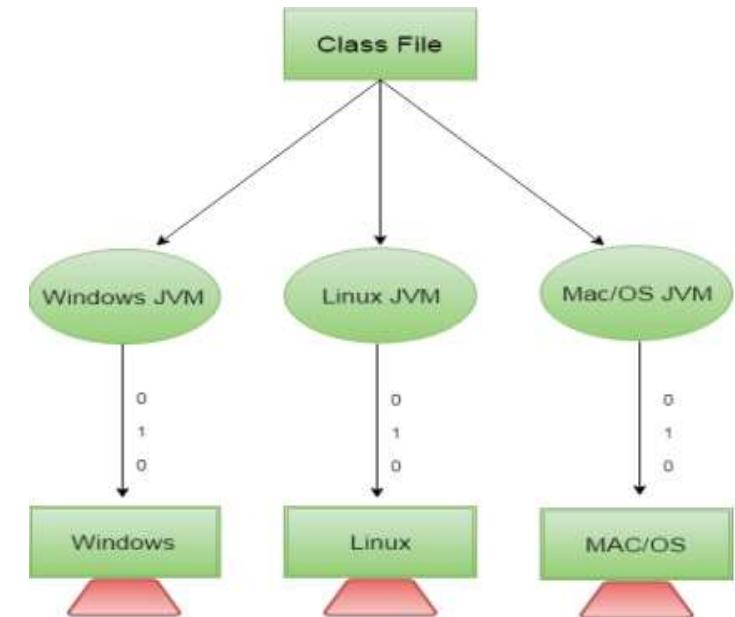
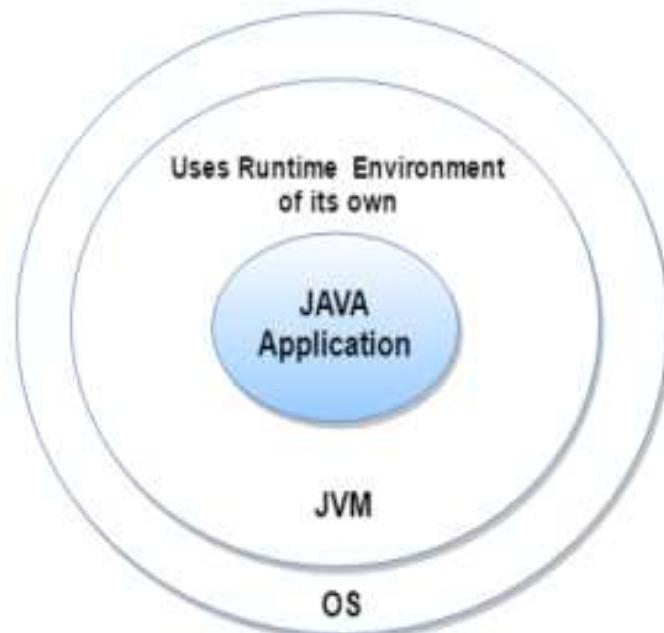
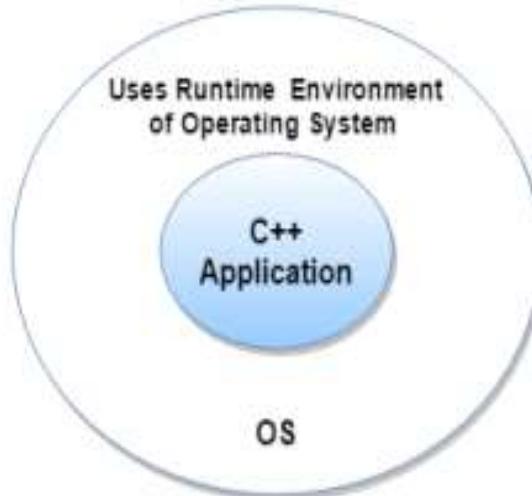
- **Object Oriented:** In Java, **everything is an Object**. Java can be **easily extended** since it is **based on the Object model**.
- **Simple:** Java is designed to be **easy to learn**. If you **understand the basic concept of OOP** Java would be **easy to master**.
- **Secure:** With Java's **secure feature** it enables to develop virus-free applications. **Authentication** techniques are based on **public-key encryption**.
- **Platform independent:** Unlike many other programming languages including **C and C++**, when **Java** is compiled, it is **not compiled into platform specific machine**, rather into platform independent byte code. **Robust:** Java makes **an effort to eliminate error prone situations** by emphasizing **mainly on compile time error checking** and **runtime checking**.

Features of Java / Java Buzz Words

- **Portable:** Being architectural-neutral and having no implementation dependent . We may carry the java bytecode to any platform.
- **Architectural-neutral:** Java compiler generates an architecture-neutral object file format which makes the compiled code to be executable on many processors, with the presence of Java runtime system.
- **Dynamic:** Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of runtime information that can be used to verify and resolve accesses to objects on run-time.
- **Interpreted:** Java byte code is translated to native machine instructions and is not stored anywhere.
- **High Performance:** With the use of Just-In-Time compilers, Java enables high performance.

Features of Java / Java Buzz Words

- **Multithreaded:** With Java's multithreaded feature it is **possible to write programs that can do many tasks simultaneously.** This feature allows developers to construct smoothly running interactive applications.
- **Distributed:** We can **create distributed applications in java. RMI and EJB are used for creating distributed applications.** We may **access files by calling the methods from any machine on the internet.**



Basic Structure of Java Program

Documentation Section

Package Statement

Import Statement

Interface Statement

Class Definition

Main Method Class

```
{  
    //Main method defintion  
}
```

save this file as Simple.java

To compile: javac Simple.java

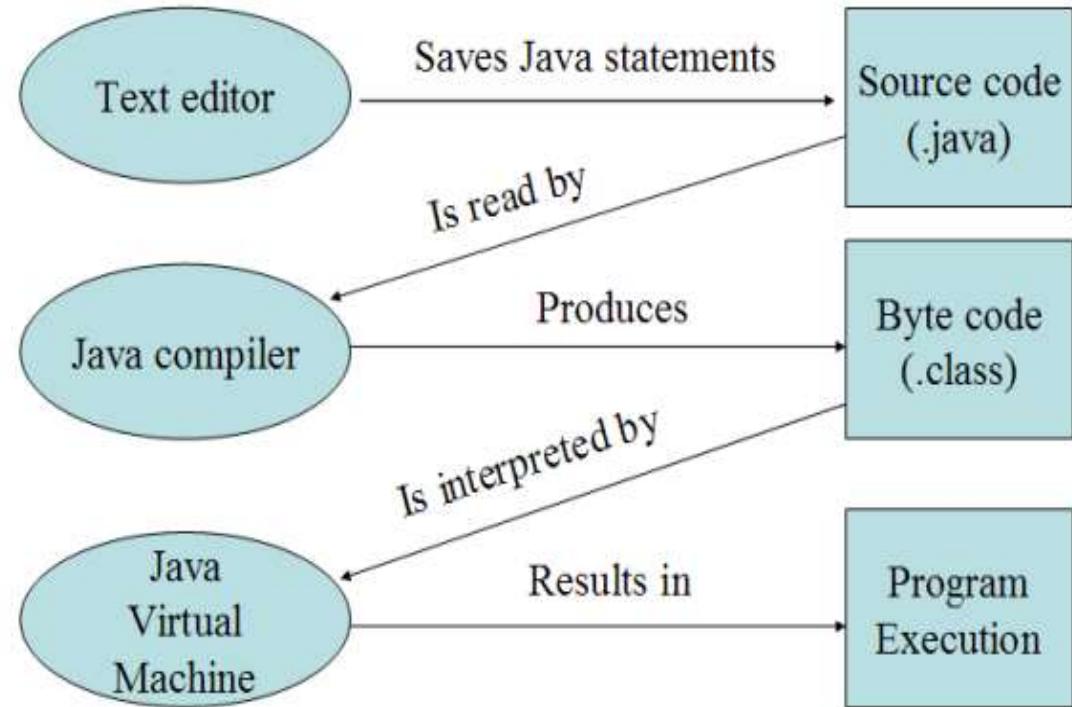
To execute: java Simple

```
public class Simple  
{  
    public static void main( String args[ ] )  
    {  
        System.out.println("Welcome To the World of Java");  
    }  
}
```

Basic Structure of Java Program

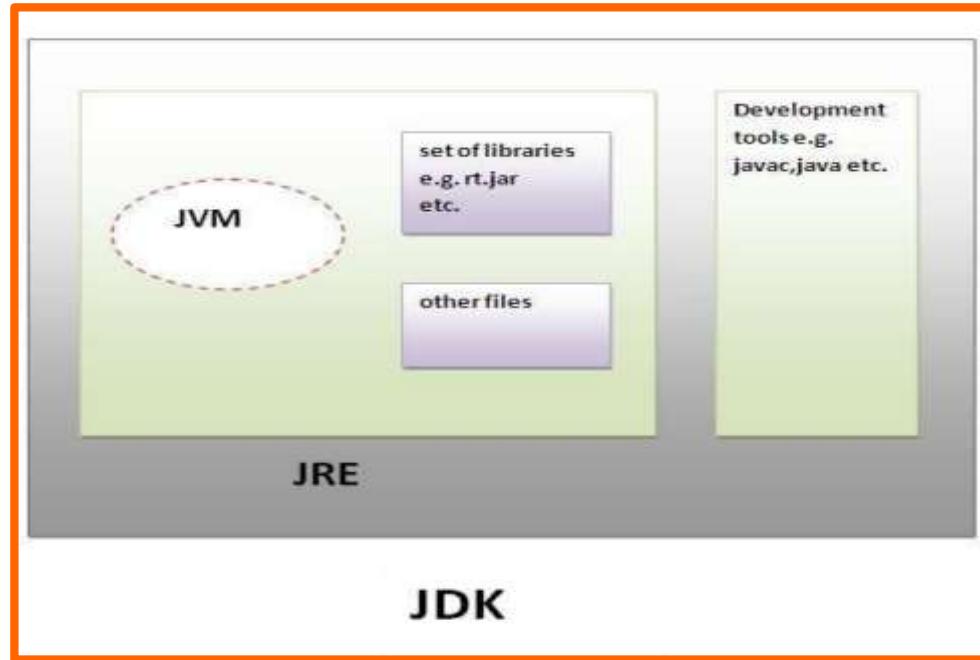
- i. **class keyword** is used to declare a class in java.
- ii. **public keyword** is an access modifier which represents visibility, it means it is visible to all.
- iii. **static** is a keyword, if we declare any method as static, it is known as static method. The core advantage of static method is that there is no need to create object to invoke or call the static method.
- iv. **void** is the return type of the method, it means it doesn't return any value.
- v. The main method is executed by the JVM, so it doesn't require to create object to invoke the main method. It represents startup of the program.
- vi. **String[] args** is used for command line argument.
- vii. **System.out.println()** is used print statement.
 - **System** is a class Which is present in **java.lang package**
 - **Out** is a static final field (variable) of Printstream class
 - **Println():** method of Printstream Class

Java program execution



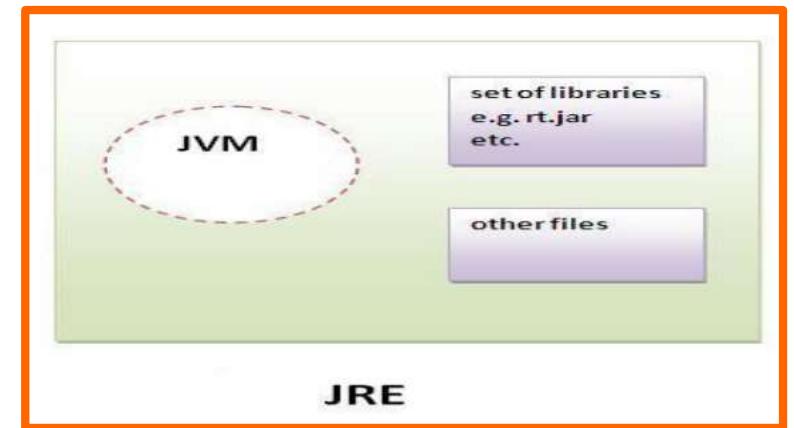
JDK (Java Development Kit)

- JDK is an acronym for It **physically exists**.
- It contains **JRE + development tools**.

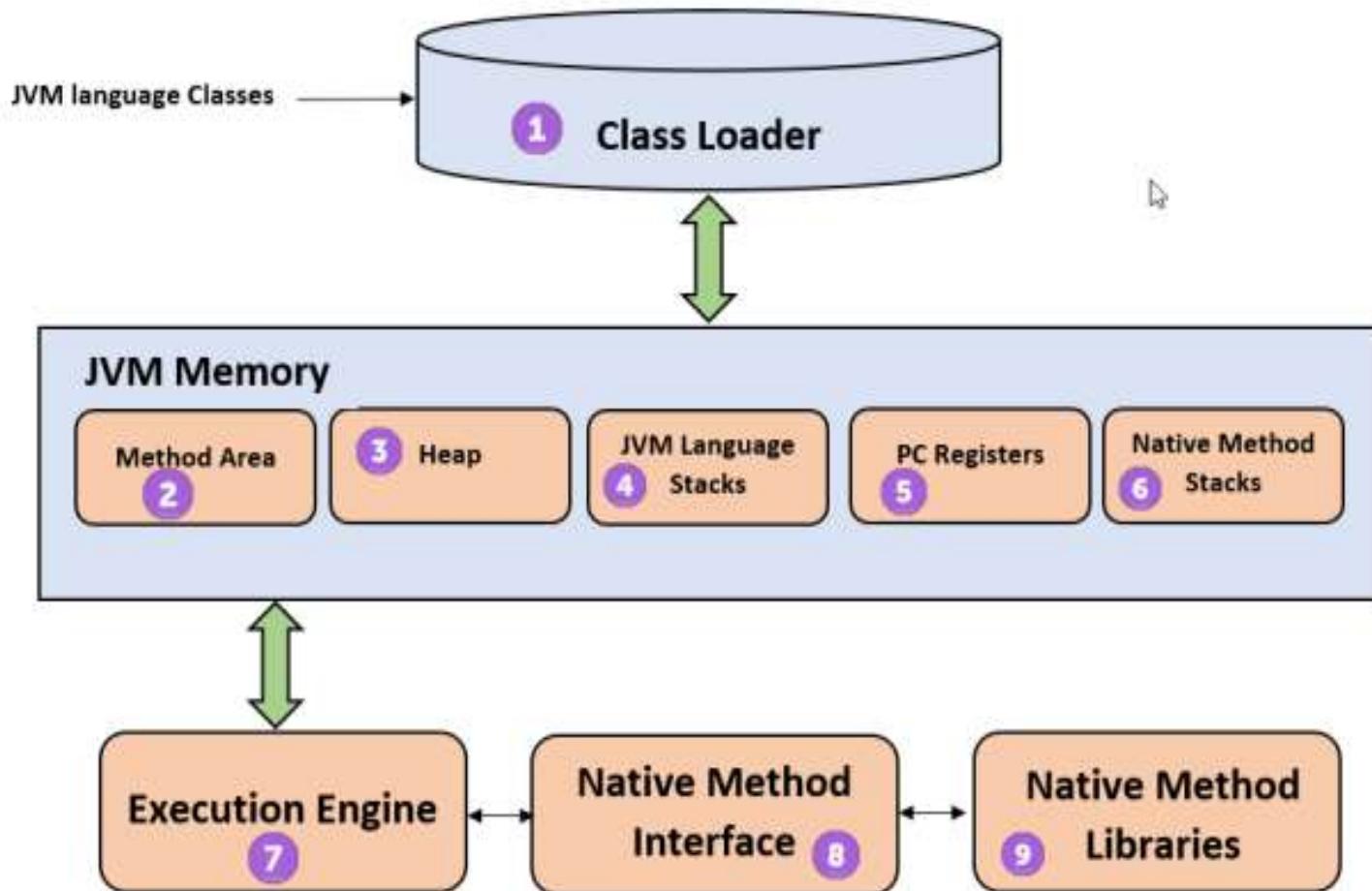


JRE (Java Runtime Environment)

- It is used to provide **runtime environment**.
- It is the implementation of **JVM**.
- It contains **set of libraries + other files that JVM uses at runtime**.
- **Implementation of JVMs** is also actively **released by other companies** besides Sun Micro Systems.



JVM (Java Virtual Machine)



JVM (Java Virtual Machine)

- JVM (Java Virtual Machine) **provides runtime environment** in which **java byte code** can be **executed**.
- The JVM performs following operation:
 - **Loads code**
 - **Verifies code**
 - **Executes code**
 - **Provides runtime environment**

1) Class loader:

- ✓ It is a **subsystem of JVM** that is **used to load class files**.
- ✓ It **loads, links** and **initializes** the **class file** when it refers to a class for the **first time at runtime**.

2) Method Area:

- ✓ It is the **class-level data** will be **stored here**, including static **variables**.
- ✓ There is **only one method area per JVM**, and it is a **shared resource**.

JVM (Java Virtual Machine)

3) Heap :

- ✓ It is the **runtime data area** in which **objects are allocated**.
- ✓ **All the Objects** and their **corresponding instance variables** and **arrays will be stored here**.

4) Stack :

- ✓ For **every thread**, a **separate runtime stack** will be **created**.
- ✓ For **every method call**, one entry will be made in the **stack memory** which is called **Stack Frame**.
- ✓ All **local variables** will **be created in the stack memory**.

5) Program Counter Register :

- ✓ **Each thread will have separate PC Registers**, to hold the **address of current executing instruction** once the instruction is executed the **PC register will be updated with the next instruction**.

6) Native Method Stack :

- ✓ It contains **all the native methods used** in the **application**.

JVM (Java Virtual Machine)



7) Execution Engine :

- ✓ The Execution Engine **reads the bytecode** and **executes it piece by piece**.
- ✓ **It contains three components:**

7.1 Interpreter:

- ✓ **Read byte code** stream **then execute the instructions**.
- ✓ The **interpreter** **interprets the bytecode faster** but **executes slowly**.
- ✓ The **disadvantage** of the **interpreter** is that when **one method** is **called multiple times**, every time a **new interpretation is required**.

7.2 Just-In-Time (JIT) compiler:

- ✓ The JIT Compiler **neutralizes** the **disadvantage** of the **interpreter**.
- ✓ The **Execution Engine** will be using the **help of the interpreter** in **converting byte code**, but **when it finds repeated code it uses the JIT compiler**, which compiles the entire **bytecode** and changes it to **native code**.
- ✓ The JIT Compiler compiles bytecode to machine code at runtime and **improves the performance** of Java applications.

JVM (Java Virtual Machine)



7.3 Garbage Collector:

- ✓ Automatic freeing of Heap Memory.
- ✓ Collects and removes unreferenced objects.
- ✓ Garbage Collection can be triggered by calling System.gc().

8) Java Native Interface (JNI) :

- ✓ JNI will be interacting with the Native Method Libraries and provides the Native Libraries required for the Execution Engine.

Example:

- ✓ If we are running the Java application on Windows, then the native method interface will connect the Windows libraries (native method libraries) for executing Windows methods (native methods).

9) Native Method Libraries:

- ✓ This is a collection of the Native Libraries, which is required for the Execution Engine .

Java Byte Code (Java Magic)

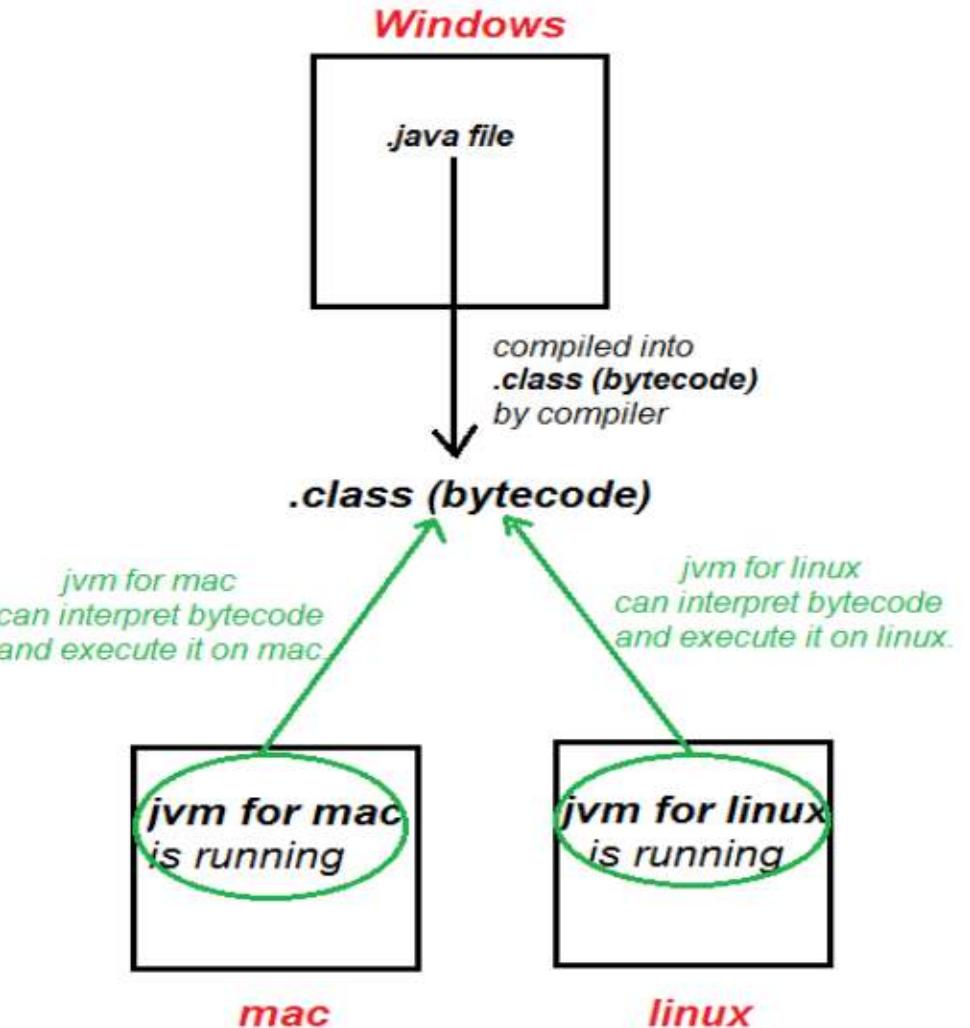


Fig. File was compiled on **windows** and could be executed on **mac** and **linux**.
Hence, making java platform independent.

JAVA KEYWORDS

- There are **49 reserved keywords** currently **defined** in the **Java** language
- These keywords **cannot be used** as **names for a variable, class, or method.**

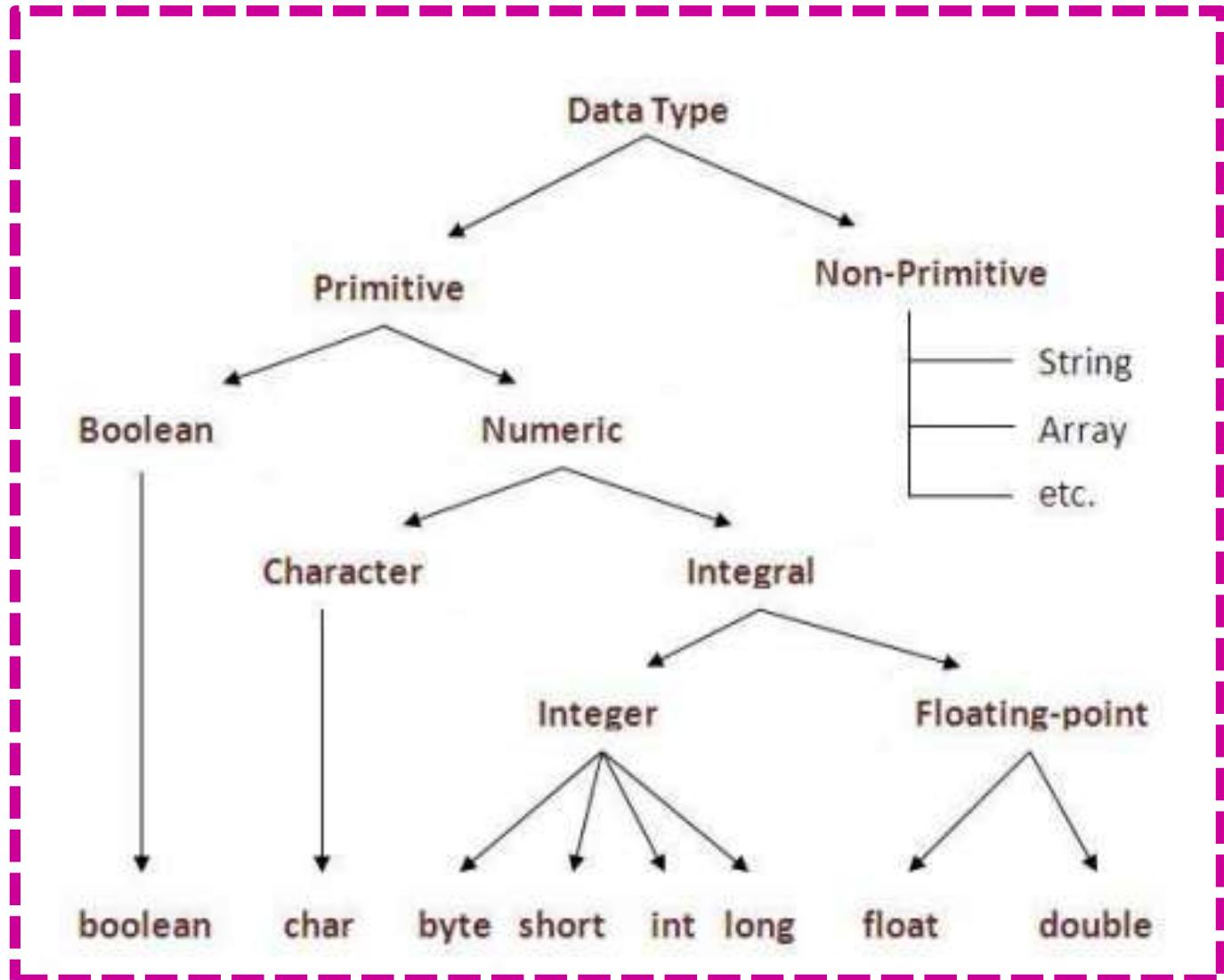
abstract	continue	goto	package	synchronized
assert	default	if	private	this
boolean	do	implements	protected	throw
break	double	import	public	throws
byte	else	instanceof	return	transient
case	extends	int	short	try
catch	final	interface	static	void
char	finally	long	strictfp	volatile
class	float	native	super	while
const	for	new	switch	

Data Types in Java

- “Data types **represent the different values** to be **stored in variable**”
- **Data type specifies:**
 - ✓ **Type of Value stored in a variable**
 - ✓ **Range of Values to a variable**
 - ✓ **Size of variable**
 - ✓ **Operations Performed**
- Java is a strongly typed language
 1. **Every variable has a type**, every expression has a type, and **every type is strictly defined**.
 2. **All assignments**, whether explicit or via parameter passing in method calls, are **checked for Type compatibility**.
 3. There are **no automatic conversions of conflicting types** as in **some languages**.
 4. The **Java compiler checks all expressions** and **parameters** to ensure that the types are compatible.
 5. Any **type mismatches are errors** that **must be corrected before** the compiler will **finish compiling the class**.

Data Types in Java

- In java, there are two types of data types:
 1. Primitive data types
 2. Non – Primitive Data Types



Data Types in Java

- All of **integer data types** are **signed, positive** and **negative values**.
- **Java does not support unsigned, positive-only integers.**
- Java implements the standard **IEEE-754** for **floating-point types**.
- There are **two kinds of floating-point types**, **float** and **double** which represent single precision (32bits) - and double precision (64bits) - numbers.
- **Java has** a **primitive type**, called **boolean**, for **logical values**. **It can have only one of two possible values, true or false.**

Data Types in Java

- Integer Type Range

Name	Width in Bits	Range
long	64 (8 Bytes)	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32 (4 Bytes)	-2,147,483,648 to 2,147,483,647
short	16 (2 Bytes)	-32,768 to 32,767
byte	8 (1 Byte)	-128 to 127

- Floating Point Range

Name	Width in Bits	Range
double	64 (8 Bytes)	4.9e-324 to 1.8e+308
float	32 (4 Bytes)	1.4e-045 to 3.4e+038

- Character type Range

Name	Width in Bits	Range
char	16 (2 Bytes)	0 to 65,535 (no negative characters) UNI – CODE system

Java Character System

- Java uses **Unicode system**. Unicode is a **universal international standard character encoding** that is capable of **representing most of** the world's written **languages**.
- **Before Unicode**, there were **many language standards like ASCII** (American Standard Code for Information Interchange).
- In Unicode, **character holds 2 byte**, so **java also uses 2 byte for characters**.
- Unicode defines a **fully international character set** that can **represent all of the characters** found in **all human languages** such as **Latin, Greek, Arabic, Cyrillic** etc.
- The **range** of a **char is 0 to 65,535**.
- There are **no negative chars**.
- Since **Java is designed** to allow programs **to be written for worldwide use**, it makes sense that **it would use Unicode to represent characters**.

Java Variables

- Variable is a **name of memory location**.

- A Variable Has

- ✓ **NAME (VALID IDENTIFIER)**
- ✓ **VALUE**
- ✓ **TYPE**
- ✓ **SCOPE & LIFETIME**

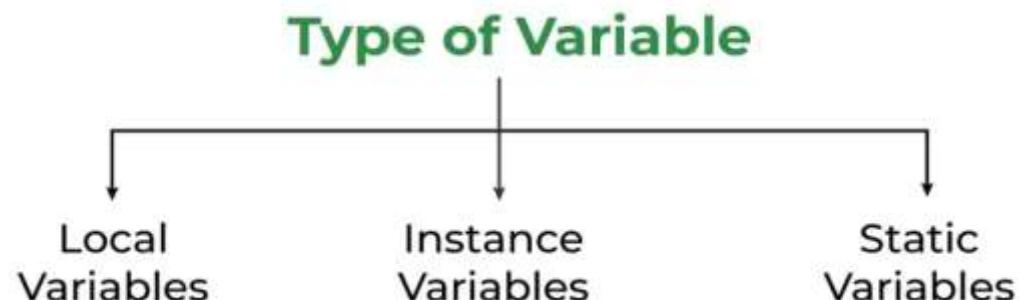
Syntax

```
datatype var_name = value;
```

```
int data=50; //Here data is variable  
int x,y;  
float p=3.142, result;  
char grade ='A';
```

Types of Variable:

- There are three types of variables in java:



Java Variables

i. Local Variable:

- ✓ A **variable** which is **declared inside** the **method** is called **local variable**. (Stored in Stack Area of Memory)
- ✓ These variables are **created when** the **block is entered**, or the **function is called** and **destroyed after exiting from the block** or when the call returns from the function.
- ✓ The **scope** of these variables exists **only within the block**.

ii. Instance Variable:

- ✓ A **variable** which is **declared inside** the **class** but **outside** the **method**, is called **instance variable**. This is not declared as static. (Stored in Heap area of Memory).
- ✓ These variables are **created when an object** of the **class is created** and **destroyed** when the **object is destroyed**.
- ✓ Its **default value** is **dependent on** the **data type** of variable. For **String** it is **null**, for **float** it is **0.0f**, for **int** it is **0**.

Java Variables

iii. Static variable:

- ✓ A **variable** that is **declared** as **static** is called **static variable**. It **cannot be local**. It **belongs to a class**. (Stored in Class Area of Memory).
- ✓ These static variables are declared **within a class outside of any method**, constructor, or block.
- ✓ we can only have **one copy of a static variable per class**, irrespective of how many objects we create.
- ✓ Static variables are **created at the start of program execution** and **destroyed automatically** when **execution ends**.

iv. Reference variable:

A **variable** that **refers to object** of a **class**. (Stored in Stack Area of Memory)

Java Variables



```
Scanner sc = new Scanner(System.in);           //sc-Reference variable-stack
class A
{
    int data=50;                            //instance variable-heap
    static int m=100;                         //static variable-class area
    void display( )
    {
        int n=90;                           //local variable- stack
        area
        -----
        -----
    }
}
```

Java Variables

Example

```
public class Simple
{
    public static void main(String[] args)
    {
        int a=10;
        int b=10;
        int c=a+b;
        System.out.println("The sum is " +c);
    }
}
```

Example

```
class DynInit
{
    public static void main(String args[])
    {
        double a = 3.0, b = 4.0;
        a=a+5;
        b=b-1;
        double c = Math.sqrt(a * a + b * b);
        System.out.println("Result is " + c);
    }
}
```

NOTE:
Java allows variables can be **initialized dynamically using any expression.**

Programs to Read Data from Standard Input



- Using **Java Scanner Class** to **read input**.
- Scanner is a class in “**java.util**” package used for obtaining the input of the primitive types like int, double, and strings etc.
- It is the **easiest way to read input** in a Java program.
- The **Java Scanner class breaks** the **input into tokens** using a **delimiter** that is **whitespace** by default.
- **Commonly used methods** of Scanner class

Syntax

```
Scanner sc=new Scanner(System.in);  
  
datatype Variable_Name=sc.method_name( );
```

Example

```
Scanner sc=new Scanner(System.in);  
  
int rollno=sc.nextInt( );
```

Programs to Read Data from Standard Input



Method	Description
<code>public String next()</code>	It returns the next token (string) from the scanner.
<code>public String nextLine()</code>	It moves the scanner position to the next line and returns the value as a string.
<code>public byte nextByte()</code>	It scans the next token as a byte .
<code>public short nextShort()</code>	It scans the next token as a short value.
<code>public int nextInt()</code>	It scans the next token as an int value.
<code>public long nextLong()</code>	It scans the next token as a long value.
<code>public float nextFloat()</code>	It scans the next token as a float value.
<code>public double nextDouble()</code>	It scans the next token as a double value.
<code>public double nextChar()</code>	It scans the next token as a charcater value.

// Example to read and display data

```
import java.util.*;
import java.io.*;
public class DataRead
{
    public static void main(String args[])
    {
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter your rollno");
        int rollno=sc.nextInt();
        System.out.println("Enter your name");
        String name=sc.next();
        System.out.println("Enter your fee");
        double fee=sc.nextDouble();
        System.out.println("Roll No : " +rollno);
        System.out.println("Name is " +name);
        System.out.println("Fee is " +fee);
        //System.out.println("Rollno:"+rollno +" name:" +name + " fee:" +fee);
        sc.close();
    }
}
```

```
//Program to read a number and check it is even or odd.  
import java.util.*;  
public class Even  
{  
    public static void main(String[ ] args)  
    {  
        Scanner sc = new Scanner(System.in);  
        System.out.println("Enter a number");  
        int n = sc.nextInt();  
        if( n % 2 == 0)  
            System.out.println("Even Number " +n);  
        else  
            System.out.println("Odd Number "+n);  
    }  
}
```

//Program to find the average of a student in three subjects marks (Double).

```
import java.util.*;
public class Grade
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the Marks in 3 Subjects");
        double m1 = sc.nextDouble();
        double m2 = sc.nextDouble();
        double m3 = sc.nextDouble();
        double avg = (m1+m2+m3)/3;
        System.out.println("The average is "+avg);
    }
}
```

Type Conversions and Type Casting



- When you **assign value of one data type to another**, the **two types might not be compatible** with **each other**.
- If the **data types are compatible**, then **Java will perform the conversion automatically** known as **Automatic Type Conversion**.

Example: it is always **possible to assign an int value to a long variable.**

- If the **data types are not compatible** then **they need to be casted** or **converted explicitly**.

Example: there is **no automatic conversion defined** from **double** to **byte**.

Java is a strongly typed language

- ✓ **Every variable has a type**, every **expression has a type**, and **every type is strictly defined**.
- ✓ All assignments, whether explicit or via parameter passing in method calls, are checked for Type compatibility.
- ✓ There are **no automatic conversions** of **conflicting types** as in **some languages**.
- ✓ The **Java compiler** checks **all expressions** and **parameters** to ensure **that the types are compatible**.
- ✓ **Any type mismatches** are **errors** that **must be corrected** before the **compiler will finish compiling the class**.

Type Conversions and Type Casting

Java Type conversion are in **two forms :**

1. Automatic Type Conversion / Widening / Type Coersion

- When **one type of data is assigned to another type of variable**, an automatic type conversion will take place if the following two conditions are met:
 - i. The **two types are compatible**.
 - ii. The **destination type is larger than the source type**.
- When these two conditions are met, a widening conversion takes place.

Example: **int** type is **always** large **enough to hold all valid byte values**, so **no explicit cast statement** is **required**.

For widening conversions

- The **numeric types**, including **integer and floating-point types**, are **compatible with each other**.

Eg: **int a=10;**

float f=a;

System.out.println(a); //10

System.out.println(f); //10.0

Type Conversions and Type Casting



- No automatic conversions from the **numeric types** to **char** or
- **char** and **boolean** are **not compatible** with **each other**. (In compatible Types)

2. Explicit Type Conversion / Narrowing / Type Casting

- The **automatic type conversions** will **not fulfill** all **needs**.
- For example an **int value** to a **byte variable conversion will not** be performed automatically, because a byte is smaller than an int.

```
int m = 67;  
  
byte b = m;  
  
System.out.println("The value of m is " +m); //67  
  
System.out.println("The value of ch is " +b);
```

- This kind of conversion is sometimes called a **narrowing conversion**, since we are **explicitly making** the **value narrower so that it will fit** into the **target type**.
- To create a **conversion between two incompatible** types, we **must use a cast**.

Operators in Java

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

- **Arithmetic Operators**
- **Relational Operators**
- **Bitwise Operators**
- **Logical Operators**
- **Assignment Operators**
- **Conditional Operators**

Operators in Java

i. Arithmetic Operators:

SR.NO	Operator and Example
1	+ (Addition) Adds values on either side of the operator Example: A + B will give 30
2	- (Subtraction) Subtracts right hand operand from left hand operand Example: A - B will give -10
3	* (Multiplication) Multiplies values on either side of the operator Example: A * B will give 200
4	/ (Division) Divides left hand operand by right hand operand Example: B / A will give 2
5	% (Modulus) Divides left hand operand by right hand operand and returns remainder Example: B % A will give 0
6	++ (Increment) Increases the value of operand by 1 Example: B++ gives 21
7	-- (Decrement) Decreases the value of operand by 1 Example: B-- gives 19

```

class OperatorDemo
{
    public static void main(String args[])
    {
        int a=10,b=5;
        System.out.println(a+b); //15
        System.out.println(a-b); //5
        System.out.println(a*b); //50
        System.out.println(a/b); //2
        System.out.println(a%3); //1
        double c=42.25;
        System.out.println(c%10); //2.25possiblein java
    }
}
  
```

Operators in Java

i. Arithmetic Operators:

```
class OperatorDemo
{
    public static void main(String args[])
    {
        int x=10;
        System.out.println(x++); //10 (11)
        System.out.println(++x); //12
        System.out.println(x--); //12 (11)
        System.out.println(--x); //10
    }
}
```

```
class OperatorDemo
{
    public static void main(String args[])
    {
        int a=10;
        int b=10;
        System.out.println(a++ + ++a); //10+12=22
        System.out.println(a); //12
        System.out.println(b++ + b++); //10+11=21
        System.out.println(b); //12
    }
}
```

Operators in Java

ii. Relational Operators (Comparing two things):

SR.NO	Operator and Description
1	== (equal to) Checks if the values of two operands are equal or not, if yes then condition becomes true. Example: (A == B) is not true.
2	!= (not equal to) Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. Example: (A != B) is true.
3	> (greater than) Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. Example: (A > B) is not true.
4	< (less than) Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. Example: (A < B) is true.
5	>= (greater than or equal to) Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. Example: (A >= B) is not true.
6	<= (less than or equal to) Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. example: (A <= B) is true.

```
class OperatorDemo
{
    public static void main(String args[])
    {
        int a = 4;
        int b = 1;
        boolean c = a < b;
        System.out.println("The C value is " + c) //false
        System.out.println(a==4) //True
        System.out.println(a!=b) //False
        System.out.println(a<=4) //True
    }
}
```

Operators in Java

iii. Logical Operators (Combining two or more relations):

Operator	Description
1	&& (logical and) <p>Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.</p> <p>Example (A && B) is false.</p>
2	 (logical or) <p>Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.</p> <p>Example (A B) is true.</p>
3	! (logical not) <p>Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.</p> <p>Example !(A && B) is true.</p>

```
class OperDemo
{
    public static void main(String args[])
    {
        int a=10;
        int b=5;
        int c=20;
        System.out.println(a<b&&a<c);
        System.out.println(a<b| |a<c);
    }
}
```

Operators in Java

iv. Bitwise Operators (Operates at bit level – on binary data):

Operator	Result
<code>~</code>	Bitwise unary NOT (one's complement)
<code>&</code>	Bitwise AND
<code> </code>	Bitwise OR
<code>^</code>	Bitwise exclusive OR
<code>>></code>	Shift right
<code>>>></code>	Shift right zero fill
<code><<</code>	Shift left
<code>&=</code>	Bitwise AND assignment
<code> =</code>	Bitwise OR assignment
<code>^=</code>	Bitwise exclusive OR assignment
<code>>>=</code>	Shift right assignment
<code>>>>=</code>	Shift right zero fill assignment
<code><<=</code>	Shift left assignment

```
class Operator
{
    public static void main(String args[])
    {
        byte a=10;
        byte b=8;
        byte c;
        c = a | b;
        System.out.println(c);
        c = a & b;
        System.out.println(c);
        c = a ^ b;
        System.out.println(c);
    }
}
```

Operators in Java

iv. Bitwise Operators:

```
class Oper
{
    public static void main(String args[])
    {
        byte a=10;
        System.out.println(a<<2); //40
        byte b=-15;
        System.out.println(b<<3); //-120
        byte c=10;
        System.out.println(c>>2); //2
        byte d=-15;
        System.out.println(d>>1); //-8
    }
}
```

Operators in Java

v. Assignment Operators:

SR.NO	Operator and Description
1	= Simple assignment operator, Assigns values from right side operands to left side operand. Example: $C = A + B$ will assign value of $A + B$ into C
2	+= Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand. Example: $C += A$ is equivalent to $C = C + A$
3	-= Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand. Example: $C -= A$ is equivalent to $C = C - A$
4	*= Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand. Example: $C *= A$ is equivalent to $C = C * A$
5	/= Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand Example: $C /= A$ is equivalent to $C = C / A$

Operators in Java

v. Conditional Operator:

- Conditional operator is also known as the **ternary operator**.
- This operator consists of **three operands** and is used to evaluate expressions.
- The **goal of the operator is to decide which value should be assigned to the variable**.

Syntax

variable x = (expression) ? value if true : value if false

```
class Test
{
    public static void main(String args[ ])
    {
        int a, b;
        a = 10;
        b = (a == 1) ? 20: 30;
        System.out.println( "Value of b is : " + b );
        b = (a == 10) ? 20: 30;
        System.out.println( "Value of b is : " + b );
    }
}
```

Operator Precedence Table in Java

Operators	Notation	Precedence/Priority
Postfix	expr++, expr--	1
Unary	++expr, --expr, +expr -expr, ^, !	2
Multiplicative	*	3
Additive	+, -	4
Shift	<<, >>, >>>	5
Relational	<, >, <=, >=, instanceof	6
Equality	==, !=	7
Bitwise AND	&	8
Bitwise Exclusive OR	^	9
Bitwise Inclusive OR		10
Logical AND	&&	11
Logical OR		12
Ternary	? :	13
Assignment	=, +=, -=, *=, /=, %=, &=, ^=, = , <<=, >>=, >>>=	14

Control Statements

1. Conditional / Selection Statements:

-There are **various types** of **selection statement** in java.

1. if statement
2. if-else statement
3. Nested if statement
4. if-else-if ladder
5. Switch Statement

i. if statement

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

ii. if-else statement

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

iii. Nested if statement

```
if(condition1)
{
    if(condition2)
    {
        //code to be executed
    }
    //code to be executed
}
```

iv. if-else-if ladder

```
if(condition1)
{
    //code to be executed
}
else if(condition2)
{
    //code to be executed
}
else
{
    //code to be executed
}
```

v. Switch Statement

```
switch(expression)
{
    case value1: //code to be executed;
                  break;
    case value2: //code to be executed;
                  break;
    .....
    default: //code to be executed;
}
```

Control Statements

class Example

```
{public static void main(String[] args)
{
    int number=20;
    switch(number)
    {
        case 10: System.out.println("CSE");break;
        case 20: System.out.println("EEE");break;
        case 30: System.out.println("ECE");break;
        default:System.out.println("Invalid input");
    }
}}
```

Switch Statement is fall-through



- The java switch statement is fall-through. It **means it executes all statement after first match if break statement is not used** with switch cases.

```
class Example2
{
    public static void main(String[] args)
    {
        int number=20;
        switch(number)
        {
            case 10: System.out.println("10");
            case 20: System.out.println("20");
            case 30: System.out.println("30");
            default:System.out.println("Hello");
        }
    }
}
```

Control Statements

2. Loop / Repetitive / Iterative Statements :

i. while Loop:

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

Syntax:

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

Java Infinitive While Loop

- ✓ If you **pass true** in the while loop, it will be **infinitive while loop**.

```
while(true)
{
    System.out.println("never ends");
}
```

Control Statements

ii. do-while Loop:

- ✓ do-while loop is used to iterate a part of the program several times.
- ✓ If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.
- ✓ The Java do-while loop is executed at least once because condition is checked after loop body.
- ✓ It is an exit controlled loop

Syntax

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

- ✓ If you pass true in the do-while loop, it will be infinitive do-while loop.

```
do
{
    System.out.println("Never Ends");
}while(true);
```

Control Statements

iii. for Loop:

- ✓ The Java **for loop** is used to iterate a part of the program **several times**. If the **number of iteration is fixed**, it is **recommended to use for loop**.
- ✓ It is an **entry controlled loop**.
- ✓ There are two types of for loop in java.
 - a. **Simple For Loop**
 - b. **for-each or Enhanced For Loop**

a. Simple for Loop:

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

Syntax

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

Example

```
for(int i=1;i<=10;i++)
{
    System.out.println(i);
}
```

Control Statements

b. Java for-each Loop:

- ✓ The **for-each loop** is used to traverse array or **collection in java**. It is **easier** to use than **simple for loop** because we **don't need to increment** value and use subscript notation.
- ✓ It works on **elements basis not index**.
- ✓ It **returns element one by one** in the defined variable.

Syntax

```
for(type var: array)
{
    //code to be executed
}
```

Example

```
int list[] = { 11, 2, 3, 14, 5, 62, 7, 8, 9, 10 };
int sum = 0;
for(int x : list)
{
    System.out.println("Value is: " + x);
    sum = sum + x;
}
```

Java Infinitive for Loop

```
for( ; ; )
{
    System.out.println("Never Ends");
}
```

Java break Statement

- The Java break is used to **break loop** or **switch statement**. It breaks the current flow of the program at specified condition.
- When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.
- **In case of inner loop, it breaks only inner loop.**

Example

```
for(int i=1;i<=10;i++)
{
    if(i==5)
    {
        break;
    }
    System.out.println(i);
}
```

Java Continue Statement

- The Java **continue statement** is used to **continue loop**.
- We might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration.
- It **continues the current flow** of the program and **skips the remaining code at specified condition**.
- In case of inner loop, it continues only inner loop.

Example

```
for(int i=1;i<=6;i++)
{
    if(i==3)
    {
        continue;
    }
    System.out.println(i);
}
```

Java return Statement

- The return statement is used to **explicitly return from a method**. That is, it **causes program control to transfer back to the caller** of the method.
- Thus, the return statement **immediately terminates the method in which it is executed**.

```
class Demo
{
    public static void main(String[] args)
    {
        int res;
        res = add(20,30);
        System.out.println("The result is "+res);
    }
    public static int add(int a , int b)
    {
        return(a+b);
    }
}
```

Arrays In Java

An array is **a group of like-typed (homogeneous) elements** that are **referred to by a common name**.

- ✓ **Arrays of any type** can be **created** and **may have one or more dimensions**. (One ,Two or More)
- ✓ A **specific element** in an **array is accessed** by **its index**. (index starts from 0)
- ✓ The **memory for an array is dynamically allocated** unlike C or C++.

i. One-Dimensional Arrays:

- ✓ A one-dimensional array is, essentially, a **list of like-typed variables**.
- ✓ The **general form** of a one-dimensional array declaration **is**:

Syntax:

Datatype var-name[];

Example

int days[];
double avg[];

- ✓ **Although this declaration** an array variable, **no array actually exists**.

Arrays In Java

- ✓ The **array is set to null**, which represents an **array with no value**.
- ✓ To create **physical array of integers**, we must **allocate using new operator**.
- ✓ **new** is a **special operator** that **allocates memory dynamically**.
- ✓ A one-dimensional array can be created as

Syntax:

```
array-var = new type [size];  
           //creation or allocation
```

Example

```
int days = new int[7];  
double avg = new double[60];
```

- ✓ Both **declaration** and **allocation** can be combined as

Syntax:

```
Datatype array-var = new Datatype [size];  
           //creation or allocation
```

Example

```
int days = new int[7];  
double avg = new double[60];
```

Arrays In Java

```
class Array
{
    public static void main(String args[])
    {
        int days[] = new int[7];
        days[0] = 12; //assigning a value at index
        days[1] = 22;
        days[2] = 92;
        days[3] = 28;
        days[4] = 32;
        days[5] = 40;
        days[6] = 39;
        //Accessing an array
        for(int i=0; i<6;i++)
        {
            System.out.println( days[i]);
        }
    }
}
```

Array Initialization



- ✓ Arrays **can be initialized when they are declared.**
- ✓ An array initializer is a **list of comma-separated** expressions **Surrounded by curly braces.**
- ✓ The commas separate the values of the array elements.

Syntax:

Datatype arr_name[size] = { List of elements separated by comma};

Example

int days[7] = {7,9,5,12,45,23,10};

- ✓ The **size of array need not to be specified.**

```
class Average
{
    public static void main(String args[])
    {
        double a[] = {75.1, 76.2, 65.3, 77.4, 84.5};
        double sum = 0;
        int i;
        for(i=0; i<5; i++)
            sum = sum + a[i];
        System.out.println("Average is " + sum / 5);
    }
}
```

Multi-Dimensional Arrays

- In Java, multidimensional **arrays are actually arrays of arrays**.
- To declare a multidimensional array variable, **specify each additional index using another set of square brackets**.

Example:

A two-dimensional array can be declared and allocated as

```
int two[ ][ ] = new int[4][5];
```

- When you allocate memory for a multidimensional array we **need to** only **specify** the memory for the **first (leftmost) dimension**.
- We can **allocate the remaining dimensions separately**.

```
int a[][] = new int[4][ ];  
a[0] = new int[3];  
a[1] = new int[3];  
a[2] = new int[3];  
a[3] = new int[3];
```

```
class TwoD  
{  
    public static void main(String args[])  
    {  
        int a[][]= new int[4][5];  
        int i, j, k = 0;  
        for(i=0; i<4; i++)  
            for(j=0; j<5; j++)  
            {  
                a[i][j] = k;  
                k++;  
            }  
        for(i=0; i<4; i++)  
        {  
            for(j=0; j<5; j++)  
                System.out.print(a[i][j] + " ");  
            System.out.println();  
        }  
    }  
}
```

Multi-Dimensional Arrays

- We can allocate **each row with different number of elements**; such an array is called **Jagged / Ragged Array**. i.e. Each row can be of different columns.

```
int a[][] = new int[4][ ];
```

```
a[0] = new int[2];
```

```
a[1] = new int[3];
```

```
a[2] = new int[4];
```

```
a[3] = new int[5];
```

- A multi dimensional array can be initialized as

```
int a[3][3] = {{1,2,3}, {3,4,5},{6,7,8}};
```

```
int a[3][] = {{1,2,3}, {3,4,5},{6,7,8}};
```

```
int a[4][] = {{1,2,3,4}, {3,4 },{6,7,8,9},{1,2}}; //Jagged array
```

Multi-Dimensional Arrays



```
import java.io.*;
import java.util.*;
//Jagged or Ragged Array
class ArrDemo
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the number of rows");
        int m = sc.nextInt();
        int a[][] = new int[m][ ];
        int n;
        for(int i =0; i< m;i++)
        {
            System.out.println("Enter the number of elements in " +i);
            n = sc.nextInt();
            a[i] = new int[n];
        }
    }
}
```

```
System.out.println("Enter the elements");
for(int i = 0; i<m ; i++)
for(int j=0;j<a[i].length;j++)
a[i][j] = sc.nextInt();
System.out.println("the matrix elements are");
for(int i = 0; i<m ; i++)
{
    for(int j=0;j<a[i].length;j++)
    System.out.print(" " +a[i][j]);
    System.out.println();
}
}
```

//3. Matrix Multiplication

```
import java.util.*;
public class MatrixDemo
{
    public static void main(String args[])
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the size of Matrix A - m * n");
        int m = sc.nextInt();
        int n = sc.nextInt();

        System.out.println("Enter the size of Matrix B - p * q");
        int p = sc.nextInt();
        int q = sc.nextInt();
        if (n != p)
        {
            System.out.println("Multiplication cannot be performed");
            System.exit(0);
        }
        int a[][] = new int[m][n];
        int b[][] = new int[p][q];
        int c[][] = new int[m][q];
```

```
System.out.println("Enter the elements of Matrix-A");
for(int i =0; i<m;i++)
for(int j=0;j<n;j++)
a[i][j] = sc.nextInt();
System.out.println("Enter the elements of Matrix-B");
for(int i =0; i<p;i++)
for(int j=0;j<q;j++)
b[i][j] = sc.nextInt();
// Actual Code
for(int i =0 ; i<m ;i++)
{
for(int j=0;j<q;j++)
{
c[i][j]=0;
for(int k=0;k<p;k++)
c[i][j] = c[i][j] + a[i][k]*b[k][j];
}
}
```

```
System.out.println("The Product Matrix-C is");
for(int i =0; i<m;i++)
{
for(int j=0;j<q;j++)
System.out.print(" " + c[i][j]);
System.out.println();
}
```

- Alternative Array Declaration Syntax

```
type[ ] var-name;  
int a[] = new int[3];  
int[] a1 = new int[3];  
char a[][] = new char[3][4];  
char[][] a = new char[3][4];
```

- when **declaring several arrays at the same time.**

```
int[] a,b,c; // create three arrays  
char[][] a,b,c;
```

Classes and Objects

- The class is the **logical construct** upon which the entire **Java language is built** .
- The **class** forms the **basis** for **object-oriented programming** in Java.
- **Any concept** we implement **in a Java** program **must be encapsulated** within a **class**.
- A class is **declared by use** of the **class keyword**.
- A **class defines a new data type**.
- A class is a **template for an object**, and an **object** is an **instance** of a **class**.
- A **class contains data** and the **code** that operates on that data.
- The **data, or variables**, defined **within a class** are called **instance variables**, because **each object** of the **class contains** its **own copy** of these **variables**.
- The **methods** and **variables** defined **within a class** are called **members of the class**.

Classes and Objects

- The **general form of a class** is :

```
modifier class ClassName
{
    modifier type instance-variable1;
    modifier type instance-variable2;
    -----
    modifier type instance-variableN;
    modifier type methodname1(parameter-list)
    {
        // body of method
    }
    modifier type methodname2(parameter-list)
    {
        // body of method
    }
}
```

Classes and Objects

Reference variables can be assigned

Example-1:

```
Box mybox; //declaration  
  
mybox=new Box(); //Allocation
```

Example-2:

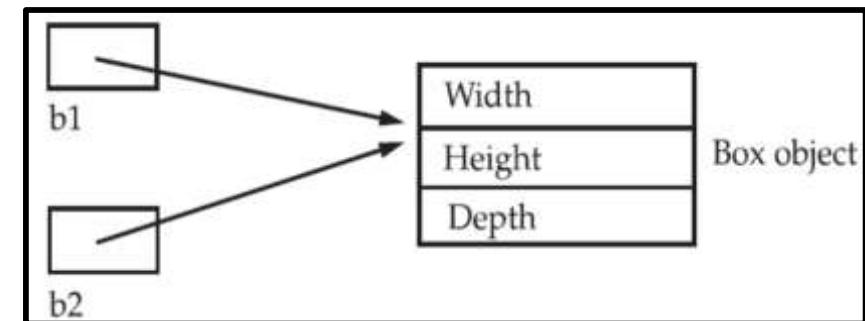
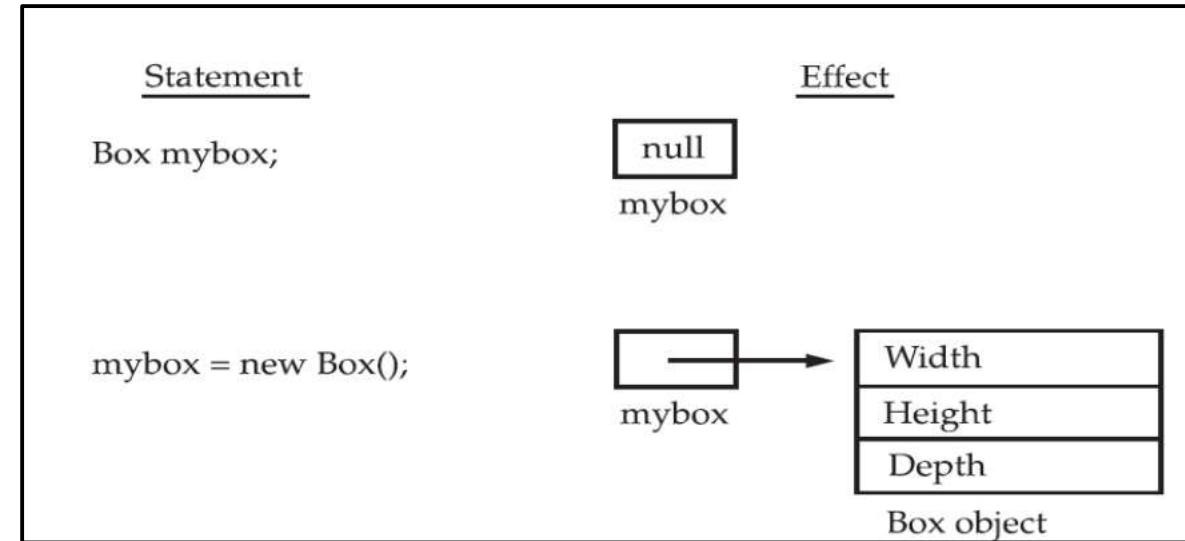
```
Box b1 = new Box();  
  
Box b2 = b1; // b1, b2 refers to same object
```

Example-3:

```
Box b1 = new Box() , b2 = new Box(); // both are different
```

Example-4:

```
Box b1 = new Box();  
  
Box b2 = new Box(); // both are different
```



Methods

- **type specifies** the type of **data returned** by the **method**.
- If the **method does not return a value**, its **return type must be void**.
- The **name of the method** is **any valid identifier** name.
- The **parameter-list** is a sequence of type and identifier pairs **separated by commas**.
- If the **method has no parameters**, then the parameter **list will be empty**.

Syntax

```
type methodName(parameter-list)
{
    // body of method
}
```

Example

```
class Box
{
    double width;
    double height;
    double depth;
    double volume()
    {
        double x; // local variable
        x= width * height * depth;
        return x;
    }
    void setDim(double w, double h, double d)
    {
        width = w;
        height = h;
        depth = d;
    }
}
```

```
class BoxDemo
{
    public static void main(String args[])
    {
        Box b1= new Box(); //b1 – reference variable
        Box b2 = new Box(); //b2-reference variable
        b1.setDim(10, 20, 15);
        b2.setDim(3, 6, 9);
        double vol1 = b1.volume();
        System.out.println("box1 Volume is " +vol1);
        System.out.println("box1 Volume is " +b2.volume());
    }
}
```

Example

Example-2

```
class Emp
{
    int id;
    String name;
    double salary;
    void insert(int i, String n, double s)
    {
        id=i;
        name=n;
        salary=s;
    }
    void display()
    {
        System.out.println(id+" "+name+" "+salary);
    }
}
```

```
class TestEmp
{
    public static void main(String[] args)
    {
        Emp e1=new Emp();
        Emp e2=new Emp(),e3=new Emp();
        e1.insert(101,"Varun",45000);
        e2.insert(102,"Tharun",25000);
        e3.insert(103,"Nirun",55000);
        e1.display();
        e2.display();
        e3.display();
    }
}
```

Forms of Methods



1. With No Parameters and No Return Value

```
void volume()
{
    double x;
    x= width * height * depth;
    System.out.println("The Volume is " +x);
}
```

2. With Parameters and No Return Value

```
void volume(double len , double wid, double hgt)
{
    double x=len*wid*hgt;
    System.out.println("The Volume is " +x);
}
```

3. With Parameters and a Return Value

```
double volume(double len , double wid, double hgt)
{
    double x= len*wid*hgt;
    return x;
}
```

4. With No Parameters and a Return Value

```
double volume()
{
    double x;
    x= width * height * depth;
    return x;
}
```

Constructors

- Java Allows **objects to be initialized themselves** when they are **created**.
- **Constructor** in java is a special type of method that is **used to initialize the object**.
- Java **constructor is invoked at the time of object creation**. It **provides data** for the **object** hence it is **known as constructor**.

Characteristics:

- Once defined, constructor is **automatically called after the object is created**,.
- Syntactically **similar to a method**
- The **name of the constructor must be** same as **its class name**.
- If **no constructor is defined**, a **default constructor** is **invoked by Java**.
- **Constructor must not** have **explicit return type**. By **default a constructor returns the class instance after creation**.

Rules for creating java constructor

There are basically two rules defined for the constructor.

- i. **Constructor name must be same as its class name**
- ii. **Constructor must have no explicit return type**

Types of constructors

- There are **two types of constructors:**

i. Default constructor (No-Argument constructor):

- Default constructor **provides the default values** to the object like 0 (int), 0.0(double), null (String) etc. **depending on the type.**
- The Constructor **has no arguments.**
- **Initializes all objects** to same, hence no longer used.

Box()

{

Len = wid= hgt =0.0;

}

Eg: Box b1 = new Box();

Box b2 = new Box();

//Example for default constructor-1

```
class Bike
```

```
{
```

```
Bike()
```

```
{
```

```
System.out.println("Bike is created");
```

```
}
```

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

```
{
```

```
Bike b=new Bike ();
```

```
}
```

```
}
```

// Example for default constructor-2

```
class Box
{
    double len ;
    double wid;
    double hgt;
    Box()
    {
        Len = wid= hgt =0.0;
    }
    double volume( )
    {
        double res;
        res = len * wid * hgt;
        return res;
    }
    void display()
    {
        System.out.println("The Dimensions of box are");
        System.out.println("Length= "+len +"Width= " +wid + "Height= " +hgt);
    }
}
```



```
class const1
{
    public static void main(String args[])
    {
        Box b1 = new Box();
        b1.display();
        System.out.println("The volume of b1 is " + b1.volume());
        Box b2 = new Box();
        b2.display();
        System.out.println("The volume of b2 is " + b2.volume());
    }
}
```

ii. Parameterized constructor:

- A **constructor that has parameters** is known as **parameterized constructor**.
- **Used to create objects of different state and type.**
- Parameterized constructor is **used to provide different values** to the **distinct objects**.

Box(double x)

```
{  
    len = wid= hgt =x;  
}
```

Box(double x ,double y , double z)

```
{  
    len = x;  
    wid = y;  
    hgt = z;  
}
```

//Example for Parameterized constructor-1

```
class Student
{
    int id;
    String name;
    Student (int i,String n)
    {
        id = i;
        name = n;
    }
    void display()
    {
        System.out.println(id+" "+name);
    }
    public static void main(String args[])
    {
        Student s1 = new Student (111,"Ramesh");
        Student s2 = new Student (222,"Suresh");
        s1.display(); s2.display();
    }
}
```

//Example for Parameterized constructor-2

```
class Box
{
    double len ;
    double wid;
    double hgt;
    Box(double x ,double y , double z)
    {
        len = x;
        wid = y;
        hgt = z;
    }
    double volume( )
    {
        double res;
        res = len * wid * hgt;
        return res;
    }
    void display()
    {
        System.out.println("The Dimensions of box are");
        System.out.println(len + ":" +wid + ":" +hgt);
    }
}
```

```
class const1
```

```
{
    public static void main(String args[])
    {
        Box b1 = new Box(4,5,6);
        b1.display();
        System.out.println("The volume of b1 is " + b1.volume());
        Box b2 = new Box(7.5,8.5,12.5);
        b2.display();
        System.out.println("The volume of b2 is " + b2.volume());
    }
}
```

iii.Copy constructor:

- There is **no copy constructor specific in java**. But, **we can copy** the values of **one object to another**.
- **Copy one object into another By constructor.**
- By **assigning the values of one object into another**.

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

```
public static void main(String args[])
{
    Student s1 = new Student(1001, "Varun");
    Student s2 = new Student(s1);
    s1.display();
    s2.display();
}
```

Difference between constructor and method in java



i. Does constructor return any value?

Yes, that is current class instance. (**We cannot** use **return type explicitly**, it returns an **object** by constructing it).

ii. Can constructor perform other tasks instead of initialization?

Yes, like **object creation, starting a thread, opening a file, calling method** etc. We **can perform any operation** in the constructor **like** we perform in the **method**.

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

this keyword

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

Usage of this keyword

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

i. How 'this' refer current class instance variable:

- When a **local variable has the same name** as an **instance variable**, the local variable **hides** the **instance variable**. This is called **instance variable hiding.(Naming Collision)**.
- we can use this keyword to **resolve any namespace collisions** that might occur **between instance** variables and **local variables**.

this keyword

Understanding the problem without this keyword

```

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

Output

0 null 0.0
0 null 0.0

Solution of the above problem by this keyword

```

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

Output

111 ankit 5000.0
112 sumit 6000.0

ii. How 'this' invoke current class method:

- We may **call the method** of the **current class** by using this keyword.
- If we don't use the keyword, **compiler automatically adds this** keyword **while calling the method.**

```
class College
{
    void VCE( )
    {
        System.out.println("hello VCE");
    }
    void IT( )
    {
        System.out.println("hello IT");
        this.hello();
    }
}
class Test
{
    public static void main(String args[])
    {
        College t1=new College();
        t1.IT();
    }
}
```

Output
hello n
hello m

iii. How 'this()' invoke current class constructor:

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

```
class Student
{
    int rollno;
    String name;
    float fee;
    Student(int r,String n)
    {
        rollno=r;
        name=n;
    }
    Student(int r,String n,float f)
    {
        this(r,n);          //reusing constructor
        fee=f;
    }
    void display()
    {
        System.out.println(rollno+" "+name+" "+fee);
    }
}
```

```
class Test
{
    public static void main(String args[])
    {
        Student s1=new Student(111,"Ram");
        Student s2=new Student(112,"Rishi",6000f);
        s1.display();
        s2.display();
    }
}
```

Output
111 Ram 0.0
112 Rishi 6000.0

iv. How to pass 'this' as an argument in the method :

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

```
class Test
{
    void VCE(Test obj)
    {
        System.out.println("method is invoked");
    }
    void IT()
    {
        VCE(this);
    }
    public static void main(String args[])
    {
        Test t1 = new Test();
        t1.IT();
    }
}
```

Output
method is invoked

static Keyword

- If we **apply static keyword with any method then the method is called as** static method.
- The restrictions of static on methods declared is:
 - i. They can **directly call other static methods.**
 - ii. They can **directly access static data.**
 - iii. They **cannot refer to with “this” or “super” keys** in any way.
- A **static method or variable belongs** to a class rather than **object**.
- A static method can be **invoked without** the **need for creating an object of a class**.
- **static method cannot access non static variables** or **non static methods directly**.
- It is **used to initialize or modify** the **static data**.
- It **can be accessed directly** using **ClassName**.

Syntax to access Static Variable

ClassName.Variable_Name;

Syntax to access Static Method

ClassName.methodName();

Why Java main() method is static?

Java main method is static because

- i) An **object is not required to call** static method.
- ii) If it is **non static method, JVM creates object first then call main()** this leads to problem of **extra memory allocation.**

static variable example

//Program for static variable

```
class Student
{
    int rno;                                //instance variable
    String name;
    static String college ="VCE"; //static variable

    Student(int r, String n)
    {
        rno=r;
        name=n;
    }
    void display ()
    {
        System.out.println(rno+" "+name+" "+college);
    }
}
```

class TestStudent

```
{
    public static void main(String args[])
    {
        Student s1 = new Student(1201,"Akhil");
        Student s2 = new Student(1202,"Avinash");
        s1.display();
        s2.display();
        Student.college="NIT";
        s1.display();
        s2.display();
    }
}
```

Output		
1201	Akhil	VCE
1202	Avinash	VCE
1201	Akhil	NIT
1202	Avinash	NIT

static variable and Instance variable Difference



//Program for instance variable

```
class Test
{
    int count=0;

    Test()
    {
        count++;
        System.out.println(count);
    }

    public static void main(String args[])
    {
        Test t1=new Test();
        Test t2=new Test();
        Test t3=new Test();
    }
}
```

Output

```
1  
1  
1
```

//Program for static variable

```
class Test
{
    static int count=0;

    Test()
    {
        count++;
        System.out.println(count);
    }

    public static void main(String args[])
    {
        Test t1=new Test();
        Test t2=new Test();
        Test t3=new Test();
    }
}
```

Output

```
1  
2  
3
```

static method

//static method example -1

```
class Test
{
    static int cube(int x)
    {
        return x*x*x;
    }
    static void display()
    {
        int y ;
        y = cube(6);
        System.out.println(y);
    }
}
class StaticDemo1
{
    public static void main(String args[])
    {
        int result=Test(cube(5)); //Directly using class
        name
        System.out.println(result)
        Test.display();
    }
}
```

Output

125
216

Method Overloading

- **Polymorphism** in Java is a concept by which we **can perform a single action in different ways**. i.e **Poly** means “many” and **morphs** means “forms”.
- The two type of Polymorphism
 - i. **Compile Time polymorphism**
 - ii. **Run Time Polymorphism**
- **Compile Time polymorphism** in Java is **achieved** using “**method overloading**”.
- **Defining two or more methods within the same class** that **share the same name** but **parameter declarations are different**. The **methods are** said to be **overloaded** and **the process** is referred as **method overloading**.
- When an **overloaded method** is **invoked** uses **type** and **number of arguments** to **determine which version** is **called**.
- Overloaded methods **must differ in type** and/or **number of parameters**.
- When an overloaded method is called, **Java simply executes** the version of the **method whose parameters match the arguments used in the call**.

Method Overloading

Advantage of method overloading

- i. Method **overloading increases** the **readability** of the **program**.
- ii. Provides **Compile time Polymorphism**
- Different ways to overload the method
- There are **two ways to overload the method** in java
 - i. **By changing number of arguments**
 - ii. **By changing the data type**

i.Method Overloading: By changing number of arguments

```
class Adder
{
    int add(int a,int b)
    {
        return a+b;
    }
    int add(int a,int b,int c)
    {
        return a+b+c;
    }
}
class Test
{
    public static void main(String[] args)
    {
        Adder a = new Adder();
        System.out.println(a.add(10,11));      //2-args Version
        System.out.println(a.add(10,11,12));  //3-args Version
    }
}
```

Note

When a overloaded method is called, Java **executes the version of the method whose parameters match** the arguments used in the call.

ii. Method Overloading: changing data type of arguments

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

Overloading Constructors

- Constructor overloading is a technique in which a **class can have any number of constructors** that **differ in parameter lists**.
- The **compiler differentiates** these constructors **by taking** into account the **number of parameters** in the list and their data type.
- Like methods, the **constructors in java can also be overloaded**.
- This **process is called constructor overloading**.

Overloading Constructors

//Example of Constructor Overloading -1

```
class Student
{
    int id;
    String name;
    int age;
    Student (int i,String n)
    {
        id = i;
        name = n;
    }
    Student (int i,String n,int a)
    {
        id = i;
        name = n;
        age=a;
    }
}
```

```
void display()
{
    System.out.println(id+" "+name+" "+age);
}

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

Parameter Passing in Java

1. Pass By Value:

- Any modifications to the **formal parameter** variable inside the called method affect only the separate storage location and **will not be reflected** in the **actual parameter** in the calling environment.
- Inefficiency in storage allocation. For objects and arrays, the copy semantics are costly

```
//Parameter Passing Program - Call By Value
class Test
{
    public static void update(int x, int y)
    {
        x++;
        y++;
        System.out.println("x,y values are" +x + " , " +y); //11,21
    }
}
```

```
class ByVal
{
    public static void main(String[] args)
    {
        int a = 10;
        int b = 20;
        Test t = new Test();
        System.out.println("The values of a, b before method call");
        System.out.println("a, b values are" + a + "," + b); //10,20
        t.update(a, b);
        System.out.println("The values of a, b after method call");
        System.out.println("a, b values are" + a + "," + b); //10,20
    }
}
```

Parameter Passing in Java

2. Call by reference(aliasing):

- Any changes to the formal parameter are reflected in the actual parameter in the calling environment as formal .
- This method is efficient in both time and space.

```
class Demo
{
    int a, b;
    void update(Demo obj)
    {
        obj.a += 10;
        obj.b += 20;
    }
}
class ByRef
{
    public static void main(String[] args)
    {
        Demo d = new Demo();
        d.a = 10;
        d.b = 20;
        System.out.println("a, b before method call");
        System.out.println( d.a + "," + d.b); //10,20
        d.update(d); // Method Call
        System.out.println("a, b after method call");
        System.out.println("a, b values are" + d.a + "," + d.b); //20,40
    }
}
```

String Class

- String is a **sequence of characters**. But in Java, **string is an object** that **represents a sequence of characters**.
- The **java.lang.String class** is **used to create** a **string object**.
- Java String class **provides methods to perform operations on string**
- A String **object is created using**:

By String Literal

- Java **String literal** is **created** by using **double quotes**.

```
String s="welcome";
```

- Each time you create a string literal, the **JVM checks the "string constant pool"** first. **If the string already exists** in the **pool**, a **reference** to the pooled **instance is returned**.
- If the **string doesn't exist** in the **pool**, a **new string instance is created** and placed in the pool.
- To make Java more memory efficient (because **no new objects are created if it exists already** in the **string constant pool**).

```
String s1="Welcome";
```

```
String s2="Welcome"; //It doesn't create a new instance
```

String Class

By new keyword.

- String class **supports several constructors** to create strings.

1. String s = new String(); //default constructor – empty string

2. String s=new String("Welcome");

3. String(char ch[]); // parameterized constructor - char array to string object

```
char ch[] = { 'w', 'e', 'l','c','o','m','e' };
```

```
String s1 = new String(ch); // "welcome"
```

4. String(char ch[], int startIndex, int numChars) // parameterized constructor – char array to string object

```
char ch[] = { 'w', 'e', 'l','c','o','m','e' };
```

```
String s2 = new String(ch, 3,4); // "come"
```

5. String(String strObj) // copy constructor

```
String s1 = new String("Hyderabad");
```

```
String s3 = new String(s1);
```

String Class

6. String(byte asciiChars[])

```
String(byte ascii[ ])
```

```
byte ascii[] = {65, 66, 67, 68, 69, 70};
```

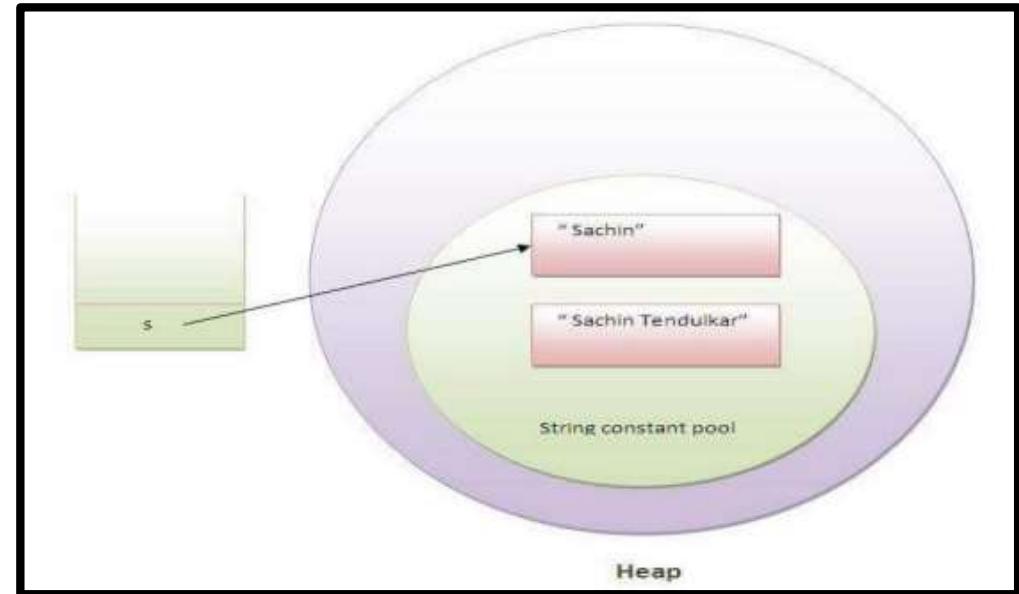
```
String s1 = new String(ascii); //ABCDEF
```

```
System.out.println(s1); //”ABCDEF”
```

- The Java **String is immutable** which **means it cannot be changed**.
- Whenever **we change any string**, a **new instance is created**.
- Once **string object is created its data or state can't be changed** but a **new string object is created with changes**.

String Class

```
class String1
{
    public static void main(String args[])
    {
        String s="Sachin";
        s.concat(" Tendulkar");
        System.out.println(s); // Sachin
    }
}
```



- Here **Sachin** is **not changed** but a **new object** is **created with Sachin Tendulkar**.
- **That is why string** is known as **immutable**.
- As you can see in the above figure that **two objects** are created but **s reference variable still** refers to "**Sachin**" not to "**Sachin Tendulkar**".
- But **if we explicitly assign** it to the **reference variable**, it will **refer to "Sachin Tendulkar"** object.

String Class



```
class String1
{
    public static void main(String args[])
    {
        String s="Sachin";
        s1= s.concat(" Tendulkar");
        System.out.println(s1);
    }
}
```

Output: SachinTendulkar

Reason:

- Suppose there are **5 reference variables**, all **refers to one object "sachin"**.
- If **one reference variable changes** the value of the object, **it will be affected to all the reference variables**.
- **That is why string objects** are **immutable** in java.
- **For mutable strings**, we can **use String Buffer and StringBuilder classes** are used.

Methods of String Class

Method Name	Description	Example
int length()	returns string length	String s1="java"; System.out.println("string length is "+s1.length()); //4 System.out.println("string length is: "+"java".length()); //4
char charAt(int index)	Extracts a single character from the string. Returns char value for the particular index.	String str="welcome"; char ch=str.charAt(3); //c
String substring(int beginIndex) String substring(int beginIndex, int endIndex)	Used to extract a substring from specified index from the main string. Returns a substring.	String s1="welcome"; String str1= s1.substring(3); //come String str2 = s1.substring(3,5); //com
String toLowerCase()	Returns a string in lowercase.	String s1="OBJECT oreinTED"; String str=s1.toLowerCase(); System.out.println(str); //object oreinted
String toUpperCase()	Returns a string in uppercase.	String s1="OBJECT oreinTED"; String str=s1.toUpperCase(); System.out.println(str); //OBJECT ORIENTED

Methods of String Class



Method Name	Description	Example
getChars (int start, int end, char target[], int tarstart)	Used to extract more than one character at a time.	Char buf[] = new char[10]; String str= "Welcome to Hyderabad"; str.getChars(3,6,buf,0) //come
boolean equals(Object str)	Returns true if the strings contain same characters in same order otherwise returns false.	String s1="hello"; String s2="hello"; String s3="HELLO"; System.out.println(s1.equals(s2)); //true System.out.println(s1.equals(s3)); //false
boolean equalsIgnoreCase(String str)	Compares two strings for equality and ignores the case differences.	boolean x=s1.equalsIgnoreCase(s3)); //true
boolean startsWith()	Returns true, if the given string begins with a specified string.	System.out.println("welcome".startsWith("wel")); //true String str = new String("Welcome"); System.out.println(str.startsWith("come")); //false
boolean endsWith()	Returns true, if the given string ends with a specified string.	String str = new String("Welcome"); System.out.println(str.endsWith("come")); //true

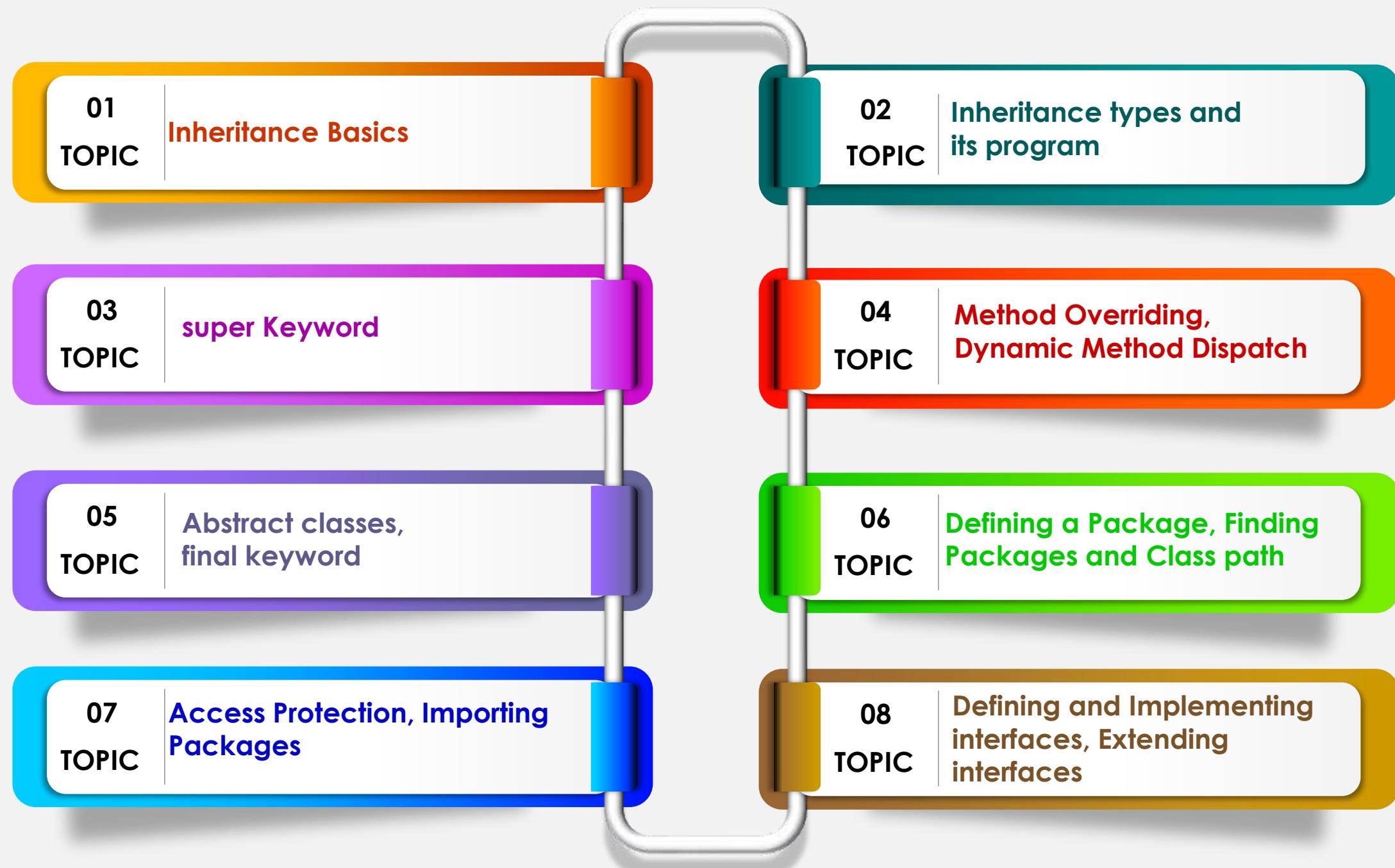
Methods of String Class

Method Name	Description	Example
String concat(String str)	The two strings are concatenated.	<code>String s1 = "Welcome"; String s2 = s1.concat("To India");</code>
String replace(char old , char new)	Replaces all occurrences of one character in the invoking string with another character.	<code>String s1 = "Welcome"; String s2 = s1.replace('e' , 'o');</code>
String trim()	Returns a copy of invoking string from which any leading and trailing white space has been removed.	<code>String s1 = " Welcome "; String s2 = s1.trim(); // "welcome"</code>
boolean isEmpty()	Returns true if the string object is empty	<code>String s=""; boolean x = s.isEmpty(); //true</code>
String[] split(String exp)	Returns a split string matching regular expression.	<code>String str="Welcome to Hyd for biryani"; String s[] = str.split(" "); //splits string with white spaces.</code>

Methods of String Class

Method Name	Description	Example
int compareTo(String str)	Used to compare two strings. It returns < 0: The invoking string is less than the given string. >0: The invoking string is greater than the given string. 0: if the two strings are equal.	<pre>String s1 ="welcome"; String s2= "hello"; String s3 = "welcome"; System.out.println(s1.compareTo(s3)); //0 System.out.println(s1.compareTo(s2)); //>0 System.out.println(s2.compareTo(s3)); //<0</pre>
Searching Strings int indexOf(int ch) int indexOf(String str) int indexOf(int ch , int start)	Used to search a string for a specified character or substring. Returns index at which the character or string is found , Otherwise returns -1.	<pre>String str = "to do welcome to indian railways to come and well do of the rail works" int x = str.indexOf('w'); int x = str.indexOf("wel"); int x = str.indexOf('w' , 10);</pre>





Inheritance

- Inheritance in java is a mechanism in which **one object acquires the properties and behaviors of another object.**
- The **idea behind inheritance** in java is that we can **create new classes that are built upon existing classes.**
- When we **inherit from an existing class**, we can **reuse methods & data of parent class;**
- Inheritance represents the **IS-A relationship**, also known as **parent-child relationship.**
- Need of inheritance in java
 - i. **For Method overriding (so runtime Polymorphism can be achieved).**
 - ii. **For Code Reusability**
 - iii. **It allows creation of Hierarchical Classification.**

Syntax of Inheritance

```
class Subclass-name extends Superclass-name  
{  
    //methods and variables of sub class  
}
```

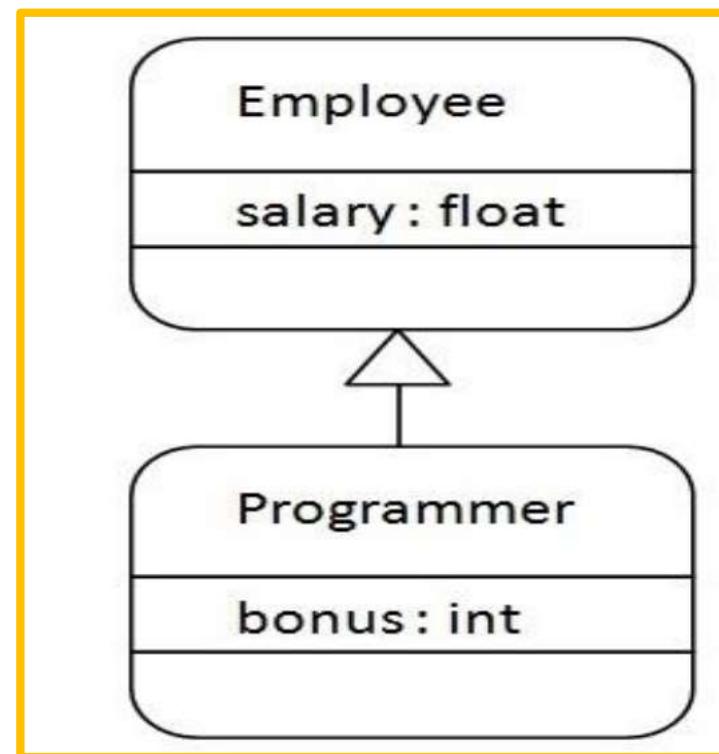
- The **extends keyword indicates** that we are making a **new class that derives from an existing class**.
- A **class which is inherited** is called **parent or super class**.
- The **class that does the inheriting** is called **child or subclass**.
- A **subclass is a specialized version of a super class**.

Example:

- ✓ A Student Is-A Person
- ✓ Programmer Is-A Employee

Example

- Here **Programmer** is the **subclass** and **Employee** is the **super class**.
- **Relationship** between **two classes** is Programmer **IS-A** Employee.
- It means that **Programmer** is a type of **Employee**.



Example

```
class Employee
{
    float salary=40000;
}

class Programmer extends Employee
{
    float bonus=(0.2)*salary;
}

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

OUTPUT

Programmer salary is: 40000.0
Bonus of programmer is: 10000

In the above example,

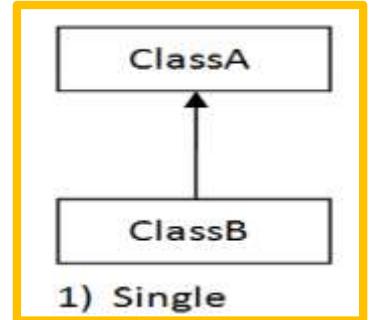
- ✓ Programmer object can access the field of own class as well as of Employee class

Types of inheritance

On the basis of class, there can be **three types of inheritance** in java:

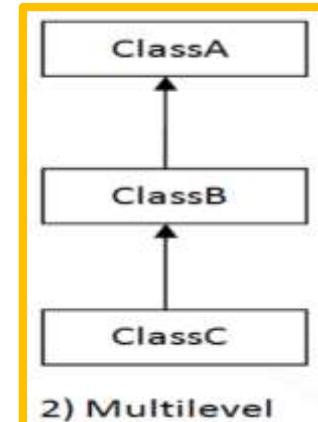
i.Single Level Inheritance:

A **super class is inherited by only one sub class.**



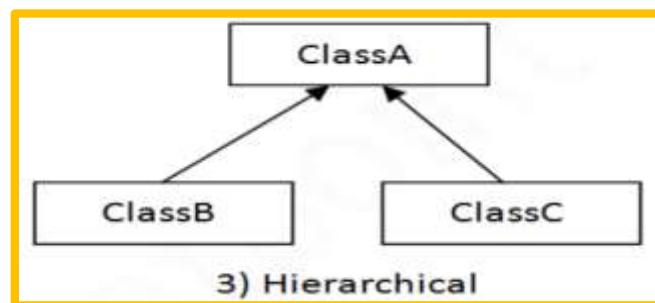
ii.Multi Level Inheritance:

A **sub-class will be inheriting parent class** and as well as, the **sub-class act as a parent class to other class.**, and so on.



iii.Hierarchical Inheritance:

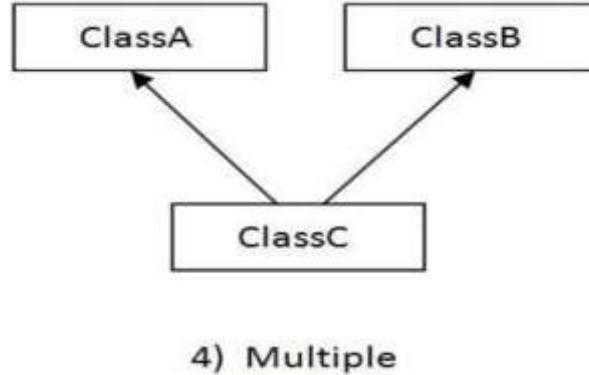
A **Super class is inherited by many sub classes.**



Types of inheritance

iv. Multiple Inheritance:

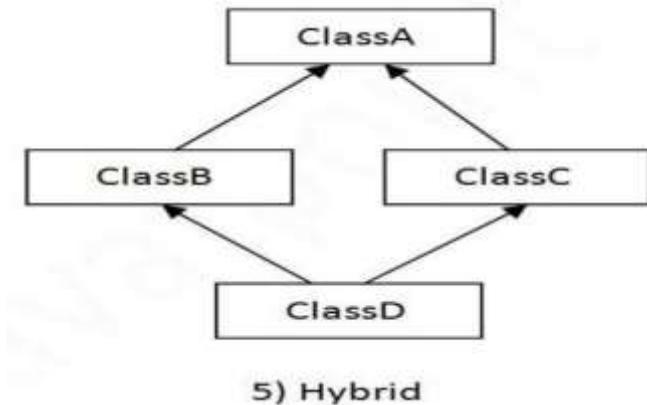
A sub class extending more than one super class.



4) Multiple

v. Hybrid Inheritance:

It is a combination of Multiple and Multilevel inheritance.



5) Hybrid

Note:

- Java does not support Multiple Inheritance and Hybrid Inheritance directly with classes.
- In Java multiple and hybrid inheritance is supported through interfaces only.

i.e : class C extends A,B

{

//ComplieTimeError

}

//Example program for Single Level Inheritance

```
class Bird
{
    void fly()
    {
        System.out.println("I am a Bird");
    }
}

class Parrot extends Bird
{
    void colour()
    {
        System.out.println("I am green!");
    }
}

class Test
{
    public static void main(String args[])
    {
        Parrot obj = new Parrot();
        obj.colour();
        obj.fly();
    }
}
```

OUTPUT:

I am green!
I am a Bird

//Program to Demonstrate Single Level Inheritance

```
class A
{
    void showA()
    {
        System.out.println("show method of ClassA");
    }
}

class B extends A
{
    void showB()
    {
        System.out.println("show method of ClassB");
    }
}

public class SingleLevel
{
    public static void main(String args[])
    {
        B b = new B();
        b.showA();
        b.showB();
    }
}
```

OUTPUT:

show method of ClassA
show method of ClassB

//Program for Multi Level Inheritance

```
class Bird
{
    void fly()
    {
        System.out.println("I am a Bird");
    }
}

class Parrot extends Bird
{
    void colour()
    {
        System.out.println("I am green!");
    }
}

class SingingParrot extends Parrot
{
    void sing()
    {
        System.out.println("I can sing");
    }
}
```

class Test

```
{}
public static void main(String args[])
{
    SingingParrot obj = new SingingParrot();
    obj.sing();
    obj.colour();
    obj.fly();
}
```

OUTPUT:

```
I can sing
I am green!
I am a Bird
```

//Program for Multi Level Inheritance

```
class A
{
    int i=10;
    void showA()
    {
        System.out.println(" show() method of A"+i);
    }
}

class B extends A
{
    int j=25;
    void showB()
    {
        System.out.println(" show() method of B"+j);
    }
}

class C extends B
{
    int k=20;
    public void showC()
    {
        System.out.println(" show() method of C"+k);
        System.out.println("The members sum is " + (i + j +k));
    }
}
```

```
class MultiLevel
{
    public static void main(String args[])
    {
        C c = new C();
        c.showA();
        c.showB();
        c.showC();
    }
}
```

OUTPUT:

```
show() method of A 10
show() method of B 25
show() method of C 20
The members sum is 55
```

//Program for Hierarchical Inheritance

```
class Bird
{
    void fly()
    {
        System.out.println("I am a Bird");
    }
}

class Parrot extends Bird
{
    void colour()
    {
        System.out.println("I am green!");
    }
}

class Crow extends Bird
{
    void colour()
    {
        System.out.println("I am black!");
    }
}
```

```
class Test
{
    public static void main(String args[])
    {
        Parrot p = new Parrot();
        Crow c = new Crow();
        p.colour();
        p.fly();
        c.colour();
        c.fly();
    }
}
```

OUTPUT:

```
I am green!
I am a Bird
I am black!
I am a Bird
```

//Program for Hierarchical Inheritance

```
class A
{
    int i=10;
    void showA()
    {
        System.out.println(" show() method of A"+i);
    }
}

class B extends A
{
    int j=25;
    void showB()
    {
        System.out.println(" show() method of B"+j);
    }
}
```

```
class C extends A
{
    int j=99;
    void showC()
    {
        System.out.println(" show() method of C"+j);
    }
}

class HierLevel
{
    public static void main(String args[])
    {
        B b = new B();
        b.showA();
        b.showB();
        C c = new C();
        c.showA();
        c.showC();
    }
}
```

OUTPUT:

```
show() method of A 10
show() method of B 25
show() method of A 10
show() method of C 99
```

super keyword

- The purpose of “super” keyword is
 - i. To call superclass methods and constructor from subclass
 - ii. If a subclass has the same member as super class, the subclass hides the members of super class. To access the super class member in subclass uses super keyword.
- ✓ super keyword is to eliminate the confusion between superclasses and subclasses that have methods with the same name.

Syntax

super (arguments list);

Syntax

super.member or super.method()

Example

super.i;
super.show();

iii.super can be used to refer immediate super class instance variable, method and constructor.

- super must be the first statement in the sub class constructor.
- super () is provided by compiler implicitly as default constructor if no constructor is specified explicitly.

// *super()* is provided by the compiler implicitly

```
class Animal
{
    Animal()
    {
        System.out.println("animal is created");
    }
}

class Dog extends Animal
{
    Dog()
    {
        // super();      calls automatically.
        System.out.println("dog is created");
    }
}

class Test
{
    public static void main(String args[])
    {
        Dog d=new Dog();
    }
}
```

Output

animal is created
dog is Created

// Program to demonstrate order of constructor execution

```
class A
{
    A()
    {
        System.out.println("Inside A's constructor");
    }
}

class B extends A
{
    B()
    {
        System.out.println("Inside B's constructor");
    }
}

class C extends B
{
    C()
    {
        System.out.println("Inside C's constructor");
    }
}
```

class SuperDemo4

```
{
    public static void main(String args[])
    {
        C c = new C();
    }
}
```

Output

Inside A's constructor
Inside B's constructor
Inside C's constructor

// Accessing data members (same name Variables) of super class

```
class Animal
{
    String name="This is Animal";
}

class Lion extends Animal
{
    String name="This is Lion";
    void display()
    {
        System.out.println(name);
        System.out.println(super.name);
    }
}

class Test
{
    public static void main(String[] args)
    {
        Lion L= new Lion();
        L.display();
    }
}
```

Output
This is Animal
This is Lion

// Accessing methods (same name methods) of super class

```
class Parent
{
    void show()
    {
        System.out.println("Parent Method");
    }
}

class Child extends Parent
{
    void show()
    {
        super.show();
        System.out.println("Child Method");
    }
}

class Test
{
    public static void main(String[] args)
    {
        Child c = new Child();
        c.show();
    }
}
```

Output
Parent Method
Child Method

Method Overriding

- If **sub class** has the **same method name and signature as a method in its super class**, it is known as **method overriding** in Java.
- In **that case sub class** will **hide** the **super class method**.
- Rules for Method Overriding
 - i. **Method in sub class** must have **same name and parameter signature as super class**.
 - ii. Must be **Is-A** relationship. (Inheritance).

Method Overriding Example

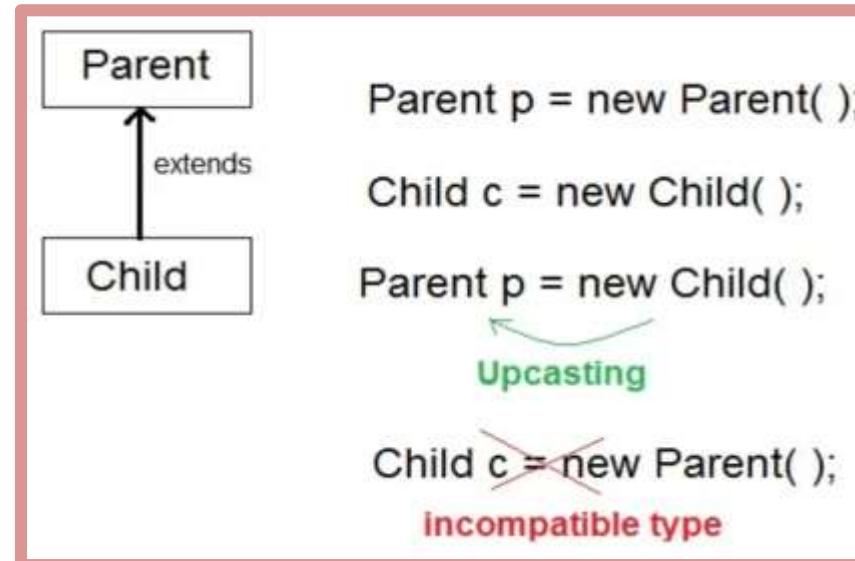
```
class A
{
    int i,j;
    A(int a , int b)
    {
        i=a;
        j=b;
    }
    void show()
    {
        System.out.println("Value of i is:"+i);
        System.out.println(("Value of i is:"+j));
    }
}
class B extends A
{
    int k;
    B(int a , int b , int c)
    {
        super(a,b);
        k=c;
    }
}
```

```
void show()
{
    super.show();
    System.out.println ("Value of k is:"+k);
}

class Test
{
    public static void main(String args[])
    {
        B b = new B(10,20,30);
        b.show();
    }
}
```

Runtime Polymorphism (Dynamic Method Dispatch)

- Using **Dynamic Method Dispatch**, a **call** to an **overridden method** is **resolved** at **runtime**, rather than at **compile time**.
- Runtime polymorphism** in java **implemented** by using **Dynamic method dispatch**.
- Upcasting:** A **super class reference variable** can **refer** to a **sub class object**. I.e. When reference variable of super class refers to the sub class object, then it is called **up casting**.



// Dynamic Method Dispatch – Example1

```
class Game
{
    void type()
    {
        System.out.println("Indoor and outdoor");
    }
}

Class Cricket extends Game
{
    void type()
    {
        System.out.println("It is outdoor game");
    }
}
```

Class Test

```
{  
public static void main(String[] args)  
{  
    Game gm = new Game();  
    Cricket ck = new Cricket();  
    gm.type();  
    ck.type();  
    gm = ck;          //gm refers to Cricket object  
    gm.type();         //calls Cricket's version of type  
}}
```

Output

```
Indoor and outdoor  
It is outdoor game  
It is outdoor game
```

```
class Bank
{
    float Interest()
    {
        return 0;
    }
}

class SBI extends Bank
{
    float Interest()
    {
        return 8.4f;
    }
}

class ICICI extends Bank
{
    float Interest()
    {
        return 7.3f;
    }
}
```

```
class AXIS extends Bank
{
    float Interest()
    {
        return 9.7f;
    }
}

class Test
{
    public static void main(String args[])
    {
        Bank b;
        b=new SBI();
        System.out.println("SBI Rate of Interest: "+b.Interest());
        b=new ICICI();
        System.out.println("ICICI Rate of Interest: "+b.Interest());
        b=new AXIS();
        System.out.println("AXIS Rate of Interest: "+b. Interest());
    }
}
```

Differentiate method overloading and method overriding

Overloading	Overriding
Must have at least two methods by same name in same class.	Must have at least one methods by same name in both Parent and child class.
Must have different number of parameters.	Must have same number of Parameters.
If number of parameters are same then must have different types of parameters.	Types of parameters also must be same.
Overloading known as compile time polymorphism.	Overriding knows as run time polymorphism.

Abstract classes

- Abstraction is a **process of hiding implementation details** and **showing only functionality** to the **users**.

Eg: Sending a Message, Driving a car etc.

- Abstraction in java achieved using :

- ✓ **Abstract classes (0 to 99.9%)**
- ✓ **Interfaces (100%)**

Abstract classes

- Creating a super class that **only defines** a **method declaration** that will be **shared by all of its sub classes**; the **sub classes** must **provide implementation**.
- The **method which is implements** by the **sub class must be specified as “abstract”**.
- An **abstract method** has **no** method **body**
- Any class that **contains one or more abstract methods**, then the **class is defines** as **abstract class**.

Abstract classes

- An **abstract class** is **non-concrete class**.
- **Concrete class** is a **class that has an implementation** for **all** of its **methods**. They **cannot have any unimplemented methods**. It is a **complete class** and **can be instantiated**.
- **Non-Concrete class** is a class that **has no implementation(no body)** for **some of the methods** defined in it. It is an **incomplete class** .

Abstract method Syntax

```
abstract returnType  
methodName(arguments);
```

Abstract class Syntax

```
abstract class class_name  
{  
    abstractMethod();  
    normalMethod()  
    {  
        #body of the method  
    }  
}
```

Rules for abstract modifier

- All **abstract methods** of super class must be **implemented** in **sub class** by **overriding**.
- If any **abstract method** is **not implemented** in **sub class**, then the **sub class** is also **made as abstract class**.
- **No objects** of an **abstract class** **exist**. i.e. An **abstract class cannot be directly instantiated** with the **new operator**.
- An **abstract class not fully defined**. Hence **object cannot be created**.
- **Abstract classes** can be **used to create object references** because of **Dynamic Method Dispatch** and **Run time polymorphism**

```
abstract class Bigboss
{
    abstract void season();
    abstract void winner();
    void runner()
    {
        System.out.println("The runner is: Srihan");
    }
}

class starmaa extends Bigboss
{
    void season()
    {
        System.out.println("This is BiggBoss Season 6");
    }
    void winner()
    {
        System.out.println("The winner is: Revanth");
    }
}
```

```
class BB6
{
    public static void main(String args[])
    {
        starmaa s = new starmaa();
        s.season();
        s.winner();
        s.runner();
    }
}
```

Output
This is BiggBoss Season 6
The winner is: Revanth
The runner is: Srihan

Abstract classes

```
abstract class Bike
{
    abstract void run();
    void engine()
    {
        System.out.println("Bike engine");
    }
}
class Honda extends Bike
{
    void run()
    {
        System.out.println("Running safely");
    }
}
```

```
Class Test
{
    public static void main(String args[])
    {
        Bike B = new Honda();
        B.run();
        B.engine();
    }
}
```

Output
Running safely
Bike engine

Abstract classes

```
abstract class Calculator  
{  
    abstract void display();  
}
```

```
class Add extends Calculator  
{  
    void display()  
    {  
        System.out.println("This is Addition class");  
    }  
}
```

```
class Sub extends Calculator  
{  
    void display()  
    {  
        System.out.println("This is Subtraction Class");  
    }  
}  
class Test  
{  
    public static void main(String arg[])  
    {  
        Calculator c1 = new Add();  
        c1.display();  
        Calculator c2 = new Sub();  
        c2.display();  
    }  
}
```

Output

This is Addition class
This is Subtraction Class

Abstract classes

```
abstract class Shape  
{  
    abstract void draw();  
}  
class Rectangle extends Shape  
{  
    void draw()  
    {  
        System.out.println("drawing rectangle");  
    }  
}  
class Circle extends Shape  
{  
    void draw()  
    {  
        System.out.println("drawing circle");  
    }  
}
```

```
class Test  
{  
    public static void main(String args[])  
    {  
        Shape s1=new Circle();  
        s1.draw();  
        Shape s2=new Rectangle();  
        s2.draw();  
    }  
}
```

Output
drawing circle
drawing rectangle

Rules for abstract class

Rules for Java Abstract class



1

An abstract class must be declared with an abstract keyword.

2

It can have abstract and non-abstract methods.

3

It cannot be instantiated.

4

It can have final methods

5

It can have constructors and static methods also.

final keyword

- The **final keyword** in java is used to **restrict the user**.
- final is used with a **variable, method , or a class**.

i.final keyword with a variable

- A **variable declared as final prevents its contents from being modified.**(Constant)
- We **must initialize a final variable when it is declared.**
- The **value** can be **initialized during declaration** or in the **constructor.**
- A **local variable** or parameter **can also be made as “final”**

```
final double PI = 3.1429;
```

```
PI = PI + 2; //CTE
```

```
//Example to demonstrate final variable
class test
{
    public static void main(String[] args)
    {
        final int AGE = 22;
        AGE = 25;
        System.out.println("Age: " + AGE);
    }
}
```

final keyword with a method

```
class Bike
{
    final void show()
    {
        System.out.println("Bike Method");
    }
}

class Honda extends Bike
{
    //inherits show()
    void show()
    {
        System.out.println("Honda's Show");
    }
    void display()
    {
        System.out.println("Honda's Show");
    }
}
```

A **method declared as final**, will **not allow overriding**. i.e.

It prevents overriding / stops overriding.

A **constructor cannot be declared as final**. Because it is never inherited.

A **final method is inherited, but not overridden**.

```
class Test
{
    public static void main(String args[])
    {
        Honda h1 = new Honda();
        h1.display();
        h1.show();      //compile time error
    }
}
```

final keyword with a class

- A class declared as final will not allow extending. i.e. It stops inheritance.

```
final class Bike
{
    final void show()
    {
        System.out.println("Bike's Method");
    }
}

class Honda extends Bike //Compile time error
{
    //trying to inherit - CTE
}
```

Conclusion of Final keyword

- ✓ Stop Value Change
- ✓ Stop Method overriding
- ✓ Stop inheritance

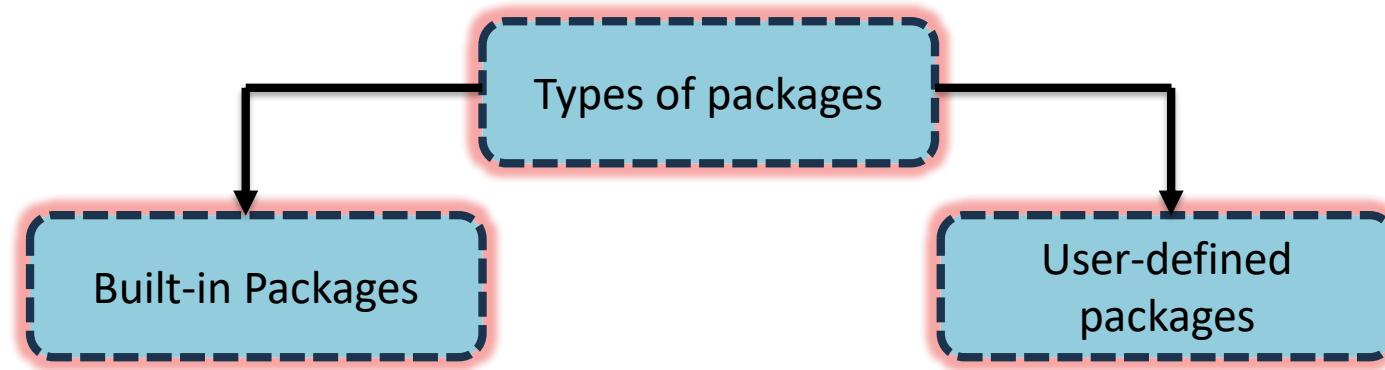
Is final method inherited?

- If we declare a parameter as final, it cannot allow changing in its method.

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

Packages

- Package in Java is a **mechanism to encapsulate** a **group of classes, sub packages** and **interfaces**. **Packages** are **containers** for **classes**.
- Packages are **stored in hierarchical manner** and are **explicitly imported** into **new classes**.
- Package in java can be **categorized in two form**:



1. Built-in Packages :

- ✓ These **packages consist** of a **large number of classes** which are a **part of Java API**. Some of the **commonly used built-in packages** are:

- i. **java.lang:** Contains **language support classes** (Integer, Throwable, Threadetc). This package is automatically imported.

Introduction to Packages

ii. **java.io:** Contains classes for **supporting input / output operations.**

iii. **java.util:** **Contains** utility classes which implement data structures like **Scanner, Collections, Date, StringTokenizer** etc.

2. User-defined packages

- ✓ These are the **packages that are defined by the user.**

Advantages of Java Packages

- ✓ We can **reuse existing classes** from the packages **as many times as we need it in our program.**
- ✓ Java Package **provides access protection.**
- ✓ Java Packages **removes naming collision.**
- ✓ Packages can be **considered as data encapsulation.**
- ✓ Packages can be **organized** as **hierarchical structures** with **sub packages and classes.**

Defining a Package

- The “**package**” keyword is used to **create a package** in **java**.
- A package is defined using “**package**” command as the **first statement** in the **java source file**.
- The “**package**” statement defines a **name space in which** the **classes** are **stored**.
- The general form of package is :

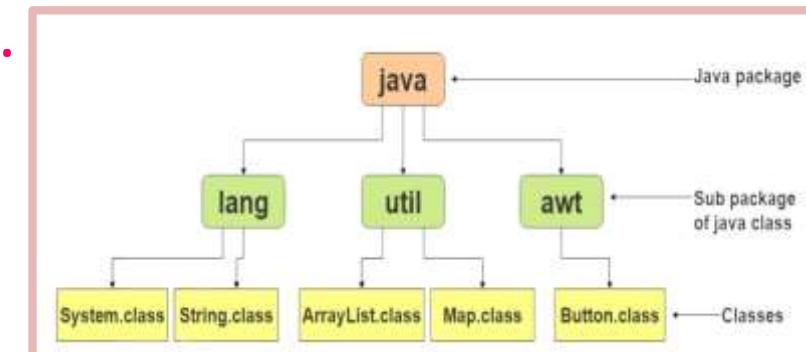
Syntax

package pkgname;

Example

package mypack;

- Java **uses file system directories to store the packages**. i.e. **All .class** files must be **kept** in a **directory** called **mypad**.
- **More than one file** can include the **same package statement**.
- We can create a **hierarchy of packages**. Thus forms **multilevel package**.
- The “**.” Operator** is **used** to create **multilevel packages**.
Eg: package com.vce.cse
- The **directory structure** is **com/vce/cse** to store “.class” files.



Importing Packages

- The “**import**” statement in java allows **accessing a package**.
- The form of “import” statement is:

Syntax-1

```
import pkg1.pkg2. .... pkgn;
```

Example-1

```
import java.util.Scanner;
```

Syntax-2

```
import pkg1.*;
```

Example-2

```
import java.util.*;
```

Note:After creation of Package program,we need compile it by using below syntax:

Syntax-1

```
>javac -d . Classname.java
```

Example

```
>javac -d . Animal.java
```

–d is used to save the class file in the directory and the ‘.’ (dot) denotes the package in the current directory. To avoid name conflicts, please use lower case for package names.

Example-1

//Program to Demonstrate package

```
package Animals;
public class PetAnimals
{
    public void Dog()
    {
        System.out.println("This is Dog");
    }
    public void Cat()
    {
        System.out.println("This is Cat");
    }
}
```

Note:

'PetAnimals .class' file is stored in' Animals' folder.

```
package Animals;
public class WildAnimals
{
    public void Lion()
    {
        System.out.println("This is Lion");
    }
    public void Tiger()
    {
        System.out.println("This is Tiger");
    }
}
```

Note:

WildAnimals.class file is stored in Animals folder.

Example-1

//Program for Accessing Package

```
import Animals.PetAnimals;
import Animals.WildAnimals;
class Test
{
    public static void main(String args[])
    {
        PetAnimals p=new PetAnimals();
        p.Dog();
        p.Cat();
        WildAnimals w=new WildAnimals();
        w.Lion();
        w.Tiger();
    }
}
```

//Program to Demonstrate package

```
package mypack;  
public class Calculator  
{  
    public int add(int a , int b)  
    {  
        return(a+b);  
    }  
    public int sub(int a , int b)  
    {  
        return(a-b);  
    }  
    public int mul(int a , int b)  
    {  
        return(a*b);  
    }  
    public int div(int a , int b)  
    {  
        return(a/b);  
    }  
}
```

Calculator.class file is stored in mypack folder.

Example-2

```
package mypack;  
public class Factorial  
{  
    int n;  
    public Factorial(int n)  
    {  
        this.n = n;  
    }  
    public int fact()  
    {  
        int f=1;  
        for(int i=1;i<=n;i++)  
            f=f*i;  
        return f;  
    }  
}
```

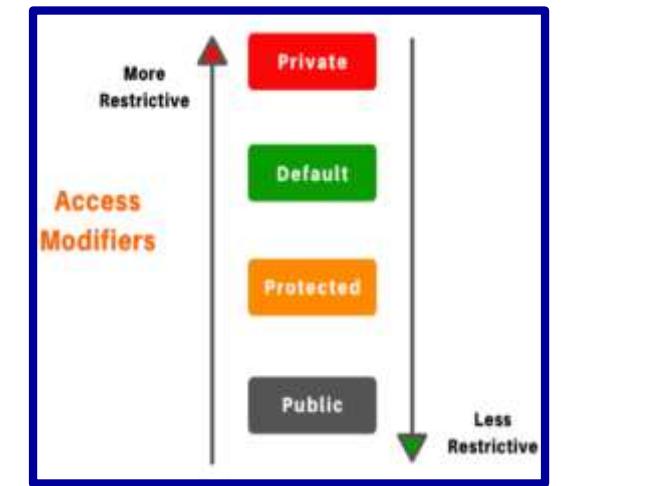
Factorial.class file is stored in mypack folder.

Example-2

```
//Accessing Package
import mypack.Calculator;
import mypack.Factorial;
public class PkgDemo
{
    public static void main(String args[])
    {
        Calculator c = new Calculator();
        int x=10;
        int y=20;
        System.out.println("Addition" + c.add(x,y));
        System.out.println("Multiplication" + c.mul(x,y));
        Factorial f = new Factorial(6);
        int res = f.fact();
        System.out.println("The factorial is " +res);
    }
}
```

Access Modifiers In Java

- These are **powerful tools** help you determine **who can use or modify different parts of your code**, helping to **keep your projects organized** and **secure**.
 - **Access modifiers** are **keywords** that can be **used to control the accessibility** of **fields, methods, and constructors in a class**.
 - The four access modifiers in Java are:
 - public, protected, default, and private**
- i. **public:** Anything declared **public** can be **accessed from anywhere**.
 - ii. **private:** Anything declared **private** cannot be seen outside of its class.
 - iii. **default:** When a member **does not have an explicit access specification**, it is **visible to subclasses** as well as to **other classes in the same package**. This is the default access.
 - iv. **protected:** The access level of a protected modifier is **within the package** and **outside the package through child class**. If you do not make the child class, it cannot be accessed from outside the package.



Access Modifiers In Java

Members of JAVA	Private	Default	Protected	Public
Class	No	Yes	No	Yes
Variable	Yes	Yes	Yes	Yes
Method	Yes	Yes	Yes	Yes
Constructor	Yes	Yes	Yes	Yes
interface	No	Yes	No	Yes

Visibility	Default	Public	Protected	Private
Same class	Yes	Yes	Yes	Yes
Class in same package	Yes	Yes	Yes	No
Subclass in same package	Yes	Yes	Yes	No
Subclass outside the same package	No	Yes	Yes	No
Non-subclass outside the same package	No	Yes	No	No

//Accessing a private data and method from same class

```
class college
{
    private String Name="Vardhaman";
    private void show()
    {
        System.out.println("I am private function");
    }
    public static void main(String[] args)
    {
        college c=new college();
        c.show();
        System.out.println(c.Name);
    }
}
```

Output

```
I am private function  
Vardhaman
```

//Accessing a private data and method from different class

```
class college
{
    private String Name="Vardhaman";
    private void show()
    {
        System.out.println("I am private function");
    }
}
class Test
{
    public static void main(String[] args)
    {
        college c=new college();
        c.show();
        System.out.println(c.Name);
    }
}
```

Output

```
Compile Time Error
```

```
//Accessing a protected data and methods
class A
{
    protected void display()
    {
        System.out.println("I am protected function");
    }
}
class B extends A {}
class C extends B {}

class Test
{
    public static void main(String args[])
    {
        B obj1 = new B();
        obj1.display();
        C obj2 = new C();
        obj2.display ();
    }
}
```

Output

I am protected function
I am protected function

Interfaces

- The **keyword ‘interface’ used to defining** an **interface**.
- An Interface **allows creating** a **fully abstract class**.
- An **interface specifies** what a **class must do** but **not how it does it**.
- An Interface has **methods declared without any body** and it can have **Final and static variables**.
- Once an interface is defined, any number of classes can implement an interface.
- The **class which implements** an **interface must provide definition** for all the methods of an **interface**.
- It is **used to achieve abstraction** and **multiple inheritance in Java**.
- Java **Interface** also **represents IS-A relationship**.
- It **cannot be instantiated** just like **abstract class**.
- To **implement** an **interface**, use “**implements**” clause in the **class definition** .
- All methods are **implicitly public** and **abstract**.
- All variables are **implicitly final and static**, must be initialized.

Syntax for Defining Interface

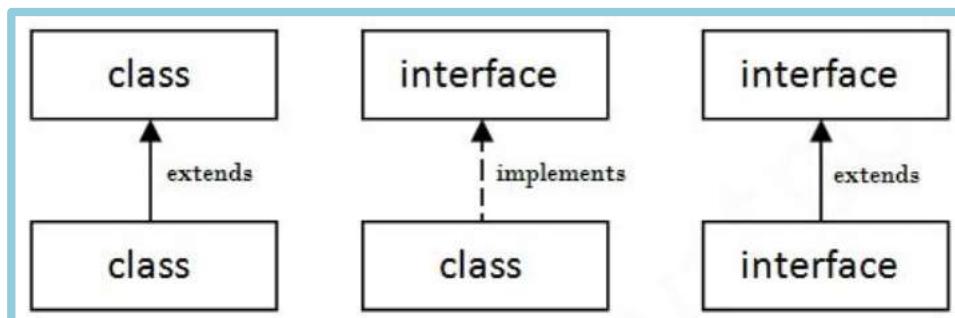
interface InterfaceName

{

 returntype methodName1(Arguments);
 returntype methodName2(Arguments);

 Datatype variable1 = value;
 Datatype variable1 = value;

}



Syntax for implementation of Interface

Class className implements interface1 , interface2

{

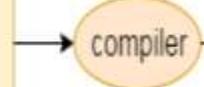
 class body

}

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

Printable.java

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



Printable.class

//Program for Interface

interface Language

```
{  
    void Name();  
    void type();  
    void version();  
}
```

class Proglang implements Language

```
{  
    public void Name()  
    {  
        System.out.println("I am JAVA");  
    }  
    public void type()  
    {  
        System.out.println("I am Programming Language");  
    }  
    public void version()  
    {  
        System.out.println("My Latest version 20.0");  
    }  
}
```

class Test

```
{  
    public static void main(String args[])  
    {  
        Proglang j = new Proglang();  
        j.Name();  
        j.type();  
        j.version();  
    }  
}
```

Output

```
I am JAVA  
I am Programming Language  
My Latest version 20.0
```

//Program for Multiple inheritance

Interface PetAnimals

```
{  
    void Dog();  
    void Cat();  
}
```

Interface WildAnimals

```
{  
    void Lion();  
    void Tiger();  
}
```

class Animals implements PetAnimals,WildAnimals

```
{  
    void Dog()  
    { System.out.println("This is Dog"); }  
    void Cat()  
    { System.out.println(" This is Cat "); }  
    void Lion()  
    { System.out.println(" This is Lion"); }  
    void Tiger()  
    { System.out.println(" This is Tiger"); }  
}
```

class Test

```
{
```

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

```
{
```

```
    Animals A = new Animals();  
    A.Dog();  
    A.Cat();  
    A.Lion();  
    A.Tiger();  
}
```

Output

```
This is Dog  
This is Cat  
This is Lion  
This is Tiger
```

How to extend Interface?

Syntax for extending Interface

```
interface InterfaceName-1
{
    returntype methodName-1(Arguments);
    returntype methodName-n(Arguments);

}

interface InterfaceName-2 extends InterfaceName-1
{
    returntype methodName-1(Arguments);
    returntype methodName-m(Arguments);

}

Class implements InterfaceName-2
{
    -----
    -----
}
```

//Program for extending Interface

```
interface Books
{
    void Notebooks();
    void Textbooks();
}

Interface Storybooks extends Books
{
    void Title();
}

class Booktype implements Storybooks
{
    void Notebooks()
    {   System.out.println("I am Notebook");
    }
    void Textbooks()
    {   System.out.println("I am Textbook");
    }
    void Title()
    {
        System.out.println("I am Storybook");
    }
}
```

class Test

{

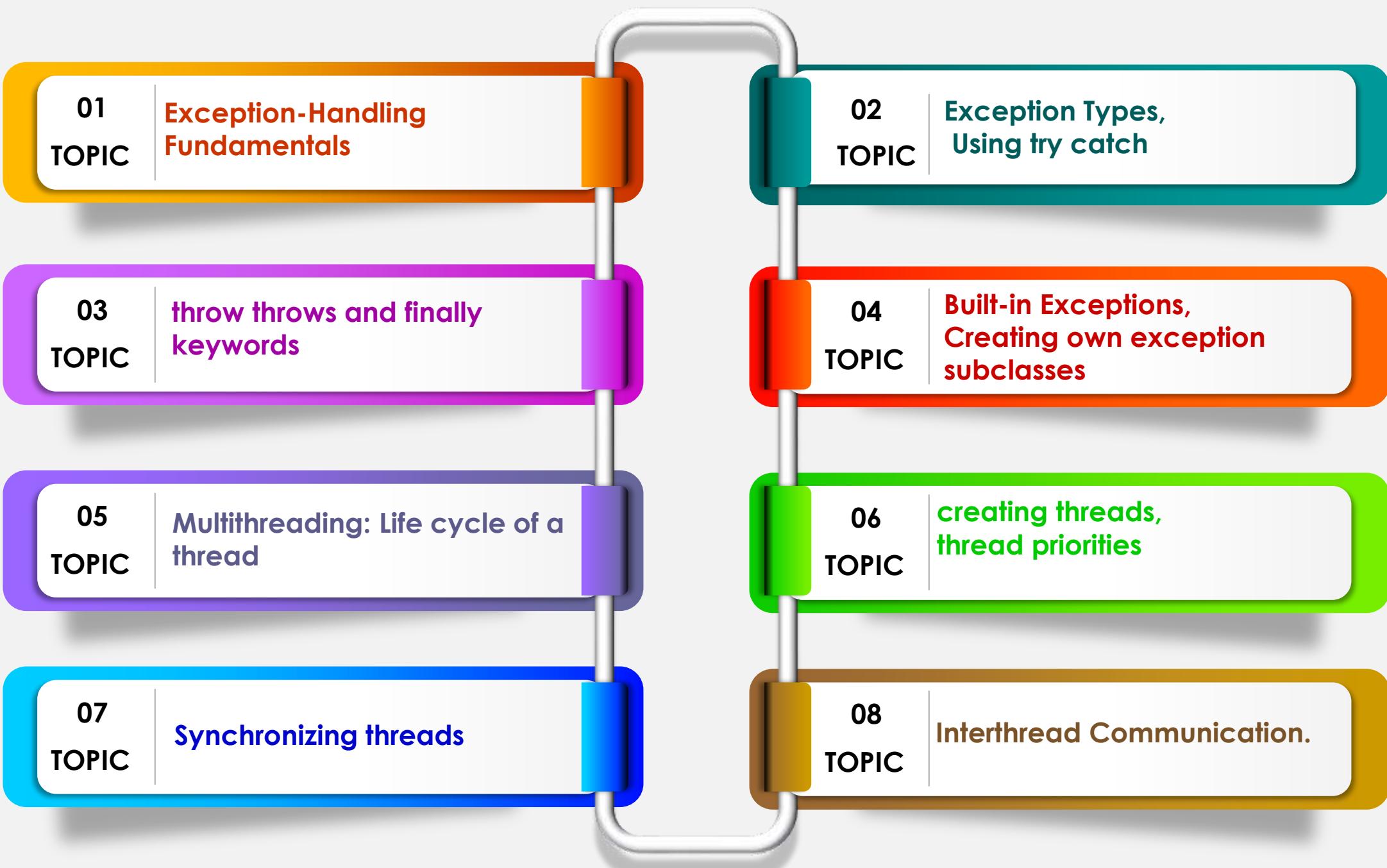
public static void main(String args[])
{

Booktypes B = new Booktypes();
 B. Notebooks();
 B. Textbooks();
 B. Title();
}

Output

I am Notebook
I am Textbook
I am Storybook







Exception-Handling Fundamentals

- An exception is an **abnormal condition that arises** in a **code at run time**. In other words, an exception is a **runtime error**.
- When an **Exception occurs the normal flow** of the program is **disrupted** and the **program/Application terminates abnormally**, which is **not recommended**, therefore these **exceptions are to be handled**.
- An exception can occur for **many different reasons**, below given are some scenarios where exception occurs.
 - i. Division by zero
 - ii. Array out of bound access exception
 - iii. A user has entered invalid data.
 - iv. A file that needs to be opened cannot be found.
 - v. A network connection has been lost in the middle of communications
- Some of **these exceptions** are **caused by user error**, others by **programmer error**, and others by **physical resources that have failed** in some manner.

Exception-Handling

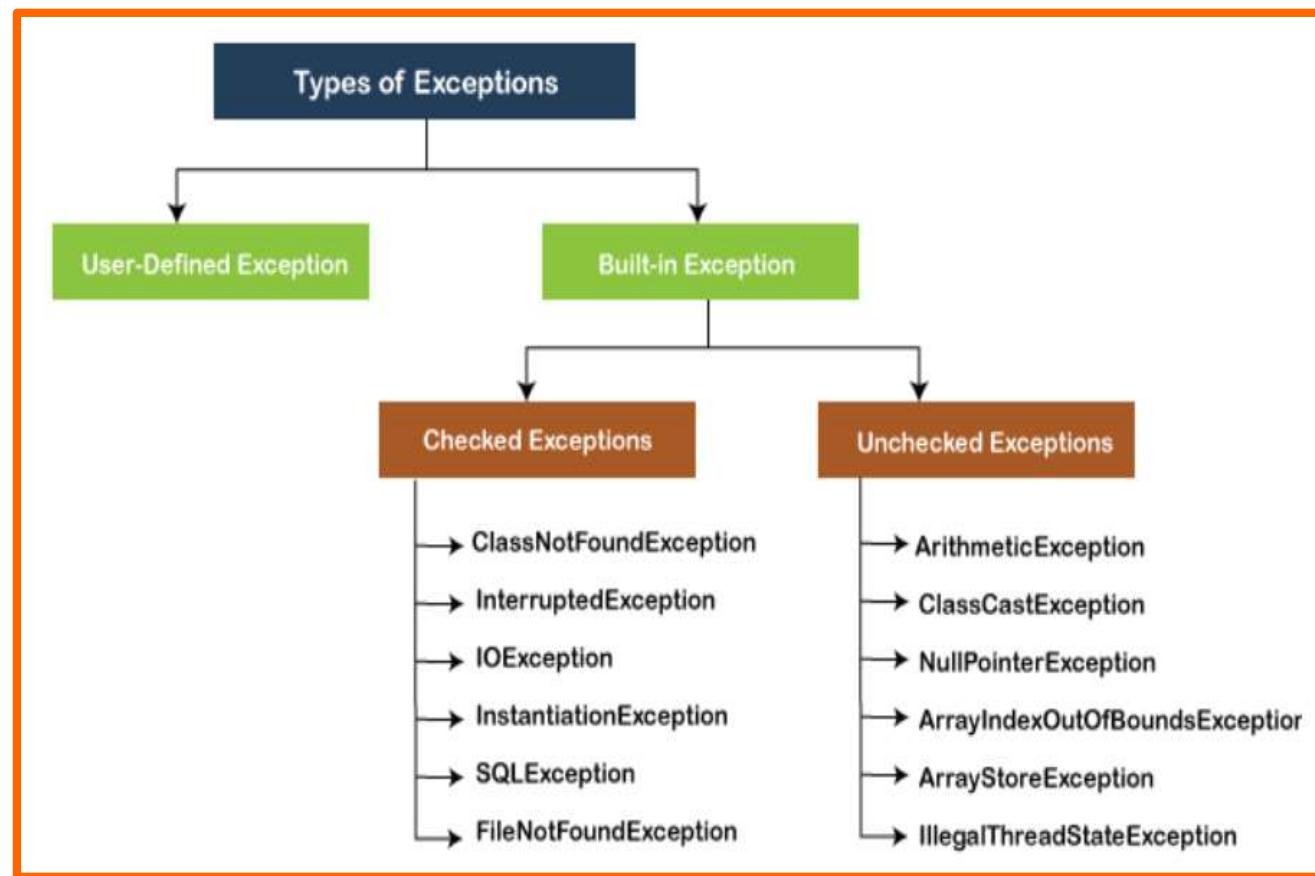
- “Exception handling is the **mechanism to handle run time errors**, so that the **normal flow of application** can be **maintained.**”
- Exception Handling is **managed by using 5 Keywords.**

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

Types of Java Exceptions

There are mainly **three types of exceptions:**

- i. **User Defined Exceptions**
- ii. **Built in Exceptions**
 - i. **Checked Exception**
 - ii. **Unchecked Exception**
- iii. **Error**



Types of Exceptions

i. Checked exceptions:

- A **checked exception** is an exception that **occurs at the compile time**, these are also called as **compile time exceptions**.
- These exceptions **cannot simply be ignored** at the time of compilation, the Programmer should take care of (handle) these exceptions.
- For example, if you **use FileReader class in your program to read data from a file**, if the **file specified** in its constructor **doesn't exist**, then an **FileNotFoundException** occurs, and compiler prompts the programmer to handle the exception.

i.Checked exceptions

1. ClassNotFoundException: This exception is thrown when the **JVM tries to load a class, which is not present** in the **classpath**.

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

2. FileNotFoundException: This exception is thrown when the **program tries to access a file that does not exist** or **does not open**. This error occurs mainly in file handling programs.

```
File file = new File("E:// file.txt");
FileReader fr = new FileReader(file);
```

3. IOException: This exception is thrown when the **input-output operation in a program fails** or is interrupted during the program's execution.

4. InterruptedException: This exception occurs **whenever a thread is processing, sleeping or waiting in a multithreading program** and **it is interrupted**.

```
Thread t = new Thread();
t.sleep(10000);
```

Types of Exceptions

```
import java.io.File;
import java.io.FileReader;
public class FilenotFound_Demo
{
    public static void main(String args[])
    {
        File file=new File("E://file.txt");
        FileReader fr = new FileReader(file);
    }
}
```

If you try to compile the above program you will get exceptions as shown below.

C:\>**javac FilenotFound_Demo.java**

FilenotFound_Demo.java:8: error: unreported exception **FileNotFoundException**; must be caught or declared to be thrown

FileReader fr = new FileReader(file);
^

1 error

Some of the examples of checked exceptions

Types of Exceptions

ii.Unchecked exceptions:

- An **Unchecked exception** is an exception that **occurs at the time of execution**, these are also called as **Runtime Exceptions**.
- These include **programming bugs**, such as **logic errors** or improper use of an API.
- Runtime exceptions are **ignored at the time of compilation**.
- For example, if you have **declared an array of size 5 in your program**, and **trying to call the 6th element of the array** then an **ArrayIndexOutOfBoundsException** occurs.
- To **guard against** and handle a **run-time error**, simply **enclose the code** that **you want to monitor** inside a **try block**.
- **Immediately** following the **try block**, **include a catch clause** that **specifies the exception type** that you wish to catch.

ii.Unchecked exceptions

1.ArithmaticException: This exception occurs when a **program encounters** an error in **arithmetic operations such as divide by zero.**

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println(120/0);
    }
}
```

OUTPUT

D:\>java Test
Exception in thread "main"
java.lang.ArithmaticException: / by zero

2.ArrayIndexOutOfBoundsException: This exception is thrown when an **array is accessed using an illegal index.** The index used is either **more than the size of the array** or is a **negative index.** Java doesn't support negative indexing.

```
class Test
{
    public static void main(String[] args)
    {
        int[] a = {10,20,30};
        System.out.println(a[50]);
    }
}
```

OUTPUT

D:\>java Test
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: Index
50 out of bounds for length 3

ii.Unchecked exceptions

3.NullPointerException: This exception is raised when a **null object is referred** to in a program.

NullPointerException is the most important and common exception in Java.

```
class Test
{
    public static void main(String[] args)
    {
        String s = null;
        System.out.println(s.length());
    }
}
```

OUTPUT

```
D:\>java Test
Exception in thread "main" java.lang.NullPointerException:
Cannot invoke "String.length()" because "<local1>" is null
at Test.main(Test.java:6)
```

4.NumberFormatException: This exception is thrown when a method could not convert a string into a numeric format.

```
class Test
{
    public static void main(String[] args)
    {
        String a = "abc";
        int num=Integer.parseInt(a);
    }
}
```



ii.Unchecked exceptions

5. StringIndexOutOfBoundsException: This exception is thrown by the string class, and it indicates that the index is beyond the size of the string object or is negative.

```
class Test
{
    public static void main(String[] args)
    {
        String a = "JAVA";
        char c = a.charAt(6); // accessing 6 index element
        System.out.println(c);
    }
}
```

ii.Unchecked exceptions

Exception	Meaning
ArithmaticException	Arithmatic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.
ReflectiveOperationException	Superclass of reflection-related exceptions.

Printing Exception information

The following are the methods to fetch exception information:

i. `obj.stackTrace()` -----> Name of the Exception: Description -> StackTrace

ii. `obj.toString()` -----> Name of the Exception: Description

iii. `obj.getMessage()` -----> Description

```
class Test
{
    public static void main(String[] args)
    {
        int a[] = {11,22};
        try
        {
            System.out.println(a[9]);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

`java.lang.ArrayIndexOutOfBoundsException: Index 9 out of bounds for length 2
at Test.main(Test.java:8)`

```
class Test
{
    public static void main(String[] args)
    {
        int a[] = {11,22};
        try
        {
            System.out.println(a[9]);
        }
        catch (Exception e)
        {
            System.out.println(e.toString());
        }
    }
}
```

`java.lang.ArrayIndexOutOfBoundsException: Index 9 out of bounds for length 2`

```
class Test
{
    public static void main(String[] args)
    {
        int a[] = {11,22};
        try
        {
            System.out.println(a[9]);
        }
        catch (Exception e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

`Index 9 out of bounds for length 2`

USING TRY AND CATCH

- This is the **general form** of an exception-handling block:

```
try
{
    -----// block of code to monitor for errors
}
catch (ExceptionType1 exOb)
{
    -----// exception handler for ExceptionType1
}
```

USING TRY AND CATCH

- **ExceptionType** is the **type of exception** that has **occurred**.
- Program **Statements that we monitor** for exceptions are **contained in try block**.
- If an **exception occurs in try block**, it is **thrown**.
 - ✓ The **code can catch this exception using catch block** and **handle the exception** in a rational manner.
 - ✓ **System generated exceptions** are **automatically thrown** by the Java **runtime**.
 - ✓ **Catch block** is used to handle the exception, called **Exception Handler**.
 - ✓ Must be **used after try block only**. We can use **multiple catch blocks with a single try block**.
- The **throw keyword** is **used to manually throw an exception**
- **Any code** that absolutely **must be executed after a try block** completes is **put in finally block**.

class Exc2

```
{  
    public static void main(String args[]){  
        int d, a;  
        try{  
            d = 0;  
            a = 42 / d;  
            System.out.println("This will not be printed.");  
        }  
        catch (ArithmeticException e){  
            System.out.println("Division by zero.");  
        }  
        System.out.println("After catch statement.");  
    }  
}
```

Output

Division by zero.
After catch statement

MULTIPLE CATCH CLAUSES

- In some cases, **more than one exception** could be **raised** by a **single piece of code**.
- **To handle this type of situation, you can specify two or more catch clauses**, each catching a different type of exception.
- When an **exception is thrown**, each **catch statement is inspected in order**, and the first one **whose type matches that** of the **exception** is **executed**.
- After **one catch statement executes**, the **others are bypassed**, and **execution continues after** the **try/catch block**.
- Syntax of Multiple catch statements:

```
try
{
    // block of code to monitor for errors
}
catch (ExceptionType1 exOb)
{
    // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{
    // exception handler for ExceptionType2
}
```

MULTIPLE CATCH CLAUSES

```
// Demonstrate multiple catch statements.
```

```
class MultiCatch {  
public static void main(String args[])  
{  
    try  
    {  
        int a = 10;  
        System.out.println("a = " + a);  
        int b = 42 / a;  
        int c[] = { 1 };  
        c[42] = 99;  
    }  
    catch(ArithmeticException e)  
    {  
        System.out.println("Divide by 0: " + e);  
    }  
    catch(ArrayIndexOutOfBoundsException e)  
    {  
        System.out.println("Array index oob: " + e);  
    }  
    System.out.println("After try and catch blocks.");  
}
```

Output

```
a = 10  
Array index oob:  
java.lang.ArrayIndexOutOfBoundsException:  
Index 42 out of bounds for length 1  
After try and catch blocks.
```

NESTED TRY STATEMENTS

- The **try statement can be nested**.
- That is, **a try statement** can be **inside** the block of **another try**.
- If an **inner try statement does not** have a **catch handler for a particular exception**, the stack is unwound and the **next try statement's catch handlers are inspected for a match**.
- This **continues until one of the catch statements succeeds**, or **until the entire nested try statements are exhausted**.
- If **no catch statement matches**, then the **Java run-time system will handle the exception**. Here is an example that uses nested try statements:

// An example of nested try statements.

```
class NestTry
{
public static void main(String args[])
{
    try
    {
        int a[] = {1,2,3,0,4};
        try
        {
            int b=a[2]/a[3];
        }
        catch(ArithmeticException e)
        {
            System.out.println( e);
        }
        a[20]=44;

    }
    catch(ArrayIndexOutOfBoundsException e)
    {
        System.out.println(e);
    }
}
```

OUTPUT:

```
C:\>java NestTry
java.lang.ArithmetricException: / by zero
java.lang.ArrayIndexOutOfBoundsException
```

THROW

- So far, **you have only been catching exceptions** that are **thrown by the Java run-time system.**
- However, it is **possible for your program to throw an exception explicitly**, using the **throw statement**
- **throw keyword** is used to **explicitly throw** an **exception from a method** or constructor.
- **We can throw** either **checked** or **unchecked exceptions** in java by throw keyword.
- The "**throw**" key-word is mainly **used to throw a custom exception**.
- When a **throw statement is encountered**, program **execution is halted**, and the **nearest catch statement is searched** for a **matching kind of exception**.
- The **general form** of throw is shown here:

Syntax
`throw ThrowabIeInstancE;`

Here, **ThrowabIeInstancE** must be an **object of type ThrowabIe** or a **subclasse of ThrowabIe**.

Example-1

```
throw new ArithmeticException( );
```

Example-2

```
throw new ArithmeticException("Something went wrong!");
```

```
// Demonstrate throw.  
class Test  
{  
    static void avg()  
    {  
        try  
        {  
            throw new ArithmeticException("demo");  
        }  
        catch(ArithmeticException e)  
        {  
            System.out.println("Exception caught"+e);  
        }  
    }  
  
    public static void main(String args[])  
    {  
        avg();  
    }  
}
```

Output

Exception caught java.lang.ArithmetiException: demo

```
import java.util.Scanner;
class Test
{
    public static void main(String[] args)
    {
        Scanner s = new Scanner(System.in);
        System.out.println("Please enter your roll number");
        int roll = s.nextInt();
        try
        {
            if (roll < 0)
            {
                throw new ArithmeticException("The number entered is not positive");
            }
            else
            {
                System.out.println("Valid roll number");
            }
        }
        catch (ArithmeticException e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

Output

Please enter your roll number
1201
Valid roll number

Output

Please enter your roll number
-27
The number entered is not positive

THROWS

- In Java, **Methods may throw exceptions during the execution** of the program **using** the **throws keyword**.
- The **throws keyword** is used to declare the **list of exception that a method may throw** during execution of program.
- so that **anyone calling that method gets** a **prior knowledge** about which **exceptions** are to be handled.
- This is the **general form** of a **method declaration that includes a throws clause**:

Syntax

```
<return_type> <method_name> ( ) throws <exception_name1>, <exception_name2>
{
    // body of method
}
```

Example

```
void Display( ) throws ArithmeticException, NullPointerException
{
    //code
}
```

```
class Test
{
    static void check() throws ArithmeticException
    {
        System.out.println("Inside check function");
        throw new ArithmeticException("demo");
    }

    public static void main(String args[])
    {
        try
        {
            check();
        }
        catch(ArithmetricException e)
        {
            System.out.println("caught" + e);
        }
    }
}
```

OUTPUT

Inside check function
caughtjava.lang.ArithmetricException: demo

DIFFERENCE BETWEEN THROW AND THROWS IN JAVA



throw	throws
throw keyword is used to throw an exception explicitly.	throws keyword is used to declare an exception possible during its execution.
throw keyword is followed by an instance of Throwable class or one of its sub-classes.	throws keyword is followed by one or more Exception class names separated by commas.
throw keyword is declared inside a method body.	throws keyword is used with method signature (method declaration).
We cannot throw multiple exceptions using throw keyword.	We can declare multiple exceptions (separated by commas) using throws keyword.

FINALLY

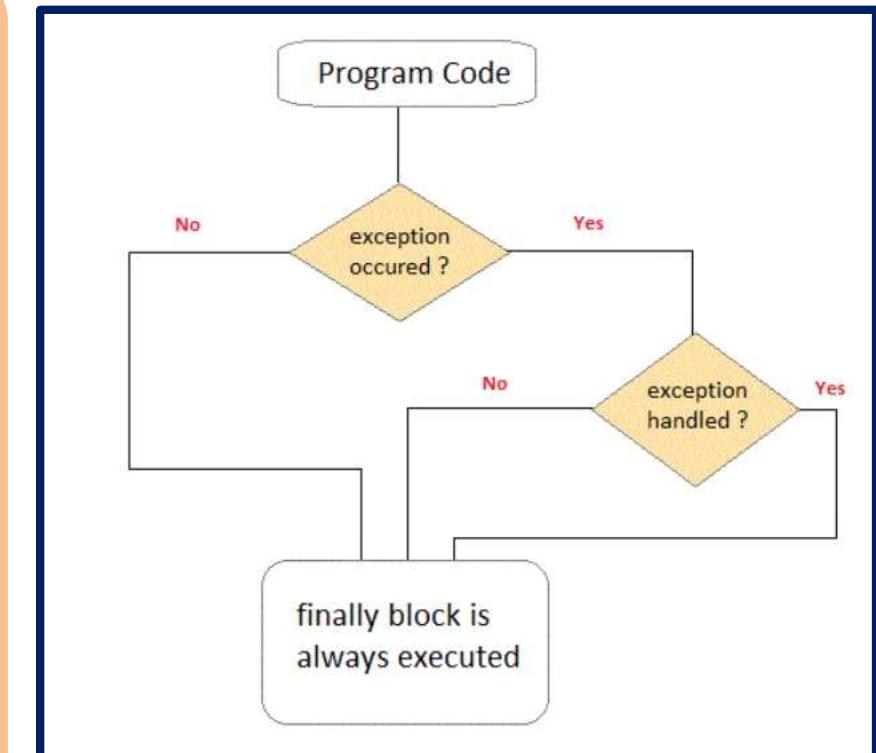
- A **finally** keyword is **used to create a block of code** that **follows a try block**.
- A finally block of code is **always executed whether an exception has occurred or not**.
- Using a finally block, it lets you run **any cleanup type statements** that you want to execute.
- A finally block appears **at the end of catch block**.

Syntax

```
try
{
    statement1;
    statement2;
}
finally
{
    statements;
}
```

Syntax

```
try
{
    statement1;
    statement2;
}
catch(Exceptiontype e)
{
    statement3;
}
finally
{
    statement4;
}
```



```
// Demonstrate finally.  
class Demo  
{  
    public static void main(String[] args)  
    {  
        int a[] = new int[2];  
        try  
        {  
            System.out.println("Access invalid element"+ a[3]);  
        }  
        catch(ArrayIndexOutOfBoundsException e)  
        {  
            System.out.println("Exception caught");  
        }  
        finally  
        {  
            System.out.println("finally is always executed.");  
        }  
    }  
}
```

OUTPUT

Exception caught
finally is always executed.

```
// Demonstrate finally.
```

```
class FinallyDemo
```

```
{
```

```
    static void procA() throws ArithmeticException
    {
        try
        {
            System.out.println("inside procA");
            throw new ArithmeticException("demo");
        }
        finally
        {
            System.out.println("procA's finally");
        }
    }
```

```
    static void procB()
```

```
{
```

```
        try
        {
            System.out.println("inside procB");
            return;
        }
        finally
        {
            System.out.println("procB's finally");
        }
    }
```

```
}
```

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

```
{
```

```
    try
    {

```

```
        procA();
    }
    catch (Exception e)
    {

```

```
        System.out.println("Exception caught");
    }
    procB();
}
```

```
}
```

```
}
```

OUTPUT

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
```

Creating own exception

- Java **allows us to create** our **own exception class** to provide own exception implementation.
- These type of exceptions are **called user-defined exceptions** or **custom exceptions**.
- You can **create your own exception** simply **by extending** java **Exception class**.
- You can **define a constructor for your Exception** (not compulsory) and you can **override the `toString()` function** to display **your customized message** on catch.

Syntax

```
class <name of the class> extends Exception
{
    public String toString()
    {
        Statements;
    }
}
```

Creating own exception

Example

```
class MyException extends Exception
{
    String str1;
    MyException(String s)
    {
        str1=s;
    }
    public String toString()
    {
        return ("MyException Occurred: "+str1);
    }
}
```

```
class Test
{
    public static void main(String args[])
    {
        try
        {
            System.out.println("Starting of try block");
            throw new MyException("This is My error Message");
        }
        catch(MyException exp)
        {
            System.out.println("Catch Block");
            System.out.println(exp);
        }
    }
}
```

OUTPUT

```
Starting of try block
Catch Block
MyException Occurred: This is My error Message
```

Example

```
class InsufficientFundsException extends Exception
{
    InsufficientFundsException(String s)
    {
        super(s);
    }
}

class Test
{
    public static void main(String[] args)
    {
        java.util.Scanner obj = new java.util.Scanner(System.in);
        double balance = 10000.00;
        double amt = obj.nextDouble();
        try
        {
            if(amt<=balance)
            {
                System.out.println("pls take the cash");
                balance = balance - amt;
            }
        }
    }
}
```

```
else
throw new InsufficientFundsException("No Balance in your account");
}
catch (InsufficientFundsException e)
{
    System.out.println(e.getMessage());
}
finally
{
    System.out.println("Updated Balance:"+balance);
    System.out.println("Pls take your card");
}
}
}
}
```

OUTPUT

Multi Threading

- Multi Threading is a **specialized form** of **multi tasking**.
- Multitasking is a **process of executing multiple tasks simultaneously**. We **use** multitasking to **utilize** the **CPU**. Multitasking can be **achieved in two ways**:
 - ✓ **Process-based Multitasking (Multiprocessing)**
 - ✓ **Thread-based Multitasking (Multithreading)**
- **Executing several tasks simultaneously** where **each task** is a separate **independent** process such type of multitasking is called **process based multitasking**.
- **Executing several tasks simultaneously** where **each task** is a **separate independent** part of the **same program**, is called **Thread based multitasking**. And **each independent part** is called a "**Thread**"
- **Multithreading** refers to a process of executing two or more threads simultaneously for maximum utilization of the CPU.
- Multithreading is a feature in Java that **concurrently executes two or more parts of the program**.

Multi Threading



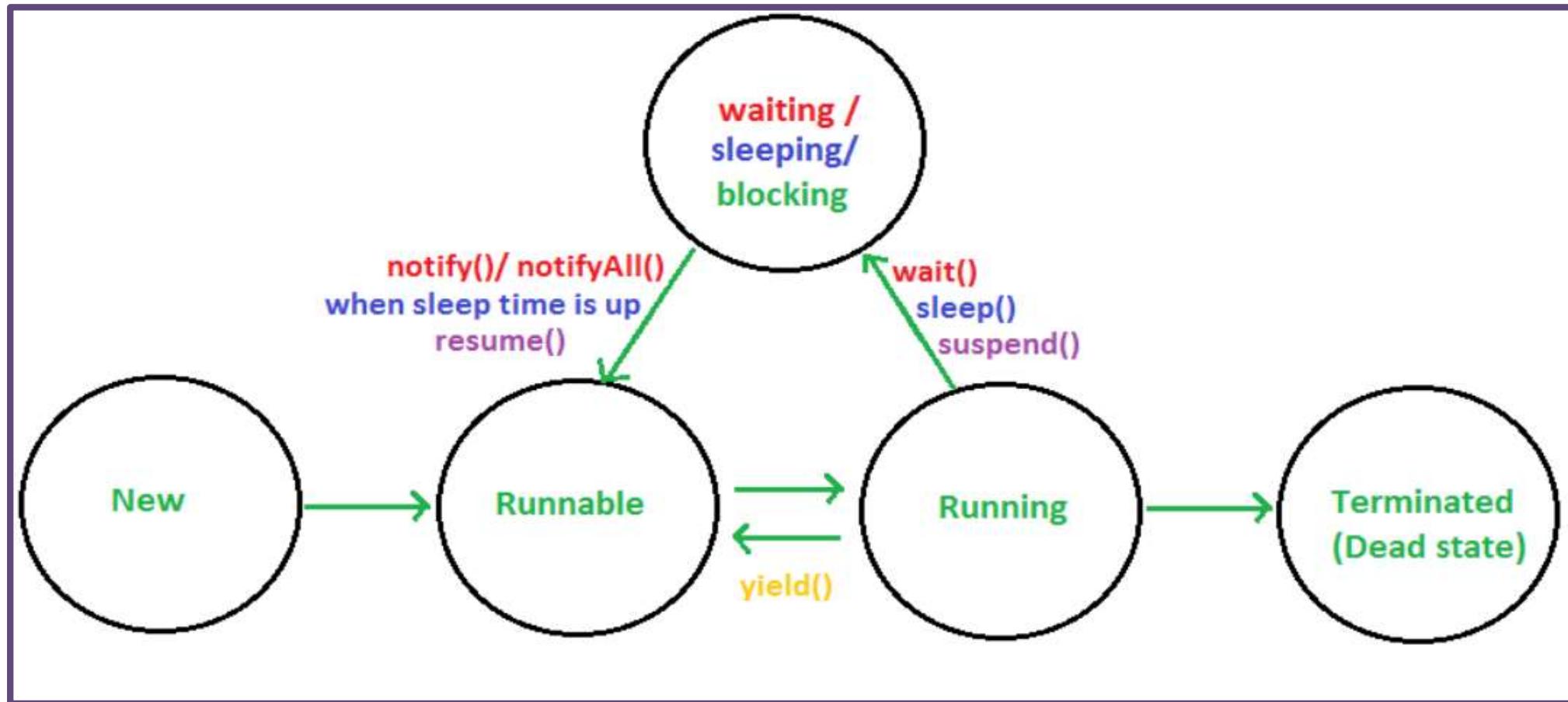
- A thread in Java is a ***lightweight process*** requiring **fewer resources**.
- **Each thread runs parallel** to each other.
- Java Provides **built in support** for **multi threaded programming**.
- Threads **share common memory** area.
- A **main program** is also single **thread**.

Advantages of MultiThreading

- Allows to write very **efficient program** that makes **maximum use of CPU**.
- Throughput – **Amount of work done in unit time increase**.
- It is used to **save time** as **multiple operations** are **performed** concurrently.
- The **threads are independent**, so it **does not block the user** to perform **multiple operations at the same time**.
- Since threads are independent, **other threads don't get affected** even if an **exception occurs** in a single thread.

Thread Life Cycle

- The **life cycle of a thread** in Java is **controlled by JVM**.
- During the **execution of thread**, it is in any of the following **five states**.
- Hence **Java thread life cycle** is defined in **five states**.



Thread Life Cycle

- i. **New:** The thread is in new state **when the instance of thread class is created but before the calling of start()** method.
- ii. **Runnable:** The **thread is in runnable state after the invocation start()** method, but the **thread scheduler** has **not selected** it to be the running thread. *Any number of threads exists in this state.*
- iii. **Running:** The thread is **in running state** when the **thread scheduler selects a thread for execution.**
- iv. **Waiting/blocked/sleeping:** this is the state **when the thread is still alive** but is **not eligible currently to run.**
- v. **(Blocked) Terminated:** A **thread** is in **terminated** or dead state when its **run() method exits.**
 - We can **define a Thread** in the **following 2 ways.**
 - i. By extending Thread class.
 - ii. By implementing Runnable interface

Multi Threading

- The **main important application** areas of **multithreading** are:
 - To implement **multimedia graphics**.
 - To develop **animations**.
 - To develop **video games** etc.
 - To develop **web** and **application servers**.

Thread class:

- Java provides **Thread class** to **achieve thread programming**.
- **Thread class** provides **constructors** and **methods** to create and **perform operations on a thread**.
- **Commonly used Constructors** of Thread class:
 - i. **Thread()**
 - ii. **Thread(String name)**
 - iii. **Thread(Runnable r)**
 - iv. **Thread(Runnable r, String name)**

Commonly used methods of Thread class

- Commonly used methods of Thread class:

1. public void start():

- **starts the execution of the thread.** start() method of Thread class is used to start a newly created thread.

It performs following tasks:

1. A new thread starts
2. The **thread moves from New state** to the **Runnable state.**
3. When the **thread gets a chance to execute**, its target **run() method will be called** and executes.

2. public void run(): It is the **entry point of a thread** , used to **perform action** for a thread.

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

4. public int setPriority(int priority): It is used to **set** or **change the priority** of the **thread.**

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

Commonly used methods of Thread class

6. public void setName(String name): It is used to **set** or **change** the **name of the thread.**

7. public static Thread currentThread(): returns the **reference** of **currently executing thread.**

8. Public static void sleep(long miliseconds): Causes the **currently executing thread to suspend** (temporarily cease execution) for the **specified number of milliseconds.**

9. public void join(): **waits** for a **thread to die/terminate.**

10. public boolean isAlive(): to **check** the **thread is still running.** Returns true if the thread is still running.

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

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

Creating Thread – Extending Thread Class



Step -1:

-Create a **new class** that **extends Thread**, and then **create an instance of that class**.

Step -2:

-The **extending class must override the run () method**, which is the **entry point of the new thread**.
-The **actual code** for the **thread** to execute **will be provided here**.
- Once the **run () method completes**, the **thread** will **die and terminate**.

Step -3:

-Calls **start () method** to begin **execution of new thread**.

Step -4:

-Call the **Thread class constructor** using **super keyword if necessary**.

Creating Thread – Extending Thread Class

```
class MyThread extends Thread
{
    public void run()
    {
        for(int i=0;i<5;i++)
        {
            System.out.println("Child Thread");
        }
    }
}
public class ThreadDemo1
{
    public static void main(String[] args)
    {
        MyThread t1 = new MyThread();
        t1.start();
        for(int i=0;i<5;i++)
        {
            System.out.println("main thread");
        }
    }
}
```

OUTPUT

main thread
main thread
main thread
main thread
main thread
Child Thread
Child Thread
Child Thread
Child Thread
Child Thread

Creating Thread – Implementing Runnable Interface



Step-1:

- To **create a new Thread**, the **class must implements Runnable interface.**

Step-2:

- Provide **implementation** for the **only one method run ()**, which is the entry point of newly created thread.
- The **actual code** for **the thread** will be **mentioned here**.
- Once the **run () method completes**, the **thread will die** and **terminate**.

Step-3:

- Instantiate an object of type **Thread** within the **newly created thread class**. `Thread(Runnable obj)`

Step-4:

- Call the **start () method explicitly to start the thread**.
- It **makes a call to run() method** to start the execution.

Creating Thread – Implementing Runnable Interface



```
class MyRunnable implements Runnable
{
    public void run()
    {
        for(int i=0;i<5;i++)
        {   System.out.println("Child Thread");
        }
    }
}
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyRunnable r=new MyRunnable();
        Thread t=new Thread(r); //here r is a Target Runnable
        t.start();
        for(int i=0;i<10;i++)
        {System.out.println("main thread");
        }
    }
}
```

OUTPUT

main thread
main thread
main thread
main thread
main thread
Child Thread
Child Thread
Child Thread
Child Thread
Child Thread

Creating Multiple Threads using Thread class

```
//Multithreading By Extending Thread class
class MyThread extends Thread
{
    public void run()
    {
        for(int i=0;i<3;i++)
        {
            System.out.println(i);
            try
            {
                Thread.sleep(1000);
            }
            catch(Exception e)
            {
                System.out.println(e);
            }
        }
    }
}
```

```
public class Own
{
    public static void main(String[] args)
    {
        Thread t = Thread.currentThread();
        System.out.println("ID:"+t);
        System.out.println("Name:"+t.getName());
        t.setName("Vardhaman");
        System.out.println("After renaming:"+t.getName());
        MyThread t1 = new MyThread();
        t1.start();
        try
        {
            t1.join();
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
        MyThread t2 = new MyThread();
        t2.start();
    }
}
```

OUTPUT

```
ID:Thread[#1,main,5,main]
Name:main
After renaming:Vardhaman
0
1
2
0
1
2
```

```
//Multithreading By Extending Thread class
class NewThread extends Thread
{
    NewThread(String threadname)
    {
        super(threadname);
        System.out.println("Child Thred -> " +this );
    }
    public void run()
    {
        try
        {
            for(int i = 5; i > 0; i--){
                System.out.println(getName ()+ ":" + i);
                sleep(500);
            }
        catch (InterruptedException e)
        {
            System.out.println(getName() + "Interrupted");
        }
        System.out.println(getName() + " Completed");
    } }
```

Creating Mutliple Threads using Thread class

```

class MultiThread1
{
    public static void main(String args[])
    {
        Thread t = Thread.currentThread();
        System.out.println("Name is --->" +t.getName());
        System.out.println("Main ---> " +t);
        NewThread t1 = new NewThread("One");
        NewThread t2= new NewThread("Two");
        NewThread t3 = new NewThread("Three");
        t1.start();
        t2.start();
        t3.start();
        try
        {
            for(int n=1;n<=5;n++){
                System.out.println("Main Thread->" +n);
                Thread.sleep(500);
            }
            catch (InterruptedException e)
            {
                System.out.println("Main thread Interrupted");
            }
            System.out.println("Main thread Completed");
        }
    }
}

```

Creating Multiple Threads using Thread class

OUTPUT

Name is --->main
 Main ---> Thread[#1,main,5,main]
 Child Thred -> Thread[#21,One,5,main]
 Child Thred -> Thread[#22,Two,5,main]
 Child Thred -> Thread[#23,Three,5,main]
 Main Thread->1
 One: 5
 Three: 5
 Two: 5
 Main Thread->2
 Two: 4
 Three: 4
 One: 4
 One: 3
 Main Thread->3
 Two: 3
 Three: 3
 One: 2
 Main Thread->4
 Two: 2
 Three: 2
 Three: 1
 Two: 1
 Main Thread->5
 One: 1
 One Completed
 Main thread Completed
 Two Completed
 Three Completed

```
class NewThread implements Runnable
{
    Thread t;
    NewThread()
    {
        t = new Thread(this, "Child Thread");
        System.out.println("Child thread: " + t);
        System.out.println("The thread name is "+t.getName());
        t.start();
    }
    public void run()
    {
        try
        {
            for(int i = 1; i <=10; i++)
            {
                System.out.println("Child Thread: ->" + t.getName() + ":" + i);
                Thread.sleep(500);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println("Child interrupted.");
        }
    }
}
```

Creating Multiple Threads using Runnable Interface

Creating Multiple Threads using Runnable Interface

```
public class MultiThread2
{
    public static void main(String args[ ] )
    {
        NewThread t1 = new NewThread();
        NewThread t2 = new NewThread();
        NewThread t3 = new NewThread();
        try
        {
            for(int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e){
            System.out.println("Main thread interrupted.");
        }
    }
}
```

Thread Priorities

- Thread priorities are used by the thread scheduler to decide which thread should be allowed to run.
- A higher-priority thread get more CPU time than lowerpriority thread.
- Thread class defines the method `setPriority()` to assign priority to a thread.
- `final void setPriority(int level)` The value of level must be within the range `MIN_PRIORITY` and `MAX_PRIORITY`. Currently, these values are 1 and 10, respectively.
- To return a thread to default priority, specify `NORM_PRIORITY`, which is currently 5.
- These priorities are defined as static final variables within Thread.

public static final int MIN_PRIORITY (1)

public static final int MAX_PRIORITY (10)

public static final int NORM_PRIORITY (5)

- To obtain the current priority setting by calling the `getPriority()` method of Thread class is used.

Thread Priorities

```
class NewThread extends Thread
{
    NewThread(String name)
    {
        super (name);
    }
    public void run()
    {
        System.out.println("Hello -->"+ getName() + "->" +getPriority() );
    }
}

public class PriorityDemo
{
    public static void main(String args[])
    {
        Thread t = Thread.currentThread();
        System.out.println("The main thread priority is: " +t.getPriority());
        NewThread t1 = new NewThread("One");
        NewThread t2 = new NewThread("Two");
        NewThread t3 = new NewThread("Three");
        t1.setPriority(Thread.MIN_PRIORITY);
        t2.setPriority(Thread.MAX_PRIORITY);
        t3.setPriority(5);
        t1.start();
        t2.start();
        t3.start();
    }
}
```

OUTPUT

The main thread priority is : 5
Hello -->Two->10
Hello -->Three->5
Hello -->One->1

Synchronizing Threads

- When **two or more threads need access** to a **shared resource**, they **need some way to ensure** that the **resource will be used by only one thread at a time**. The **process** by which this is **achieved** is **called synchronization**.
- Java **provides unique, language-level** support for it.
- Synchronization **allows only one thread to access a shared resource**.
- The **advantages of Synchronization** are:
 - ❖ To prevent **thread interferences**.
 - ❖ To prevent **inconsistency** of data.
- Thread **Synchronization can be of two forms**
 - i. **Mutual Exclusion**
 - ii. **Inter Thread communication.**

Synchronizing Threads

1. Mutual Exclusion

Keeps threads from interfering with one another while sharing data. **Allows only one thread to access shared data.** Mutual exclusive threads can be **implemented using**

i) Synchronized Method

i) Synchronized Method

- ✓ A **method defines as synchronized**, then the method is a synchronized method.
- ✓ Synchronized method is **used to lock an object for any shared resource.**
- ✓ When a **thread invokes a synchronized method**, it **automatically acquires lock** for that object and **releases it when the thread completes its task.**

ii) Synchronized Block

The form of synchronized method is:

```
class <class Name>
{
    synchronized type methodName(arguments)
    {
        //method body for Synchronization
    }
}
```

```
//synchronized Method - for Synchronization
class Table
{
    synchronized void printTable(int n)
    {
        try{
            for(int i=1;i<=10;i++)
            {
                System.out.println(n+"x"+i+"="+n*i);
                Thread.sleep(1000);
            }
        }
        catch(InterruptedException e)
        {
            System.out.println(e);
        }
    }
}

class MyThread1 extends Thread
{
    Table t;
    MyThread1(Table tab)
    {
        t=tab;
    }
    public void run()
    {
        t.printTable(8);
    }
}
```

Synchronizing Threads

```
class MyThread2 extends Thread
{
    Table t;
    MyThread2(Table tab)
    {
        t=tab;
    }
    public void run()
    {
        t.printTable(9);
    }
}

public class SyncDemo1
{
    public static void main(String args[])
    {
        Table obj = new Table();
        MyThread1 t1=new MyThread1(obj);
        MyThread2 t2=new MyThread2(obj);
        t1.start();
        t2.start();
    }
}
```

OUTPUT

8x1=8
8x2=16
8x3=24
8x4=32
8x5=40
8x6=48
8x7=56
8x8=64
8x9=72
8x10=80
9x1=9
9x2=18
9x3=27
9x4=36
9x5=45
9x6=54
9x7=63
9x8=72
9x9=81
9x10=90

Synchronizing Threads

```
//synchronized Method - for Synchronization
class Display
{
    public synchronized void wish(String name)
    {
        for(int i=0;i<3;i++)
        {
            System.out.print("good morning:");
            try
            {
                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {}
            System.out.println(name);
        }
    }
}
```

```
class MyThread extends Thread
{
    Display d;
    String name;
    MyThread(Display d2,String n2)
    {
        d=d2;
        name=n2;
    }
    public void run()
    {
        d.wish(name);
    }
}
```

```
class SynchronizedDemo2
{
    public static void main(String[] args)
    {
        Display d1=new Display();
        MyThread t1=new MyThread(d1,"dhoni");
        MyThread t2=new MyThread(d1,"yuvaraj");
        t1.start();
        t2.start();
    }
}
```

OUTPUT

```
good morning:dhoni
good morning:dhoni
good morning:dhoni
good morning:yuvaraj
good morning:yuvaraj
good morning:yuvaraj
```

Synchronizing Threads

ii) Synchronized Block:

- ✓ Synchronized block can be used to **perform synchronization** on **any specific resource of a method**.
- ✓ Synchronized block is **used to lock** an object for **any shared resource**.
- ✓ **Scope of Synchronized block** is **smaller than the method**.

The form of synchronized block is

```
synchronized(object)
{
    //Statements to be synchronized
}
```

//Thread Synchronization using - synchronized block

class Table

{

 void printTable(int n)

{

 System.out.println("I am non synched stmt");

synchronized(this)

{

for(int i=1;i<=10;i++)

{

System.out.println(n*i);

try{ Thread.sleep(1000);

}

catch(InterruptedException e)

{ System.out.println(e);

}

}

} //end of sync block

 System.out.println(" I am last non synched stmt ");

} //end of the method

}

class MyThread1 extends Thread

{ Table t;

 MyThread1(Table tab)

{

 t=tab;

}

 public void run()

{

 t.printTable(5);

}

Synchronizing Threads



class MyThread2 extends Thread

{

 Table t;

 MyThread2(Table tab)

{

 t=tab;

}

 public void run()

{

 t.printTable(100);

}

 public class SyncDemo2

{

 public static void main(String args[])

{

 Table obj = new Table();

 MyThread1 t1=new MyThread1(obj);

 MyThread2 t2=new MyThread2(obj);

 t1.start();

 t2.start();

}

}

InterThread Communication

- Inter-thread communication in Java is a **technique through which multiple threads communicate with each other.**
- It provides **an efficient way** through **which more than one thread communicate with each other** by **reducing CPU idle time.**
- When **more than one threads are executing simultaneously**, **sometimes they need to communicate with each other** by exchanging information with each other.
- A **thread exchanges information before or after it changes its state.**
- There are **several situations** where **communication** between threads is **important.**
- For example, suppose that there are **two threads A and B.**
- **Thread B uses data produced by Thread A** and performs its task.

InterThread Communication

- If Thread B waits for Thread A to produce data, it will waste many CPU cycles. But if threads A and B communicate with each other when they have completed their tasks, they do not have to wait and check each other's status every time.
- This type of information exchanging between threads is called inter-thread communication in Java.
- Inter thread communication in Java can be achieved by using three methods provided by Object class of java.lang package. They are:
 - i. **wait()**
 - ii. **notify()**
 - iii. **notifyAll()**
- These methods can be called only from within a synchronized method or synchronized block of code otherwise, an exception named **IllegalMonitorStateException** is thrown.

InterThread Communication

i. wait():

- This method is used to make the particular Thread wait until it gets a notification.
- This method pauses the current thread to the waiting room dynamically.

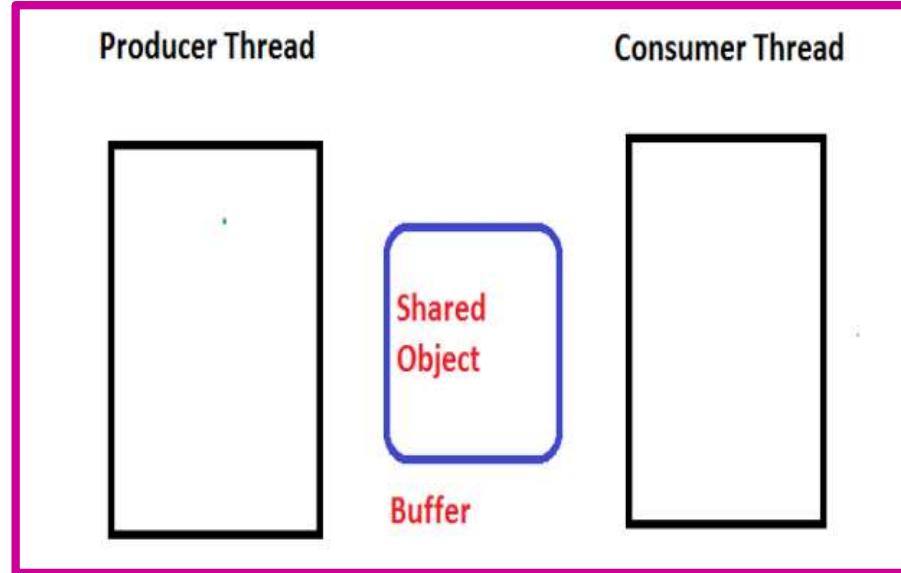
ii. notify():

- This method is used to send the notification to one of the waiting thread so that thread enters into a running state and execute the remaining task.
- This method wakeup a single thread into the active state.

iii. notifyAll():

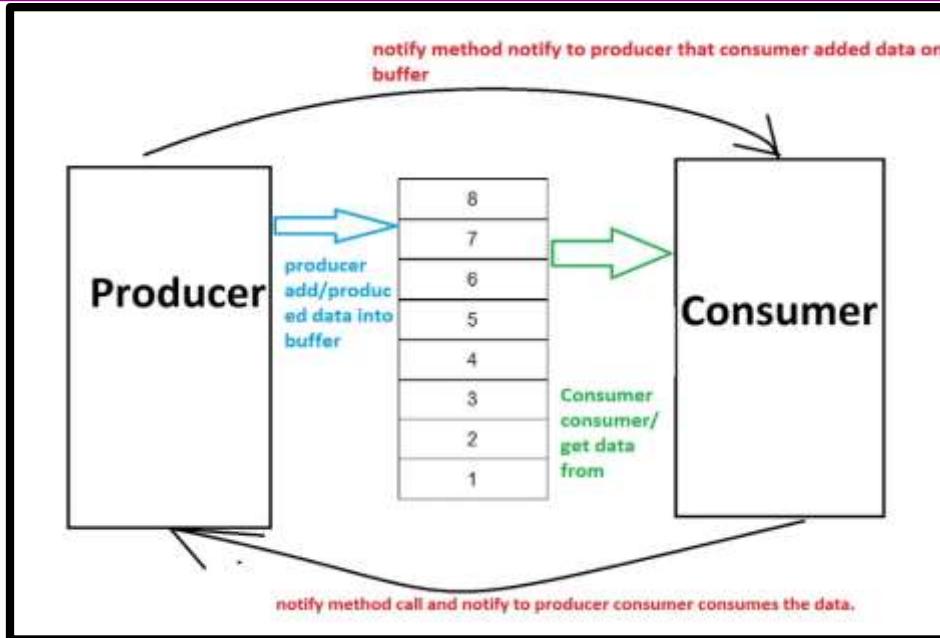
- This method is used to send the notification to all the waiting threads so that all thread enters into the running state and execute simultaneously.
- This method wakes up all the waiting threads that act on the common objects.

Producer Consumer Problem for Interthread Communication



- Producer Thread- Produce items to Buffer (Add Items) Consumer Thread- Consume items from Buffer (Removes Items).
- The **two conditions for Producer – Consumer** Problem is :
 1. Producer cannot add an item into a buffer if it is full
 2. Consumer cannot consume an item if it is empty.
- If **no communication**, these two conditions are not satisfied then the **CPU is always in polling (loop)**.
- To **Save CPU time** in Java **Interthread communication is used**.

Producer Consumer Problem for Interthread Communication



- Producer Thread- Produce items to Buffer (Add Items) Consumer Thread- Consume items from Buffer (Removes Items).
- The **two conditions for Producer – Consumer** Problem is :
 1. Producer cannot add an item into a buffer if it is full
 2. Consumer cannot consume an item if it is empty.
- If **no communication**, these two conditions are not satisfied then the **CPU is always in polling (loop)**.
- To **Save CPU time** in Java **Interthread communication is used**.

Producer Consumer Problem for Interthread Communication



//Producer-Consumer problem ---> Inter Thread Communication.

```
class Buffer
{
    int item;
    boolean produced = false;
    synchronized void produce(int x)
    {
        if(produced)
        {
            try{
                wait();
            }
            catch(InterruptedException ie)
            {
                System.out.println("Exception Caught");
            }
        }
        item =x;
        System.out.println("Producer - Produced-->" +item);
        produced =true;
        notify();
    }
}
```

```
synchronized int consume()
{
    if(!produced)
    {
        try{
            wait();
        }
        catch(InterruptedException ie)
        {
            System.out.println("Exception Caught " +ie);
        }
    }
    System.out.println("Consumer - Consumed " +item);
    produced =false;
    notify();
    return item;
}
```

Producer Consumer Problem for Interthread Communication



```
class Producer extends Thread
{
    Buffer b;
    Producer( Buffer b)
    {
        this.b = b;
        start();
    }
    public void run()
    {
        b.produce(10);
        b.produce(20);
        b.produce(30);
        b.produce(40);
        b.produce(50);
    }
}
```

```
class Consumer extends Thread
{
    Buffer b;
    Consumer(Buffer b)
    {
        this.b = b;
        start();
    }
    public void run()
    {
        b.consume();
        b.consume();
        b.consume();
        b.consume();
    }
}
public class PCDemo
{
    public static void main(String args[])
    {
        Buffer b = new Buffer(); //Synchronized Object
        Producer p = new Producer(b);
        Consumer c = new Consumer(b);
    }
}
```



01 **TOPIC** Collection classes-
ArrayList

02 **TOPIC** LinkedList

03 **TOPIC** HashSet, and TreeSet

04 **TOPIC** EVENT HANDLING-
Delegation Event Model

05 **TOPIC** Event Sources, Event
Classes

06 **TOPIC** Event Listener Interfaces

07 **TOPIC** Handling Mouse and
Keyboard Events

08 **TOPIC** Adapter classes

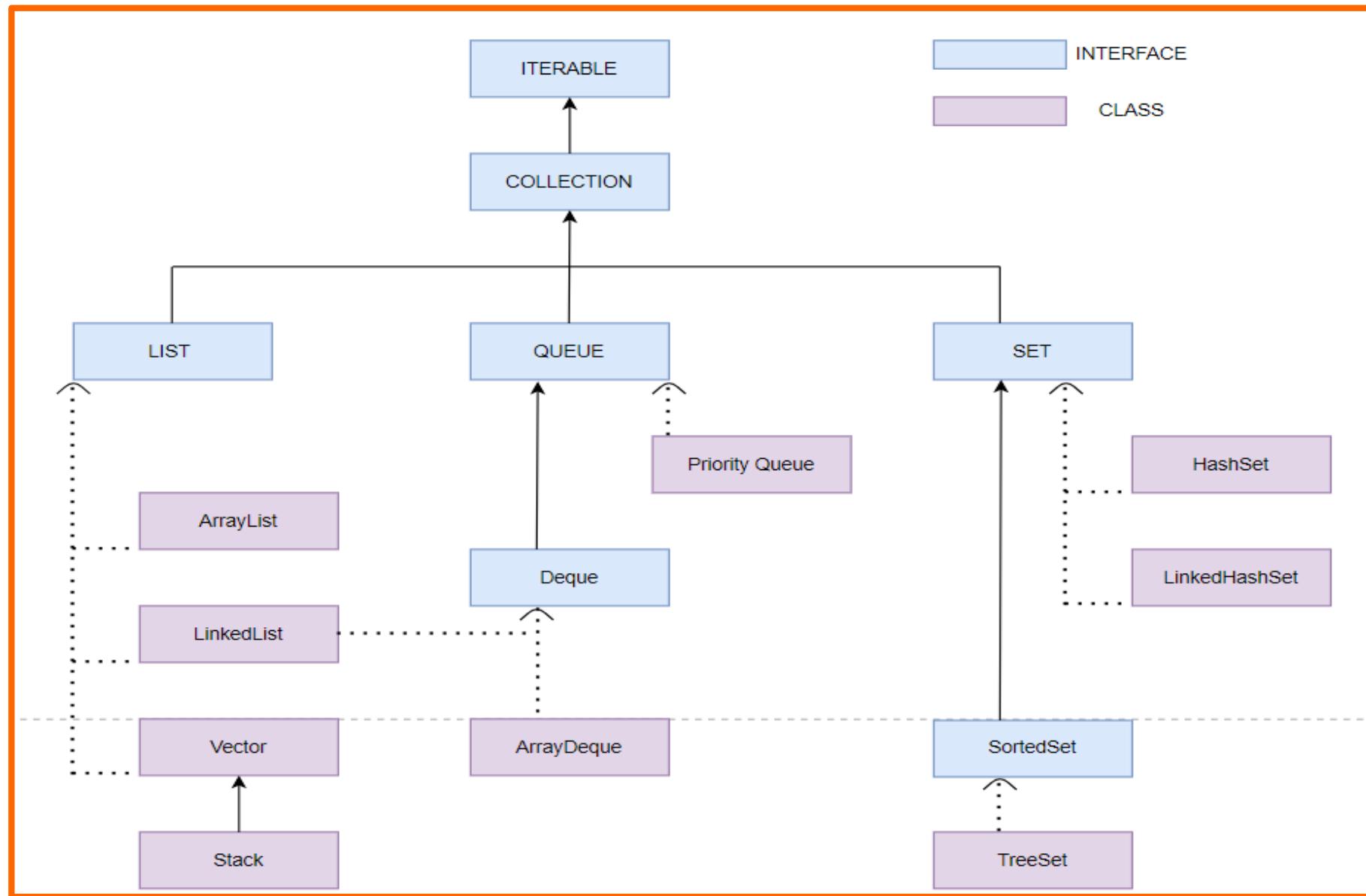
Collections Framework

- Collections in java is a **framework** that **provides an architecture** to **store** and **manipulate** the **group of objects**.
- The Java **collections framework provides** a **set of interfaces** and **classes** to implement **various data structures**(LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet etc)
- **All the operations** that you perform on a **data** such as **searching, sorting, insertion, manipulation, deletion** etc. can be **performed by Java Collections**.
- Java Collection simply means a **single unit of objects**.
- The **java.util package contains** all the **classes** and **interfaces** for **Collection framework**.

Example:

- The **LinkedList class** of the **collections framework** provides the **implementation** of the **doubly-linked list data structure**.

Hierarchy of Collection Framework





Hierarchy of Collection Framework

i. List Interface:

-The List interface **is an ordered collection** that **allows us to add and remove elements like an array**.

ii. Set Interface:

-The Set interface **allows us to store elements** in **different sets similar** to the **set in mathematics**.

-**Insertion order not preserved** i.e., They appear in the different order in which we inserted.

-**Duplicate elements** are **not allowed**.

-**Heterogeneous objects** are **allowed**.

iii. Queue Interface:

-The Queue interface is **used when we want to store** and **access elements in First In, First Out** manner.

iv. Java Iterator Interface:

-The **Iterator** interface provides **methods that can be used to access elements** of **collections**.

Basic methods of Collection Framework

SNo	Method	Description
1	<code>add(element)</code>	It is used to insert an element in this collection.
2	<code>addAll(collection_name)</code>	It is used to insert the specified collection elements in the invoking collection .
3	<code>remove(index/element)</code>	It is used to delete an element from the collection .
4	<code>removeAll(collection_name)</code>	It is used to delete all the elements of the specified collection from the invoking collection .
6	<code>int size()</code>	It returns the total number of elements in the collection.
7	<code>clear()</code>	It removes the total number of elements from the collection.
8	<code>contains(element)</code>	It is used to search an element .
9	<code>public Iterator iterator()</code>	It returns an iterator .
11	<code>boolean isEmpty()</code>	It checks if collection is empty .
12	<code>boolean equals(collection_name)</code>	It matches two collections .

1. ArrayList

- **ArrayList** is a part of **collection framework** and it is **implements** the **List interface**.
- It is **present in java. util package**.
- It provides a **dynamic array for storing the element**.
- It is an **array** but **there is no size limit**.
- We can **add or remove elements easily**.
- It is **more flexible than a traditional array**.
- It can **dynamically increase or decrease in size**.
- Array lists are **created with an initial size**. When this **size is exceeded**, the collection is **automatically enlarged**.
- When an **ArrayList is created, its default capacity or size is 10** . The **size of the ArrayList grows** based on **load factor** and **current capacity**.
- The **Load Factor** is a **measure to decide when to increase its capacity**. The **default value of load factor** of an **ArrayList** is **0.75f**

1. ArrayList

- ArrayList expands its capacity after each threshold which is calculated as the product of current capacity and load factor of the ArrayList instance.

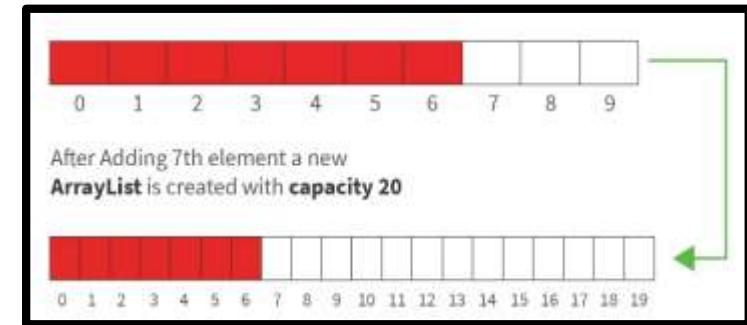
$$\text{Threshold} = (\text{Load Factor}) * (\text{Current Capacity})$$

- For example, if the user creates an ArrayList of size 10,

$$\text{Threshold} = \text{Load Factor} * \text{Current Capacity}$$

$$= 0.75 * 10$$

$$\cong 7$$



- This means after adding the 7th element to the list, the size will increase as it has reached the threshold value.
- Internally, a new ArrayList with a new capacity is created and the elements present in the old ArrayList are copied in the new ArrayList.
- The new capacity of the ArrayList is calculated to be 50% more than its old capacity.

$$\text{new_capacity} = \text{old_capacity} + (\text{old_capacity} \gg 1)$$

- In the above formula, the new capacity is calculated as 50% more than the old capacity. Here, $(\text{old_capacity} \gg 1)$ gives us half of the old capacity.

How to create ArrayList

1. `ArrayList<>():` It creates an **empty ArrayList** instance with **default initial capacity i.e. 10.**

Syntax-1

```
ArrayList <DataType> VariableName = new ArrayList <DataTYpe>()
```

Example

```
ArrayList <int> a = new ArrayList <int>()
```

2. `ArrayList(int capacity):` This constructor creates an empty ArrayList with initial capacity as mentioned by the user. // Creating an ArrayList with initial capacity equal to 50.

Syntax-2

```
ArrayList <Datatype> VariableName = new ArrayList <String> (size);
```

Example

```
ArrayList <String> arr = new ArrayList <String> (50);
```

3. `ArrayList():` An empty constructor.

Syntax-3

```
ArrayList <classname> objname = new ArrayList <classname> ( );
```

Example

```
ArrayList <Emp> obj = new ArrayList <Emp>( );
```

i. ArrayList

```
// Demonstrate ArrayList.  
import java.util.*;  
class Demo  
{  
    public static void main(String[] args)  
    {  
        ArrayList< String> a = new ArrayList< String>();  
        a.add("Java");  
        a.add("Python");  
        a.add("C-language");  
        System.out.println(a);  
    }  
}
```

OUTPUT

[Java,Python,C-language]

1. ArrayList Methods

SNO	Method	Example
1	<code>add(element)</code>	It is used to insert the specified element
2	<code>addAll(collection_name)</code>	It is used to add all of the elements in the specified collection to the end of current list .
3	<code>remove(int index)</code>	It is used to remove the element present at the specified position in the list.
4	<code>removeAll(int index)</code>	It removes all the elements from the list that are also present in the specified list .
5	<code>get(int index)</code>	It is used to find the index of particular element in the list .
6	<code>set(int index, E element)</code>	It is used to replace the specified element in the list, present at the specified position .
7	<code>size()</code>	It is used to find The length of the List .
8	<code>clear()</code>	It is used to clear entire list .
9	<code>sort()</code>	It is used to arrange entire list in ascending order .
10	<code>reverseOrder()</code>	It is used to arrange entire list in descending order .

1. ArrayList

```
import java.util.*;
public class TestA2
{
    public static void main(String[] args) throws Exception
    {
        ArrayList <String> a = new ArrayList <String>();
        ArrayList <String> b = new ArrayList <String>();
        a.add("Mango");
        a.add("Apple");
        a.add("Orange");
        b.add("Pineapple");
        b.add("Banana");
        b.add("Grapes");
        a.addAll(b);
        a.remove(4);
        System.out.println("Before removing all elements b is:" + b);
        b.removeAll(a);
        b.clear();
        System.out.println("After removing all elements b is:" + b);
        System.out.println(a);
        System.out.println("The size of the list b is:" + b.size());
        a.set(4, "Guava");
        System.out.println("After set list a is:" + a);
        System.out.println("Get the value from a:" + a.get(2));
        Collections.sort(a);
        System.out.println("After sorting list a is:" + a);
        Collections.sort(a, Collections.reverseOrder());
        System.out.println("After sorting list a is:" + a);

    }
}
```

OUTPUT

[Mango, Apple, Orange, Pineapple, Banana, Grapes]
After set list a is:[Mango, Apple, Orange, Pineapple, Guava, Grapes]
Get the value from a:Orange
After sorting list a is:[Apple, Grapes, Guava, Mango, Orange, Pineapple]
After sorting list a is:[Pineapple, Orange, Mango, Guava, Grapes, Apple]

```
import java.util.*;
class Employee
{
    int eid;
    String ename;
    double sal;
    public Employee(int x, String y, double z)
    {
        eid=x;
        ename=y;
        sal = z;
    }
}
```

OUTPUT

The number of employees are:3

The employess data is

101:Amar:75000.5

102:Akhil:85000.5

103:Anush:19500.5

After removing number of employees are:2

1. ArrayList

```
public class EmpAlist
{
    public static void main(String[] args)
    {
        ArrayList<Employee> list =new ArrayList<Employee>();
        Employee e1=new Employee(101,"Amar",75000.50);
        Employee e2=new Employee(102,"Akhil",85000.50);
        Employee e3=new Employee(103,"Anush",19500.50);
        list.add(e1);
        list.add(e2);
        list.add(e3);
        System.out.println("\n The number of employees are:" +list.size());
        System.out.println("\n The employess data is \n");
        for(Employee e:list)
        {
            System.out.println(e.eid+":"+e.ename+":"+e.sal);
            System.out.println();
        }
        list.remove(2);
        System.out.println("\n After removing number of employees are:" + list.size());
    }
}
```

LinkedList class

- **LinkedList class** uses a **doubly LinkedList** to store element. i.e., the **user can add data at the first position as well as the last position.**
- If we need to perform **insertion /Deletion operation** the **LinkedList** is preferred.
- **LinkedList** is used to implement **Stacks** and **Queues**.

How to create a LinkedList

Syntax

```
LinkedList<DataType> VariableName = new LinkedLits < DataType>();
```

LinkedList

```
//Program to Demonstrate LinkedList
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        LinkedList<String> cars = new LinkedList<String>();
        cars.add("BMW");
        cars.add("FORD");
        cars.add("KIA");
        System.out.println(cars);
    }
}
```

OUTPUT
[BMW, FORD, KIA]

Methods of LinkedList

SNO	Method	Description
1	<code>add(eelement)</code>	It is used to add the specified element to the end of a list.
2	<code>add(int position, element)</code>	It is used to insert the specified element at the specified position in a list.
3	<code>addAll(collection_Name)</code>	It is used to add all of the elements in the specified collection to the end of this list.
4	<code>addFirst(element)</code>	It is used to insert the given element at the beginning of a list.
5	<code>addLast(element)</code>	It is used to append the given element to the end of a list.
6	<code>getFirst()</code>	It is used to return the first element in a list.
7	<code>getLast()</code>	It is used to return the last element in a list.

Methods of LinkedList

SNO	Method	Description
8	<code>removeFirst()</code>	It removes and returns the first element from a list .
9	<code>removeLast()</code>	It removes and returns the last element from a list.
10	<code>removeFirstOccurrence(Object)</code>	It is used to remove the first occurrence of the specified element in a list
11	<code>removeLastOccurrence(Object)</code>	It removes the last occurrence of the specified element in a list
12	<code>lastIndexOf(Object)</code>	It is used to return the position in a list of the last occurrence of the specified element, or -1 if the list does not contain any element
13	<code>indexOf(Object)</code>	It is used to return the position in a list of the first occurrence of the specified element, or -1 if the list does not contain any element.
14	<code>get(position)</code>	It is used to return the element at the specified position in a list.

```
import java.util.LinkedList;
class TestLL1
{
    public static void main(String[] args) throws Exception
    {
        LinkedList <Integer> a = new LinkedList <Integer>();
        a.add(10);
        a.add(20);
        a.add(30);
        System.out.println(a);
        System.out.println(a.getFirst());
        System.out.println(a.getLast());
        a.addFirst(40);
        a.addLast(20);
        a.add(4,35);
        System.out.println(a);
        System.out.println(a.indexOf(20));
        a.removeFirstOccurrence(20);
        System.out.println(a);
        a.removeLastOccurrence(20);
        System.out.println(a);
    }
}
```

LinkedList

OUTPUT

```
[10, 20, 30]
10
30
[40, 10, 20, 30, 35, 20]
2
[40, 10, 30, 35, 20]
[40, 10, 30, 35]
```

Difference between ArrayList and Linked List

- The **ArrayList class creates** the list which is **internally stored** in a **dynamic array** that **grows** or **shrinks** in **size as the elements are added** or **deleted from it**.
- **LinkedList** also **creates** the **list** which is **internally stored** in a **DoublyLinked List**.
- **Both the classes** are **used to store** the **elements** in the **list**.
- **ArrayList allows random access** to the **elements** in the list as it operates on **an index-based data structure**.
- **LinkedList does not allow random access** as it **does not have indexes** to access elements directly, it has to traverse the list to retrieve or access an element from the list.

3. HashSet class

- HashSet stores the elements by using Hashing mechanism.
- It contains unique elements only.
- It allows null values.
- It does not maintain insertion order. It inserted elements based on their hashcode.
- HashSet is the best approach for the search operation.
- In HashSet get() and set() method not present because for get and set method index is required and in HashSet elements stores at a random address
- There are three different ways to create HashSet:

i. `HashSet hs = new HashSet();`

- ✓ Here, HashSet default capacity to store elements is 16 with a default load factor/fill ratio of 0.75.
- ✓ Load factor is if HashSet stores 75% element then it creates a new HashSet with increased capacity.

ii. `HashSet hs = new HashSet (100);`

- ✓ Here 100 is an initial capacity.

iii. `HashSet<Integer> hs = new HashSet < Integer> ();`: It can store only integer values in hashset.

HashSet

```
//Use HashSet methods to perform operations on collection of data
import java.util.HashSet;
public class Test
{
    public static void main(String[] args)
    {
        HashSet h = new HashSet();
        h.add(7);
        h.add("A");
        h.add(4);
        h.add(3);
        h.add("Hai");
        h.add(null);
        System.out.println(h);
        System.out.println(h.add(4));
    }
}
```

OUTPUT

```
[null, A, Hai, 3, 4, 7]
false
```

HashSet

```
//Write a program to remove duplicate elements.  
import java.util.*;  
public class Main  
{  
    public static void main(String[] args)  
    {  
        int a[]={1,1,1,2,3,5,5,5,6,6,9,9,9};  
        HashSet <Integer> hs = new HashSet<Integer>();  
        for(int i=0;i<a.length;i++)  
        {  
            hs.add(a[i]);  
        }  
        for(int i:hs)  
        {  
            System.out.print(i+" ");  
        }  
    }  
}
```

OUTPUT
1 2 3 5 6 9

HashSet

```
//Write a program to check 1 to 10 numbers are existing in hashset or not
import java.util.*;
public class TestHS3
{
    public static void main(String[] args)
    {
        HashSet<Integer> h = new HashSet<Integer>();
        h.add(8);
        h.add(3);
        h.add(7);
        for(int i = 1; i <= 10; i++)
        {
            if(h.contains(i))
            {
                System.out.println(i + " was found in the set.");
            }
            else
            {
                System.out.println(i + " was not found in the set.");
            }
        }
    }
}
```

OUTPUT

1 was not found in the set.
2 was not found in the set.
3 was found in the set.
4 was not found in the set.
5 was not found in the set.
6 was not found in the set.
7 was found in the set.
8 was found in the set.
9 was not found in the set.
10 was not found in the set.

3. HashSet Methods

SN	Method	Description
1	add(element)	It is used to add the specified element to this set if it is not already present.
2	clear()	It is used to remove all of the elements from the set.
3	contains(Object o)	It is used to return true if this set contains the specified element.
4	isEmpty()	It is used to return true if this set contains no elements.
5	iterator()	It is used to return an iterator over the elements in this set.
6	remove(element)	It is used to remove the specified element from this set if it is present.
7	size()	It is used to return the number of elements in the set.

4. TreeSet class

- TreeSet class **implements** the Set interface that **uses a tree for storage**.
- It **stores the elements** in **ascending order**.
- It **uses a Tree structure** to **store elements**.
- It **contains unique elements** only like HashSet.
- It's **access** and **retrieval times** are **quite fast**.
- **How to create a LinkedList**

Syntax

```
TreeSet<Integer> numbers = new TreeSet<>();
```

- It **creates an empty tree** set that will be sorted in an **ascending order** according to the **natural order of the tree set**
- **TreeSet(Collection C)** //It creates a new tree set that contains the elements of the Collection C

TreeSet

```
//Program to Demonstrate TreeSet
import java.util.*;
class Demo
{
    public static void main(String args[])
    {
        TreeSet t=new TreeSet();
        t.add("Z");
        t.add("D");
        t.add("T");
        t.add("a");
        System.out.println(t);
    }
}
```

OUTPUT
[D, T, Z, a]

```
//Program to Demonstrate TreeSet
import java.util.*;
class TestTL2
{
    public static void main(String args[])
    {
        TreeSet t = new TreeSet();
        t.add("Akhil");
        t.add("Hemanth");
        t.add("Shiva");
        System.out.println("Ascending:");
        Iterator i=t.iterator();
        while(i.hasNext())
        {
            System.out.println(i.next());
        }
        System.out.println("Descending:");
        Iterator j=t.descendingIterator();
        while(j.hasNext())
        {
            System.out.println(j.next());
        }
    }
}
```

OUTPUT
Ascending:
Akhil
Hemanth
Shiva
Descending:
Shiva
Hemanth
Akhil

4. TreeSet Methods

SNO	Method	Description
1	ceiling(E e)	It returns the equal or closest greatest element of the specified element from the set, or null there is no such element.
2	floor(E e)	It returns the equal or closest least element of the specified element from the set, or null there is no such element.
3	SortedSet headSet(Element)	It returns the group of elements that are less than the specified element .
4	higher(E e)	It returns the closest greatest element of the specified element from the set , or null there is no such element.
5	Iterator iterator()	It is used to iterate the elements in ascending order .
6	lower(E e)	It returns the closest least element of the specified element from the set, or null there is no such element.
7	pollFirst()	It is used to retrieve and remove the lowest(first) element .
8	pollLast()	It is used to retrieve and remove the highest(last) element .
9	E first()	It returns the first (lowest) element currently in this sorted set.
10	E last()	It returns the last (highest) element currently in this sorted set.



Delegation Event Model

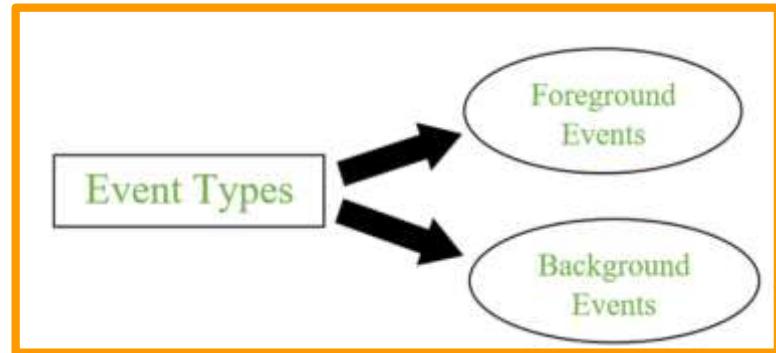
- The **Delegation Event Model** is a **programming pattern used** in Java for **handling events** in **graphical user interfaces (GUIs)**.
- **Any program** that **uses GUI** is **event driven**.
- **Event** describes the **change in state** of **any object**.

Example : Pressing a button, Entering a character in Textbox, Clicking or Dragging a mouse, etc.

- The **modern approach to event processing** is **based** on the **Delegation Model**.
- It defines a **standardized** and **compatible mechanism for generating** and **processing events**.
- In this approach, a **source generates** an **event** and **sends** it to **one or more listeners**.
- The **listener sits** and **waits** for an **event to occur**. When it gets an event, it is processed by the listener and returned.
- Events are **supported by a number of Java packages**, like **java.util**, **java.awt** and **java.awt.event**.

Types of Events

- The events can be broadly classified into two categories:



i. **Foreground Events :**

- ✓ Those events which require the direct interaction of user.
- ✓ They are generated as consequences of a person interacting with the graphical components in GUI.

Example:

- ✓ Clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page etc.

ii. **Background Events:**

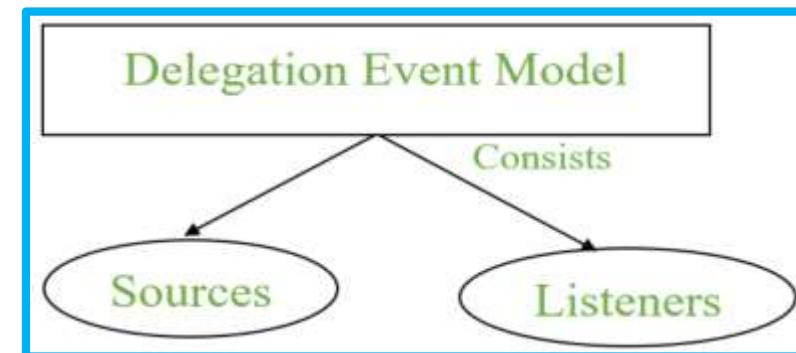
- ✓ Events that don't require interactions of users are known as background events.

Example :

- ✓ Operating system interrupts, hardware or software failure, timer expires, an operation completion.

What is Event Handling?

- It is a mechanism to control the events and to decide what should happen after an event occurs.
- **Event handling** in Java is the **procedure that controls an event** and **performs appropriate action if it occurs.**
- **Event handlers** are **responsible** for **defining the actions** or behaviors **that should occur in response to specific events.**
- They contain the **implementation code** that **handles the event** and **performs** the desired **tasks.** Event handlers are typically implemented as methods within a class.
- **When an event occurs**, the associated **event handler is invoked** to **respond to that event.**
- To handle the events, Java follows the Delegation Event model.
- The Delegation Event model consists of two components:
 - 1.Sources (or) Event Source
 - 2.Listeners (or) Event Listener



Delegation Event model

1. Event Sources

- ✓ Event sources are **objects that generate events**.
- ✓ They are the entities or **components that trigger events** when **specific actions** or **conditions occur**.
- ✓ **Examples of event sources** include **buttons, text fields, mouse clicks**, or **keyboard inputs**.
- ✓ Event sources are **responsible for creating** and **dispatching** the corresponding **event objects** when the **specific event occurs**.

2. Event Listeners

- ✓ Event **listeners are interfaces** or classes **that define the methods to handle events**.
- ✓ They are **responsible for listening to events generated by event sources** and **invoking the appropriate event handlers** to respond to those events.
- ✓ Event **listeners implement the methods defined in the listener interface**, which **contain the logic for handling the events**.
- ✓ Event listeners are **registered** with the **event sources** to **receive and process the events**.

Registering the Source With Listener

- Different Classes provide different registration methods.

Syntax

```
addTypeListener()
```

- where Type represents the type of event.

Example 1:

- For KeyEvent we use addKeyListener() to register.

Example 2:

For ActionEvent we use addActionListener() to register.

Event Classes in Java

Event Class	Listener Interface	Description
ActionEvent	ActionListener	Represents an action, such as a button click, triggered by a GUI component.
MouseEvent	MouseListener	Represents mouse events like clicks, enters, exits, and button presses on a GUI component.
KeyEvent	KeyListener	Represents keyboard events, such as key presses and releases, from a GUI component.
WindowEvent	WindowListener	Represents window-related events, like opening, closing, or resizing a GUI window.
FocusEvent	FocusListener	Represents focus-related events, including gaining and losing focus on a GUI component.

Java Event Classes and Listener Interfaces

i) Event Classes

- ✓ ActionEvent: This represents the user's action, such as clicking a button or selecting a menu item.
- ✓ MouseEvent: Represents mouse-related events, such as mouse clicks, movement, or dragging.
- ✓ KeyEvent: Represents keyboard-related events, such as key presses or key releases.
- ✓ WindowEvent: Represents events related to windows or frames, such as window opening, closing, or resizing.
- ✓ FocusEvent: Represents events related to focus, such as when a component gains or loses focus.

ii) Listener Interfaces

- ✓ ActionListener: Defines methods to handle ActionEvents.
- ✓ MouseListener: Defines methods to handle MouseEvent.
- ✓ MouseMotionListener: Defines methods to handle mouse motion events.
- ✓ KeyListener: Defines methods to handle KeyEvent.
- ✓ WindowListener: Defines methods to handle WindowEvent.

Different interfaces consists of different methods

Listener Interface	Methods
ActionListener	<ul style="list-style-type: none">•actionPerformed()
ComponentListener	<ul style="list-style-type: none">•componentResized()•componentShown()•componentMoved()•componentHidden()
ItemListener	<ul style="list-style-type: none">•itemStateChanged()
KeyListener	<ul style="list-style-type: none">•keyTyped()•keyPressed()•keyReleased()
MouseListener	<ul style="list-style-type: none">•mousePressed()•mouseClicked()•mouseEntered()•mouseExited()•mouseReleased()
MouseMotionListener	<ul style="list-style-type: none">•mouseMoved()•mouseDragged()
MouseWheelListener	<ul style="list-style-type: none">•mouseWheelMoved()
TextListener	<ul style="list-style-type: none">•textChanged()

Flow of Event Handling

Step-1:

- ✓ User Interaction with a component is required to generate an event.

Step-2:

- ✓ The object of the respective event class is created automatically after event generation, and it holds all information of the event source.

Step-3:

- ✓ The newly created object is passed to the methods of the registered listener.

Step-4:

- ✓ The method executes and returns the result.

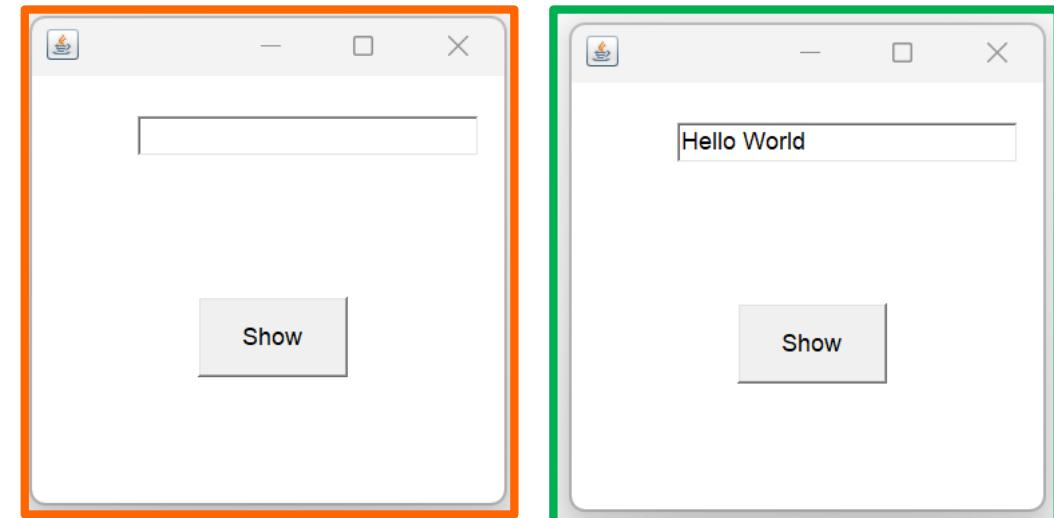
Simple Event Handling Program

```
import java.awt.*;
import java.awt.event.*;

class EventHandling extends Frame implements ActionListener
{
    EventHandling ()
    {
        TextField tf = new TextField ();
        tf.setBounds (60, 50, 170, 20);
        Button b = new Button ("Show");
        b.setBounds (90, 140, 75, 40);
        b.addActionListener (this);
        add (b);
        add (tf);
        setSize (250, 250);
        setLayout (null);
        setVisible (true);
    }
}
```

```
public void actionPerformed (ActionEvent e)
{
    tf.setText ("Hello World");
}

public static void main (String args[])
{
    EventHandling eh=new EventHandling ();
}
```



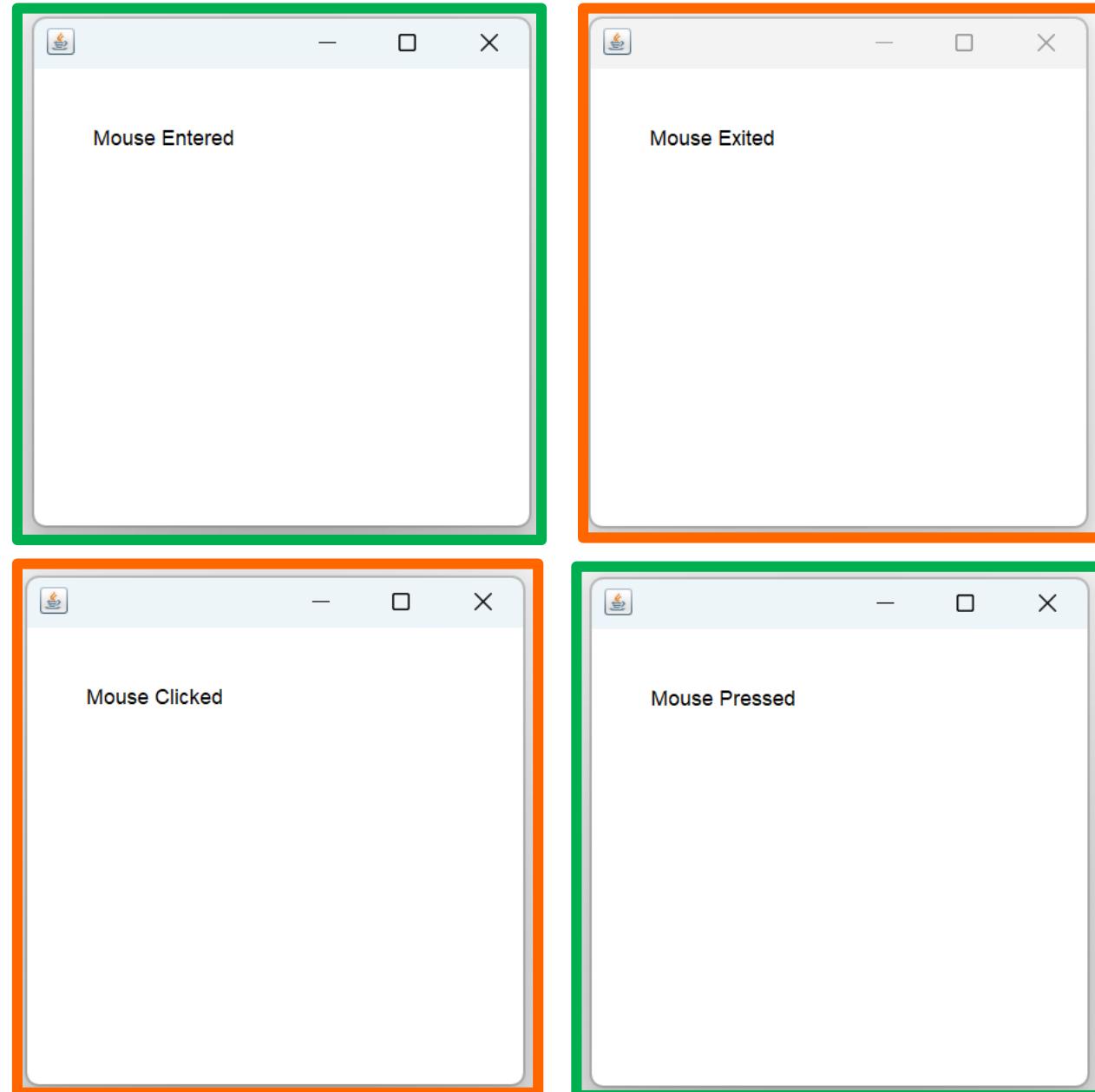


Mouse Event handling Program

```
/* Java Program to demonstrate the event actions associated with a mouse */
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class ML extends Frame implements MouseListener
{
    Label L;
    ML()
    {
        addMouseListener(this);
        L=new Label();
        L.setBounds(40,50,100,40);
        add(L);
        setSize(300,300);
        setLayout(null);
        setVisible(true);
    }
    public void mouseEntered(MouseEvent e)
    {
        L.setText("Mouse Entered");
    }
}
```

Mouse Event handling Program

```
public void mouseExited(MouseEvent e)
{
    L.setText("Mouse Exited");
}
public void mouseReleased(MouseEvent e)
{
    L.setText("Mouse Released");
}
public void mousePressed(MouseEvent e)
{
    L.setText("Mouse Pressed");
}
public void mouseClicked(MouseEvent e)
{
    L.setText("Mouse Clicked");
}
public static void main(String[] args)
{
    ML obj=new ML();
}
```



Key Event handling Program

```
import java.awt.*;
import java.awt.event.*;
class KeyDemo extends Frame implements KeyListener
{
    String msg = "";
    String msg1= "";
    int X = 140, Y = 180;
    KeyDemo(String name)
    {
        super(name);
        setForeground(Color.red);
        addKeyListener(this);
    }
}
```

```
public void keyPressed(KeyEvent ke)
{
    msg1= "Key Down";
    int key = ke.getKeyCode();
    switch(key)
    {
        case KeyEvent.VK_F1 : msg += "<F1>"; break;
        case KeyEvent.VK_F2 : msg += "<F2>"; break;
        case KeyEvent.VK_F3 : msg += "<F3>"; break;
        case KeyEvent.VK_PAGE_DOWN:msg += "<PgDn>";break;
        case KeyEvent.VK_PAGE_UP:msg += "<PgUp>";break;
        case KeyEvent.VK_LEFT:msg += "<Left Arrow>";break;
        case KeyEvent.VK_RIGHT:msg += "<Right Arrow>";break;
    }
    repaint();
}
```



Key Event handling Program

```
public void keyReleased(KeyEvent ke)
{
    msg1="Key released";
    repaint();
}
public void keyTyped(KeyEvent ke)
{
    msg += ke.getKeyChar(); //gets the char stroked
    repaint();
}
public void paint(Graphics g)
{
    Color c1 = new Color(123,50,89); //0 TO 255 rED, GREEN, BLUE
    g.setColor(c1);
    g.drawString(msg, X, Y);
    g.drawString(msg1,100,200);
}
```

```
public class KeyDemo1
{
    public static void main(String args[])
    {
        KeyDemo f = new KeyDemo("Key Events");
        f.setSize(300,400);
        f.setVisible(true);
    }
}
```

Adapter Classes

- Java adapter classes provide the default implementation of listener interfaces.
- If you inherit the adapter class, you will not be forced to provide the implementation of all the methods of listener interfaces.
- So it saves code.
- The adapter classes are found in `java.awt.event` and `javax.swing.event` packages.
- The Adapter classes with their corresponding listener interfaces are given below.

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





01
TOPIC **AWT Hierarchy, AWT controls**

02
TOPIC **Layout Managers- FlowLayout, BorderLayout**

03
TOPIC **GridLayout and CardLayout. Limitations of AWT**

04
TOPIC **Applets- Definition, Life Cycle and Execution**

05
TOPIC **SWINGS: JFrame, JPanel, JComponent**

06
TOPIC **JLabel and Image-Icon**

07
TOPIC **JTextField,JTabbedPane , Swing Buttons**

08
TOPIC **JScrollPane, JComboBox, JTable.**

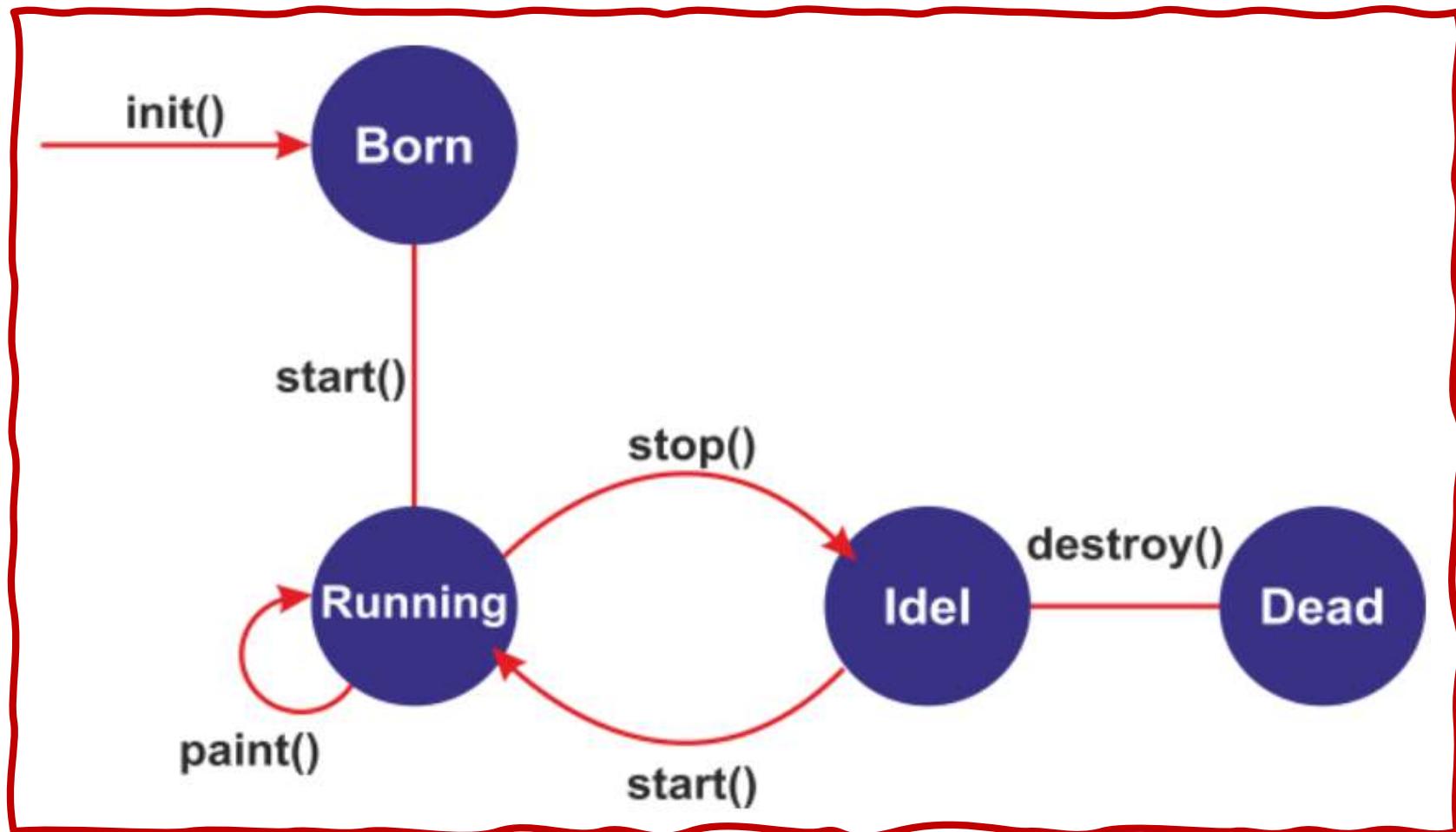
Applet

- An applet is a **special kind** of Java program that **runs** in a Java **enabled browser**.
- This is the **first Java program** that can **run over the network** using **the browser**. **Applet is** typically **embedded inside** a **web page** and **runs** in the **browser**.
- To **create an applet**, a class **must** class **extends Applet class**.
- An **Applet class does not** have any **main()** method.
- The **JVM** can use either a plug-in of the **Web browser** or a **separate runtime environment** to **run an applet application**.
- **JVM creates** an **instance** of the **applet class** and **invokes init() method** to **initialize an Applet**.
- **Every Applet application must import two packages** - **java.awt** and **java.applet**.
- **The class** in the **program** must be **declared as public**, because **it will be accessed by code** that is **outside the program**

How to run an Applet?

- There are two ways to run an applet
 - i. **By html file.**
 - ii. **By appletViewer tool.**

Applet Life Cycle



Applet Life Cycle

- The **Applet Life Cycle** in Java can be **defined as** the process of **how an applet object is created, started, stopped, and destroyed** during the entire **execution of the applet**.
- There are mainly **five methods used** in the **Applet Life Cycle** in Java namely,
 - i. **init()**
 - ii. **start()**
 - iii. **paint()**
 - iv. **stop()**
 - v. **destroy()**

i. **init() method:**

- ✓ It is **used for the initialization of the Applet** since no main() method is used.
- ✓ This init() method is **called only once** for **creating the applet**.
- ✓ All the **variables are initialized in this method**.

Applet Life Cycle

ii. start() Method:

- ✓ It is used for **starting the Applet in Java**. This method is **called after the init() method**.
- ✓ This method **contains the actual code** of the **applet**.
- ✓ The **start() method** is **invoked every time** the **browser is loaded or refreshed**.

iii. paint() Method:

- ✓ This step involves **drawing various shapes** in the applet using the **paint()** method.
- ✓ It **consists** of the **parameter of class Graphics**, which **helps** in enabling the **painting in an applet**.

iv. stop() method :

- ✓ **To stop an applet, we use the stop()** method of the **Applet class**.
- ✓ This **stop() method is invoked** when the **browser is minimized, restored, or moved to another tab**.

v. destroy() method:

- ✓ This method is called **when the applet is closed** or the **tab containing the webpage is closed**.
- ✓ This method **can only be called once**.

Applet

//Creating applet and run an applet using Java appletviewer

```
import java.applet.*;
import java.awt.*;
/*
<applet code="First.class" height="100" width="100">
</applet>
*/
public class First extends Applet
{
    public void init()
    {
        setBackground(Color.white);
        setForeground(Color.black);
    }
    public void paint(Graphics g)
    {
        g.drawString("Hello World", 300, 150);
    }
}
```

How To Run Applet

```
c:\>javac First.java
c:\>appletviewer First.java
```

OUTPUT



//Creating applet and run the applet using web browser

```
import java.applet.*;
import java.awt.*;
public class First extends Applet
{
    public void init()
    {
        setBackground(Color.white);
        setForeground(Color.black);
    }
    public void paint(Graphics g)
    {
        g.drawString("Hello World", 300, 150);
    }
}
```

//Creating html code

```
<html>
<body>
<applet code="First.class" width="150" height="25"></applet>
</body>
</html>
```

How To Run Applet:

Step-1.

Save html code as "First.html" then compile java code.

Step-2:

c:\>javac First.java

Step-3:

Now double click on First.html file

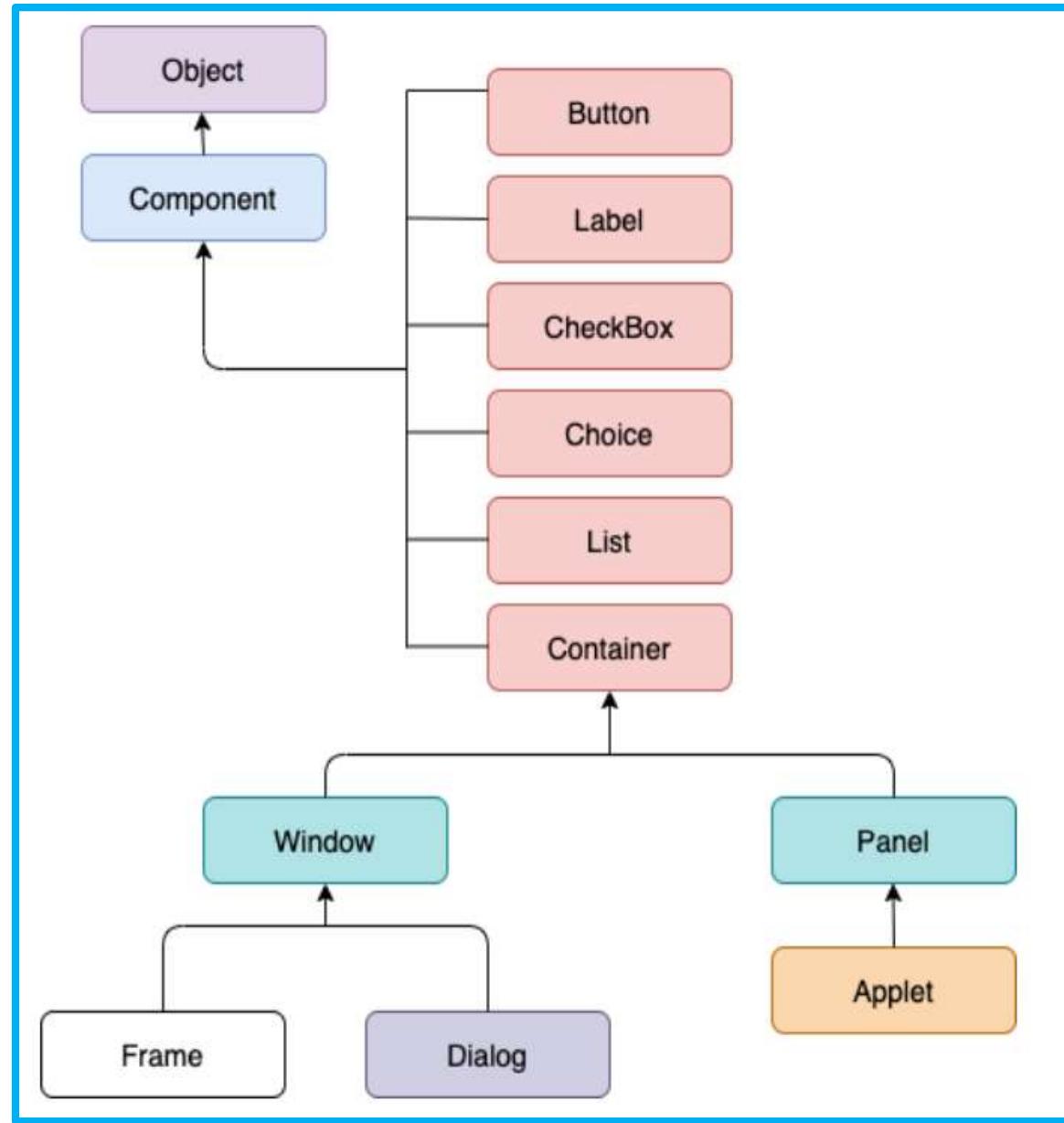
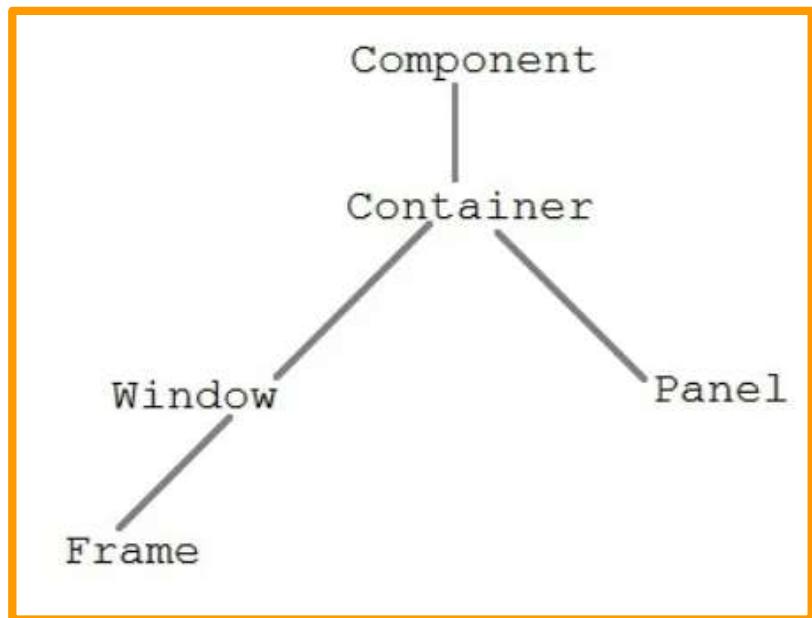
OUTPUT



AWT Hierarchy

- Java **Abstract Window Toolkit (AWT)** is an **API** that **contains large number** of **classes** and **methods** to **create** and **manage graphical user interface (GUI) applications**.
- The **AWT** was **designed to provide** a common **set of tools for GUI design** that could **work on a variety** of **platforms**.
- The **tools provided** by **the AWT** are **implemented** using each **platform's native GUI toolkit**, hence preserving the **look and feel** of each platform. This is an advantage of using AWT.
- But the **disadvantage** of such an approach is that **GUI designed on one platform** may **look different** when displayed **on another platform** that means AWT component are platform dependent.
- **AWT is the foundation** upon which Swing is made i.e **Swing is a improved GUI API that extends the AWT**.
- But **now a days AWT is merely used** because most **GUI Java programs are implemented using Swing**.

AWT Hierarchy



AWT Hierarchy

- The **hierarchy of Java AWT classes** are given above diagram, **all the classes** are **available** in **java.awt package**.

i. Component class:

- ✓ Component class is at the **top of AWT hierarchy**.
- ✓ **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**.

ii.Container:

- ✓ **Container** is a component in AWT that **contains** another **component like button, text field, tables** etc.
- ✓ It is a **subclass of component class**.
- ✓ It **keeps track of components** that are **added to another component**.

Heirarchy of component class

iii. Window class:

- ✓ The **window** is a **container** that **does not have borders** and **menu bars**.
- ✓ In order **to create a window**, you can use **frame or dialog**.

iv. Frame

- ✓ Frame is a **subclass of Window** that contain **title bar** and can have **menu bars**.
- ✓ It also **contain several different components** like **button, title bar, textfield, label** etc.
- ✓ **Most of the AWT applications** are created **using Frame window**.

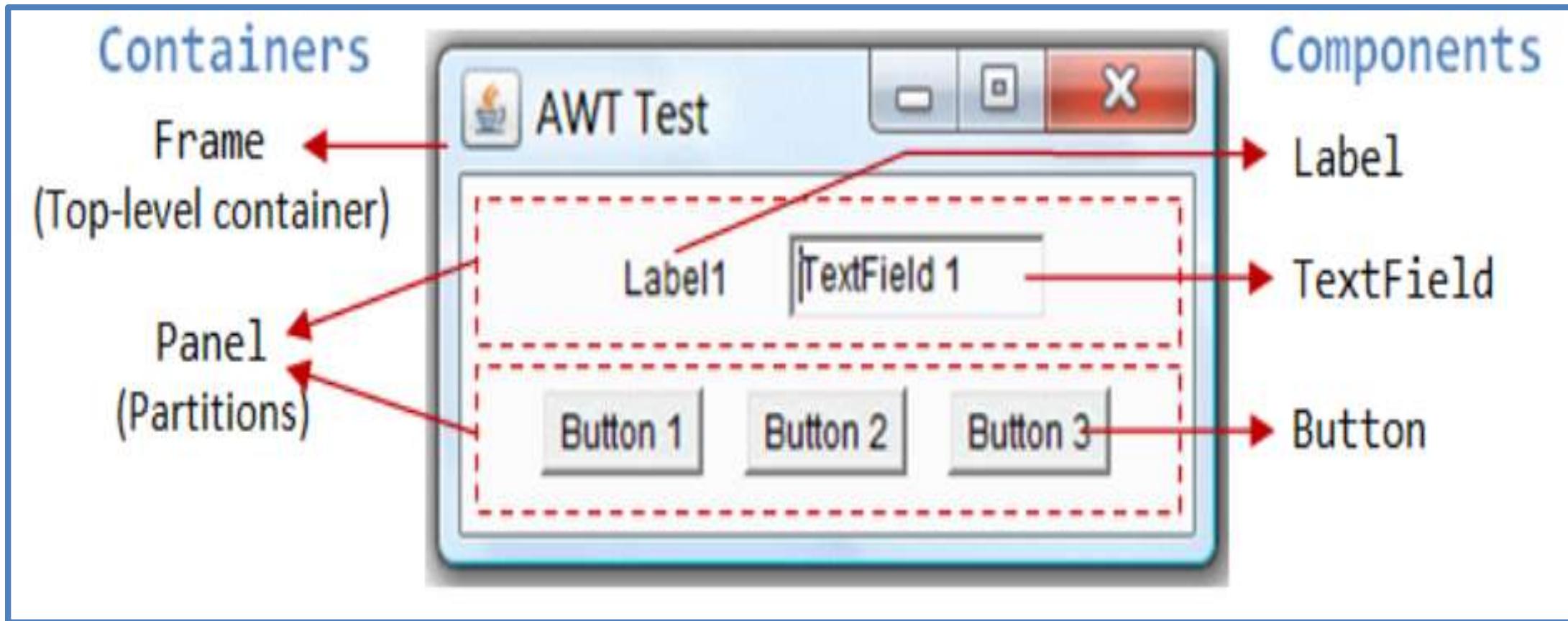
v. Dialog:

- ✓ It is the **container having a border and title**.

vi. Panel:

- ✓ It is container that is **used for holding components**.
- ✓ It is also **container** that **doesn't contain the title bar** and **menu bars**.

Hierarchy of component class



AWT controls

Frame class has two different constructors:

- a. Frame()
- b. Frame(String title)

Creating a Frame:

- There are **two ways to create a Frame**. They are,
 - i. **By Instantiating Frame class**
 - ii. **By extending Frame class**

Note:

- **While creating a frame** (either by instantiating or extending Frame class), Following **two attributes are must** for visibility of the frame:
 - i. **setSize(int width, int height)**
 - ii. **setVisible(true)**
- **When you create other components like Buttons, TextFields, etc.** Then **you need to add it** to the **frame** by using the method :

- **add(Component's Object)**

Creating Frame Window by Instantiating Frame class



//Creating Frame Window by Instantiating Frame class

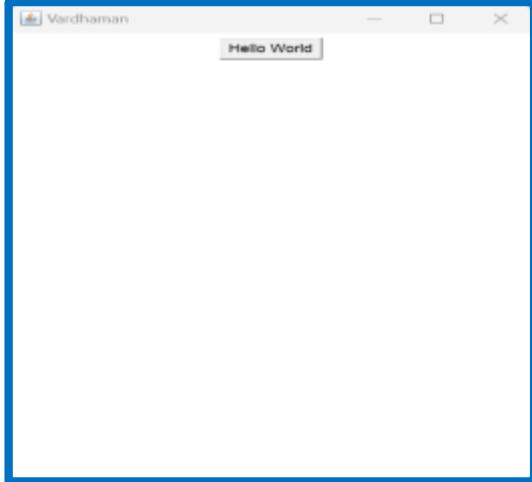
```
import java.awt.*;
public class Testawt
{
    Testawt()
    {
        Frame fm=new Frame();
        Label lb = new Label("welcome to java graphics");
        fm.add(lb);
        fm.setSize(300, 300);
        fm.setVisible(true);
    }
    public static void main(String args[])
    {
        Testawt ta = new Testawt();
    }
}
```



//Creating a frame
//Creating a label
//adding label to the frame
//setting frame size.
//set frame visibility true

Creating Frame Window by Extending Frame class

```
//Creating Frame window by extending Frame class
package testawt;
import java.awt.*;
import java.awt.event.*;
public class Testawt extends Frame
{
    Testawt()
    {
        Button btn=new Button("Hello World");
        add(btn);
        setSize(400, 500);
        setTitle("Vardhaman");
        setLayout(new FlowLayout());
        setVisible(true);
    }
    public static void main (String[] args)
    {
        Testawt ta = new Testawt();                //creating a frame.
    }
}
```



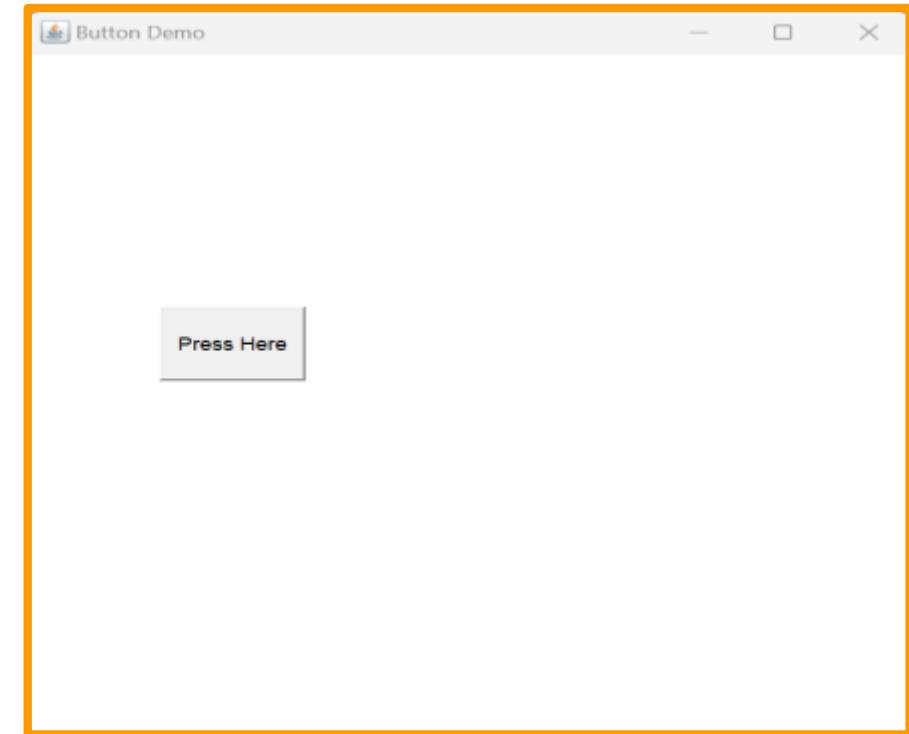
//adding a new Button.
//setting size.
//setting title.
//set default layout for frame.
//set frame visibility true.

i. Button Classs

- A **push button** is the **frequently found GUI control**.
- A **button can be created** by using the **Button class and its constructors**.
 - ✓ **Button()**
 - ✓ **Button(String str)**
- **Some of the methods available in the Button class** are as follows:
 - ✓ **void setLabel(String str)** - To set or assign the text to be displayed on the button.
 - ✓ **String getLabel()** - To retrieve the text on the button.
- When a **button is clicked**, it **generates an ActionEvent** which can be **handled using the ActionListener** interface and the event handling method is `actionPerformed()`.
- If there are **multiple buttons** we can get the **label of the button which was clicked** by using the method `getActionCommand()`.

Example to create a button

```
import java.awt.*;
public class ButtonDemo1
{
    public static void main(String[] args)
    {
        Frame f=new Frame("Button Demo");
        Button b1=new Button("Press Here");
        b1.setBounds(80,200,80,50);
        f.add(b1);
        f.setSize(500,500);
        f.setLayout(null);
        f.setVisible(true);
    }
}
```

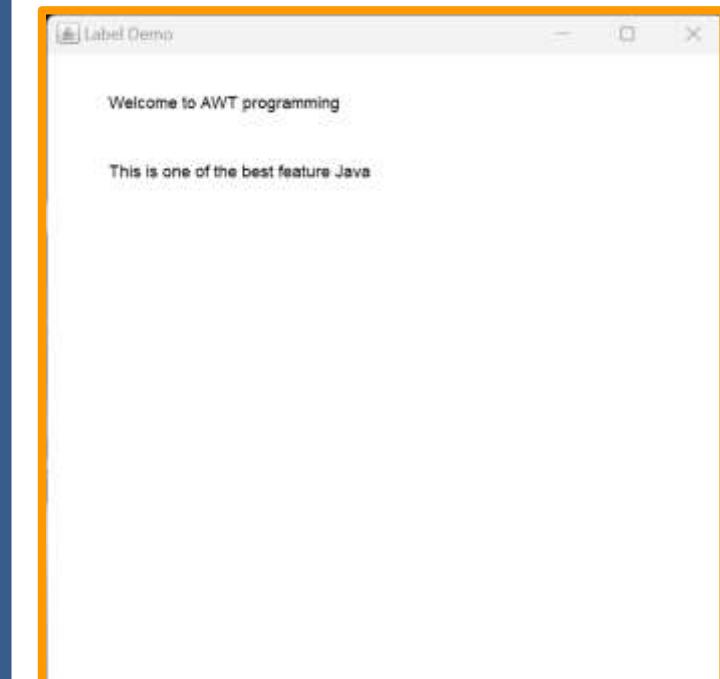


ii. Label

- It is used for **placing text in a container**. Only **Single line text** is **allowed** .
- A **label** is a GUI control which can be **used to display static text**.
- Label can be **created** using the **Label class** and its **constructors** which are listed below:
 - ✓ **Label()**
 - ✓ **Label(String str)**
 - ✓ **Label(String str, int how)**
- The parameter **how specifies the text alignment**. Valid values are **Label.LEFT**, **Label.CENTER** or **Label.RIGHT**
- Some of the **methods available** in the **Label class** are as follows:
 - i. **void setText(String str)** – To **set** or **assign text** to the label.
 - ii. **void setAlignment(int how)** – To **set the alignment of text** in a label.

Example to creating two labels to display text

```
import java.awt.*;
class LabelDemo1
{
    public static void main(String args[])
    {
        Frame f= new Frame("Label Demo");
        Label lab1=new Label("Welcome to AWT programming");
        lab1.setBounds(50,50,200,30);
        Label lab2=new Label("This is one of the best feature Java");
        lab2.setBounds(50,100,200,30);
        f.add(lab1);
        f.add(lab2);
        f.setSize(500,500);
        f.setLayout(null);
        f.setVisible(true);
    }
}
```



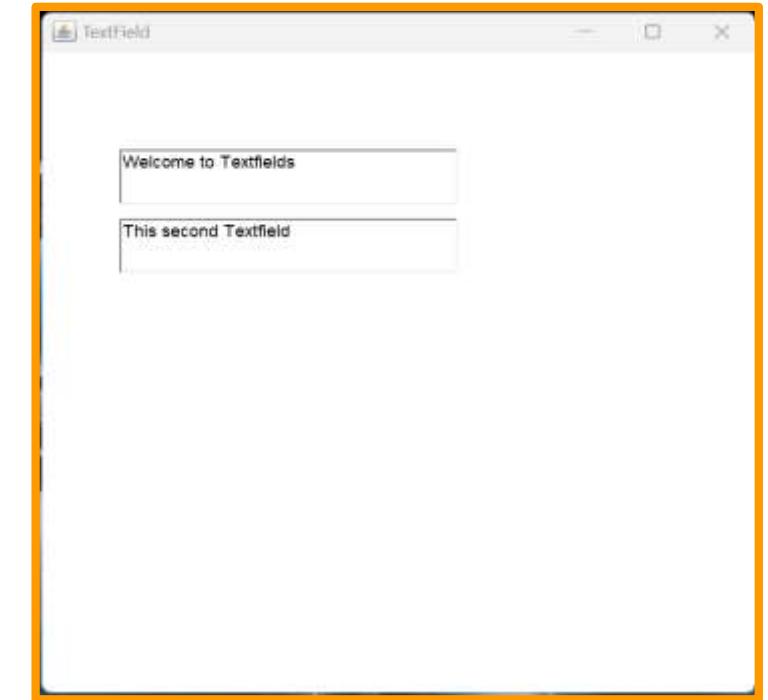
iii.TextField

A text field or **text box** is a single line text entry control which **allows the user to enter a single line of text.**

- A **text field can be created** using the `TextField` class along with **its following constructors:**
 - ✓ `TextField ()`
 - ✓ `TextField (int numChars)`
 - ✓ `TextField (String str)`
 - ✓ `TextField (String str, int numChars)`
- In the above constructors **numChars specifies** the **width of the text field**, and **str specifies the initial text** in the text field.

Example to creating two TextFields

```
import java.awt.*;  
class TextFieldDemo1  
{  
    public static void main(String args[])  
    {  
        Frame f= new Frame("TextField");  
        TextField t1=new TextField("Welcome to Textfields");  
        t1.setBounds(60,100, 230,40);  
        TextField t2=new TextField("This second Textfield");  
        t2.setBounds(60,150, 230,40);  
        f.add(t1);  
        f.add(t2);  
        f.setSize(500,500);  
        f.setLayout(null);  
        f.setVisible(true);  
    }  
}
```



iv.TextArea

- It is used for **displaying multiple-line text**.
- A **text area** is a multi-line text entry control in which **user can enter multiple lines of text**.
- A text area can be created using the following constructors:
 - ✓ `TextArea ()`
 - ✓ `TextArea (int numLines, int numChars)`
 - ✓ `TextArea (String str)`
 - ✓ `TextArea (String str, int numLines, int numChars)`
 - ✓ `TextArea (String str, int numLines, int numChars, int sBars)`
- In the above constructors,
 - **numLines specifies the height of the text area**,
 - **numChars specifies the width of the text area**,
 - **str specifies the initial text** in the text area
 - **sBars specifies the scroll bars**.

iv.TextArea

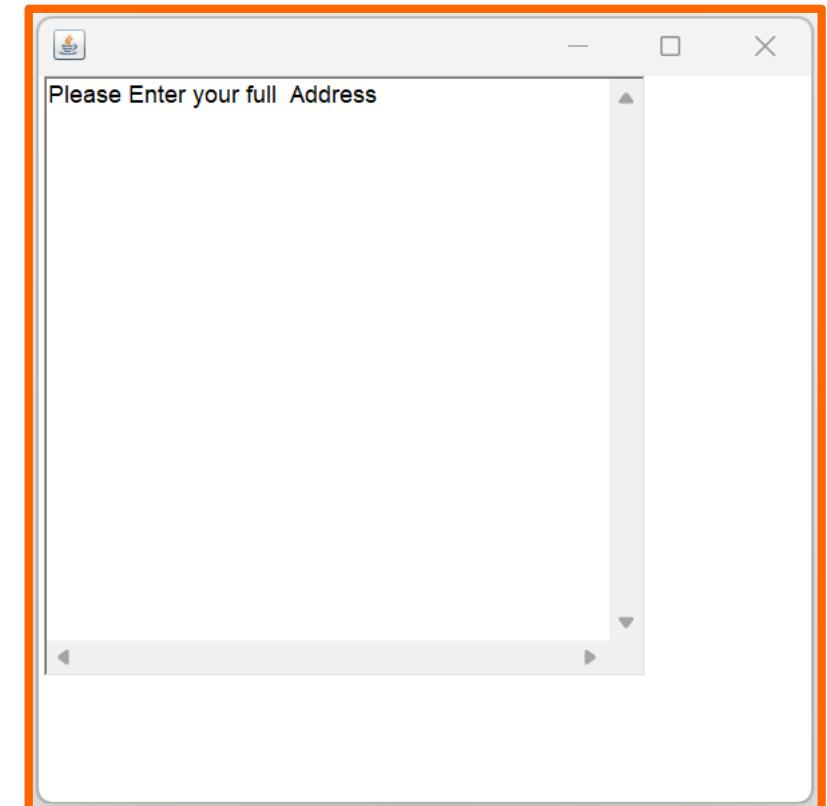
- ✓ SCROLLBARS_BOTH
- ✓ SCROLLBARS_NONE
- ✓ SCROLLBARS_HORIZONTAL_ONLY
- ✓ SCROLLBARS_VERTICAL_ONLY

■ Following are **some of the methods available** in the **TextArea class**:

1. **void setText(String str)** – To assign or set the text in a text area.
2. **void select(int startIndex, int endIndex)** – To select the text in text field from startIndex to endIndex – 1.
3. **void insert(String str, int index)** – To insert the given string at the specified index.
4. **void replaceRange(String str, int startIndex, int endIndex)** – To replace the text from startIndex to endIndex – 1 with the given string.

Example to creating TextArea

```
import java.awt.*;
public class TAExample
{
    TAExample()
    {
        Frame f = new Frame();
        TextArea ta= new TextArea("Please Enter your full Address");
        ta.setBounds(10, 30, 300, 300);
        f.add(ta);
        f.setSize(400, 400);
        f.setLayout(null);
        f.setVisible(true);
    }
    public static void main(String args[])
    {
        TAExample te=new TAExample();
    }
}
```

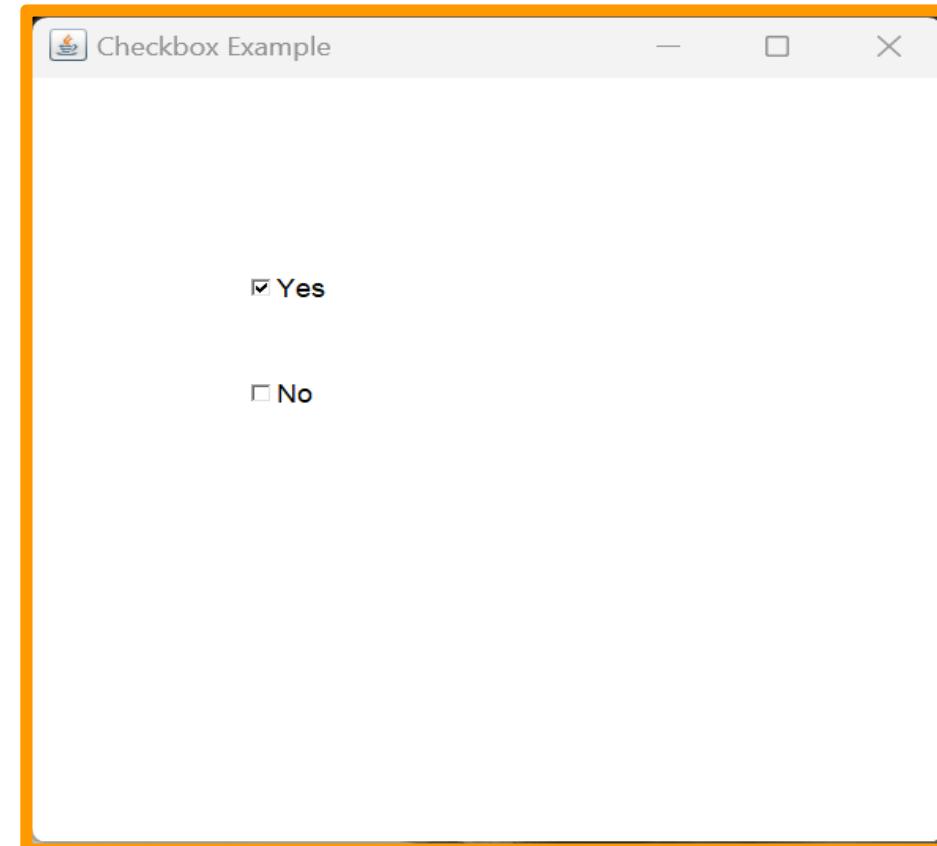


v.Checkbox

- It is **used when we want to select only one option** i.e true or false.
- When the **checkbox is checked then its state is "on" (true)** else it is "off"(false).
- A **checkbox** control can be **created using following constructors:**
 - **Checkbox()**
 - **Checkbox(String str)**
 - **Checkbox(String str, boolean b)**
- Following are various methods available in the Checkbox class:
 - **boolean getState()** – To retrieve the state of a checkbox.
 - **void setState(boolean on)** – To set the state of a checkbox.
 - **String getLabel()** – To retrieve the text of a checkbox.
 - **void setLabel(String str)** – To set the text of a checkbox.

Example to creating checkbox

```
import java.awt.*;
public class CheckboxDemo1
{
    CheckboxDemo1()
    {
        Frame f= new Frame("Checkbox Example");
        Checkbox c1 = new Checkbox("Yes", true);
        c1.setBounds(100,100, 60,60);
        Checkbox c2 = new Checkbox("No");
        c2.setBounds(100,150, 60,60);
        f.add(c1);
        f.add(c2);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
    public static void main(String args[])
    {
        CheckboxDemo1 c=new CheckboxDemo1();
    }
}
```



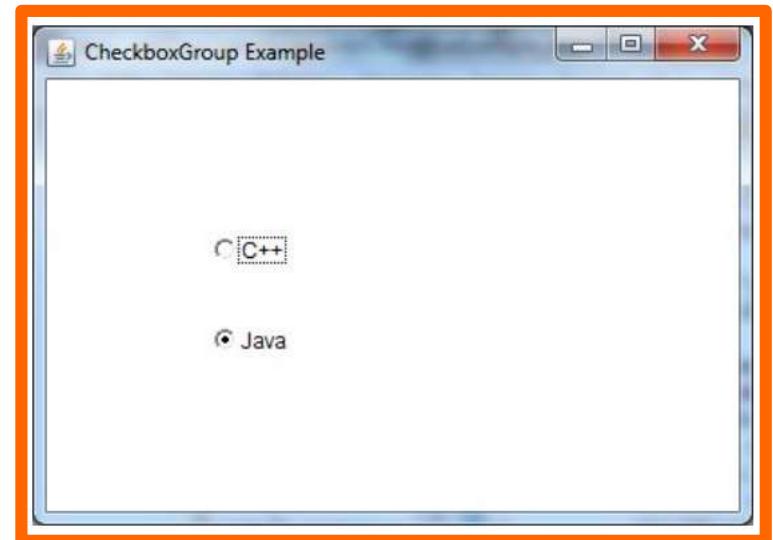
Vi. CheckboxGroup

- It is used to **group a set of Checkbox**.
- When **Checkboxes are grouped** then **only one box can be checked at a time**.
- In AWT, there is **no separate class for creating radio buttons**.
- The **difference between a checkbox** and **radio button** is, a **user can select one or more checkboxes**.
Whereas, a **user can select only one radio button in a group**.
- **Radio buttons** can be **create by CheckboxGroup**.

Example for creating CheckboxGroup

```
import java.awt.*;
public class CheckboxGroupExample
{
    CheckboxGroupExample()
    {
        Frame f= new Frame("CheckboxGroup Example");
CheckboxGroup cbg = new CheckboxGroup();
Checkbox c1 = new Checkbox("C++", cbg, false);
        c1.setBounds(100,100, 50,50);
Checkbox c2 = new Checkbox("Java", cbg, true);
        c2.setBounds(100,150, 50,50);
        f.add(c1);
        f.add(c2);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }

    public static void main(String args[])
    {
        CheckboxGroupExample c=new CheckboxGroupExample();
    }
}
```

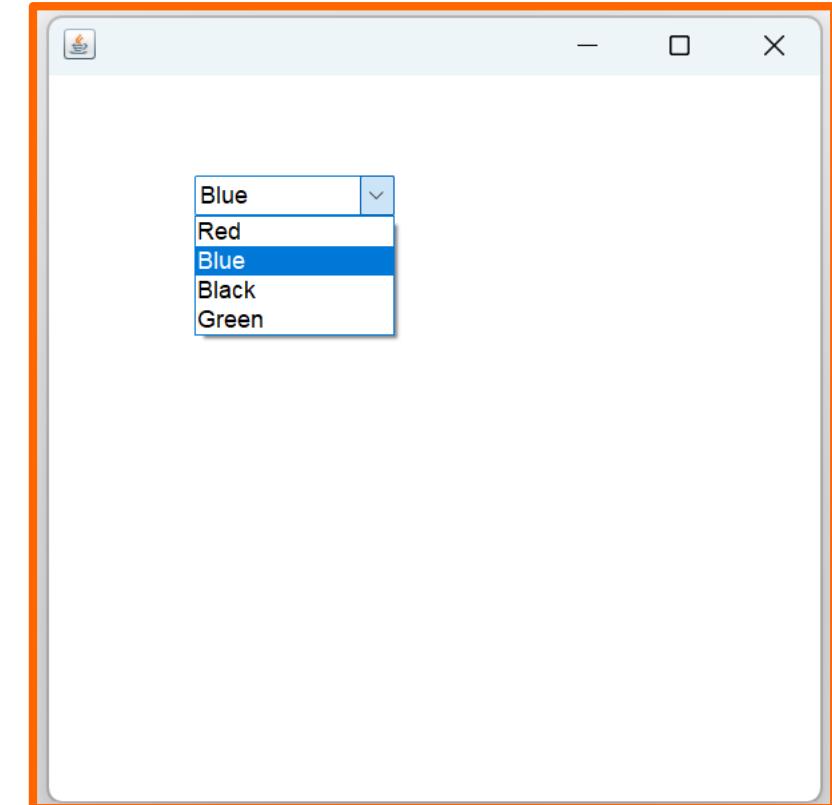


Vii.ChoiceboxGroup

- It is **used for creating** a **drop-down** menu of choices.
- When a **user selects a particular item** from the **drop-down then it is shown on the top of the menu.**
- When a **user clicks on a drop down box**, it **pops up a list of items** from which **user can select a single item.**
- Following are various methods available in Choice class:
 1. **void add(String name)** – To add an item to the drop down list.
 2. **int getItemCount()** – To retrieve the number of items in the drop down list.

Example for creating choiceBox

```
import java.awt.*;
public class ChoiceDemo
{
    ChoiceDemo()
    {
        Frame f= new Frame();
        Choice c=new Choice();
        c.setBounds(80,80, 100,100);
        c.add("Red");
        c.add("Blue");
        c.add("Black");
        c.add("Green");
        f.add(c);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
    public static void main(String args[])
    {
        ChoiceDemo c1= new ChoiceDemo();
    }
}
```



Java LayoutManagers

- The **LayoutManagers** are **used** to **arrange components** in a **particular manner**.
- It facilitates us **to control the positioning** and **size of the components** in **GUI** forms.
- **LayoutManager** is an **interface** that is **implemented** by **all the classes of layout managers**.
- There are the **four classes** that **represent** the **layout** managers:
 - i. **BorderLayout**
 - ii. **FlowLayout**
 - iii. **GridLayout**
 - iv. **CardLayout**

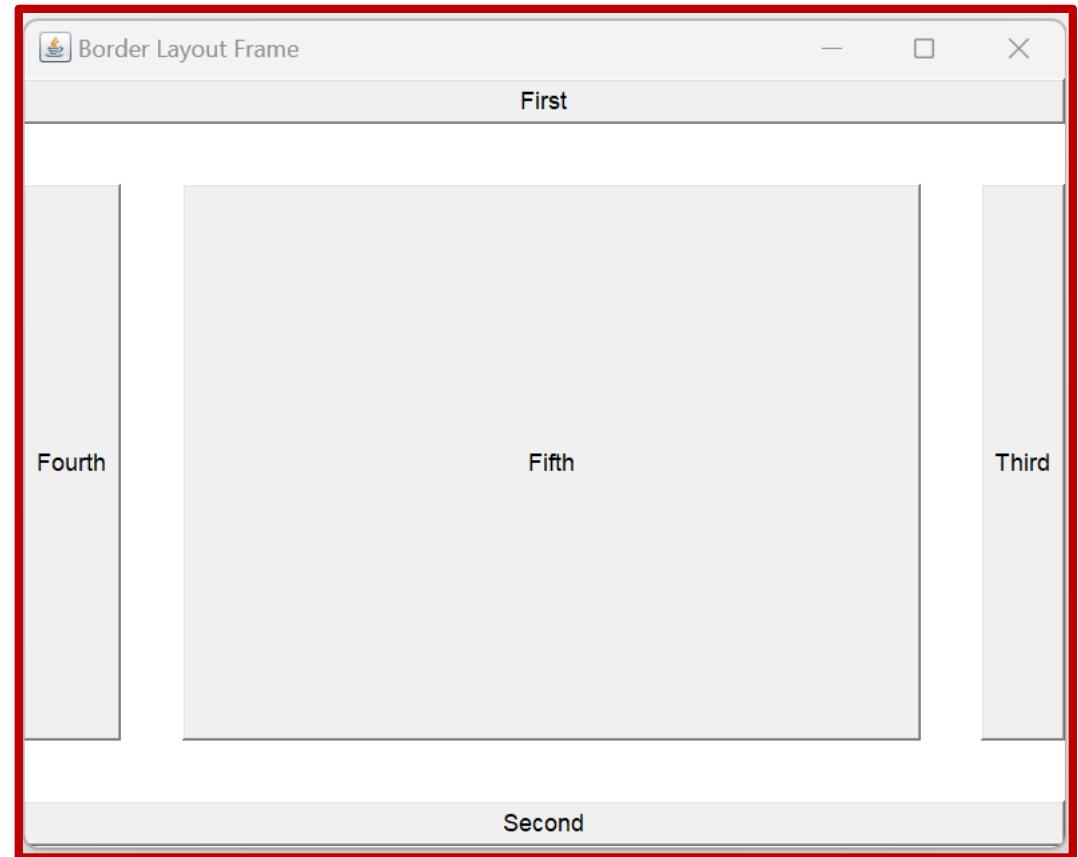
i. Border Layout

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

SNO	Method Name	Purpose
1	BorderLayout()	creates a border layout but with no gaps between the components.
2	BorderLayout(int hgap, int vgap)	creates a border layout with the given horizontal and vertical gaps between the components.

Border Layout Example Program

```
import java.awt.*;
public class BLExample
{
    public static void main(String[] args)
    {
        Frame f= new Frame("Border Layout Frame");
        Button b1= new Button("First");
        Button b2=new Button("Second");
        Button b3=new Button("Third");
        Button b4=new Button("Fourth");
        Button b5=new Button("Fifth");
f.setLayout(new BorderLayout(30,30));
        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);
    }
}
```



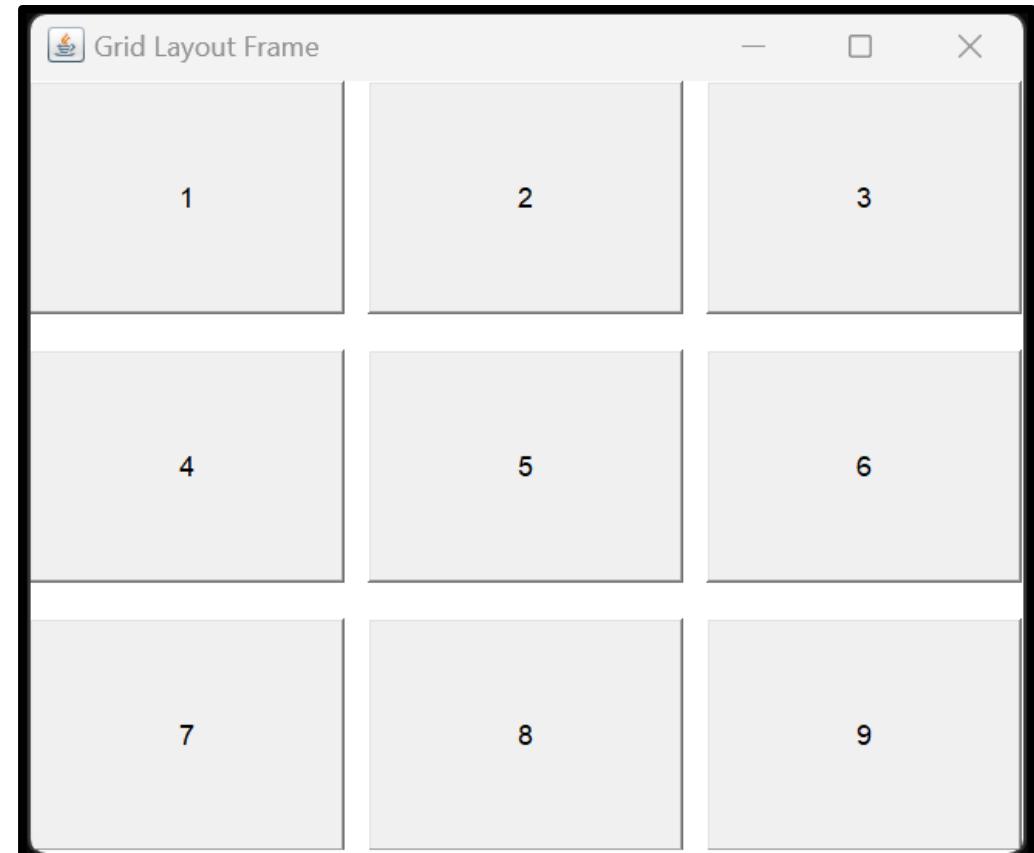
ii. Grid Layout

- It is used to arrange the components in a rectangular grid.
- One component is displayed in each rectangle.
- You start at row one, column one, then move across the row until it's full, then continue on to the next row.
- It is widely used for arranging components in rows and columns.
- The order of placement of components is directly dependant on the order in which they are added to the frame or panel.
- Constructors of GridLayout class:

SNO	Method Name	Purpose
1	GridLayout()	creates a grid layout with one column per component in a row.
2	GridLayout(int rows, int columns)	creates a grid layout with the given rows and columns but no gaps between the components.
3	GridLayout(int rows, int columns, int hgap, int vgap)	creates a grid layout with the given rows and columns along with given horizontal and vertical gaps.

Grid Layout Example Program

```
import java.awt.*;
public class GLEExample
{
    public static void main(String[] args)
    {
        Frame f= new Frame("Grid Layout Frame");
        Button b1= new Button("1");
        Button b2=new Button("2");
        Button b3=new Button("3");
        Button b4=new Button("4");
        Button b5=new Button("5");
        Button b6=new Button("6");
        Button b7=new Button("7");
        Button b8=new Button("8");
        Button b9=new Button("9");
f.setLayout(new GridLayout(3,3,10,15));
        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.setSize(300,300);
        f.setVisible(true);
    }
}
```



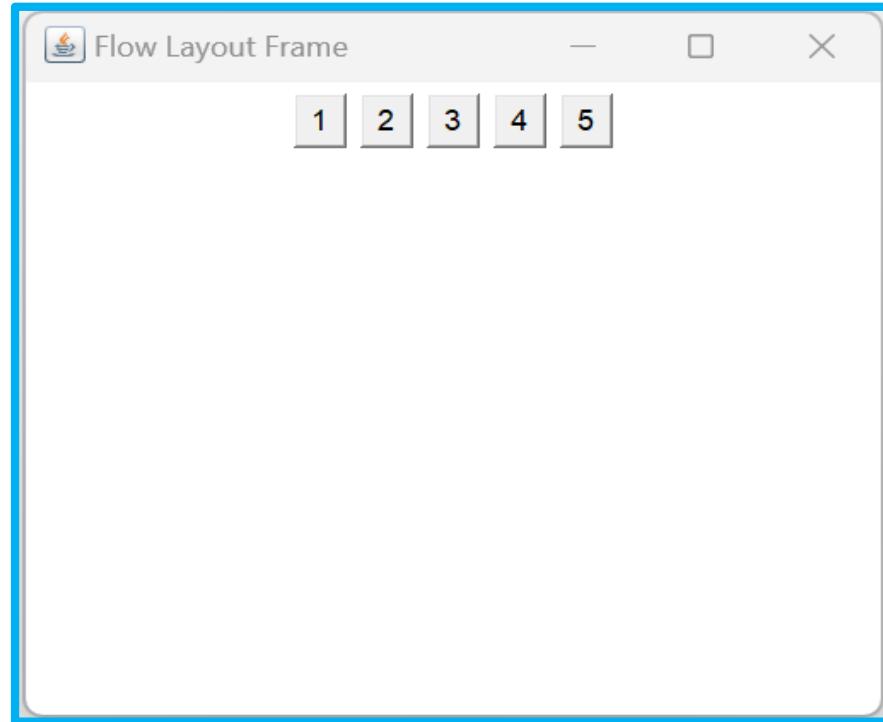
iii. FlowLayout

- **FlowLayout** class is used to arrange the components in a line, one after another (in a flow).
- It is the default layout of the applet or panel.
- It is basically helps develop more responsive UI and keep the components in a free flowing manner.
- When you add components to the screen, they move left to right based upon the order added.
- Constructors of FlowLayout class:

SNO	Method Name	Purpose
1	FlowLayout()	creates a flow layout with centered alignment and a default 5 unit horizontal and vertical gap.
2	FlowLayout(int align)	creates a flow layout with the given alignment and a default 5 unit horizontal and vertical gap.
3	FlowLayout(int align, int hgap, int vgap)	creates a flow layout with the given alignment and the given horizontal and vertical gap.

Flow Layout Example Program

```
import java.awt.*;
public class FLExample
{
    public static void main(String[] args)
    {
        Frame f= new Frame("Flow Layout Frame");
        Button b1= new Button("1");
        Button b2=new Button("2");
        Button b3=new Button("3");
        Button b4=new Button("4");
        Button b5=new Button("5");
f.setLayout(new FlowLayout());
        f.add(b1);
        f.add(b2);
        f.add(b3);
        f.add(b4);
        f.add(b5);
        f.setSize(300,300);
        f.setVisible(true);
    }
}
```



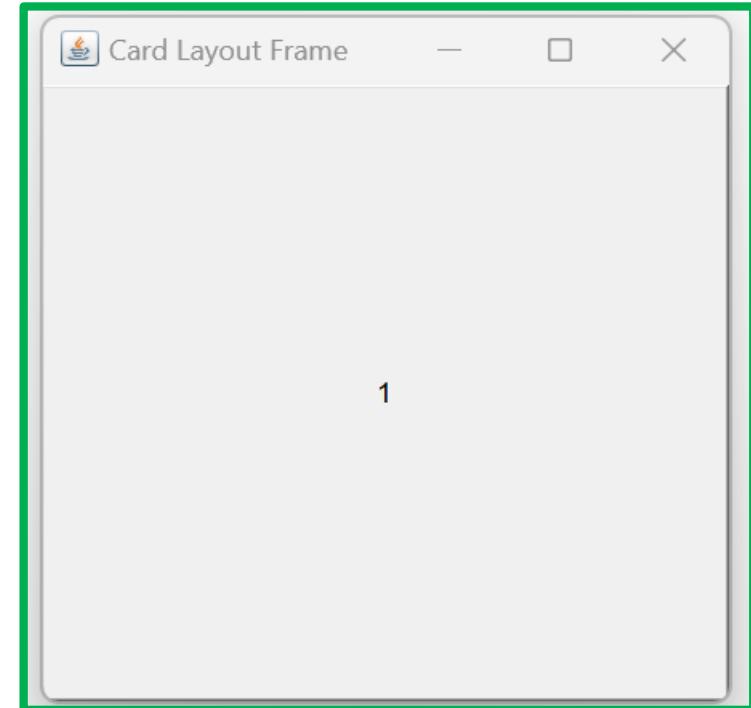
CardLayout

- CardLayout class manages the components in such a manner that only one component is visible at a time and hiding the rest.
- It treats each component as a card.
- It is rarely used and is utilized to stack up components one above another.
- Constructors of CardLayout Class

SNO	Method Name	Purpose
1	CardLayout()	creates a card layout with zero horizontal and vertical gap.
2	CardLayout(int hgap, int vgap)	creates a card layout with the given horizontal and vertical gap.

Card Layout Example Program

```
import java.awt.*;
public class CLExample
{
    public static void main(String[] args)
    {
        Frame f= new Frame("Card Layout Frame");
        Button b1= new Button("1");
        Button b2=new Button("2");
        Button b3=new Button("3");
        Button b4=new Button("4");
        Button b5=new Button("5");
        f.setLayout(new CardLayout());
        f.add(b1);
        f.add(b2);
        f.add(b3);
        f.add(b4);
        f.add(b5);
        f.setSize(300,300);
        f.setVisible(true);
    }
}
```



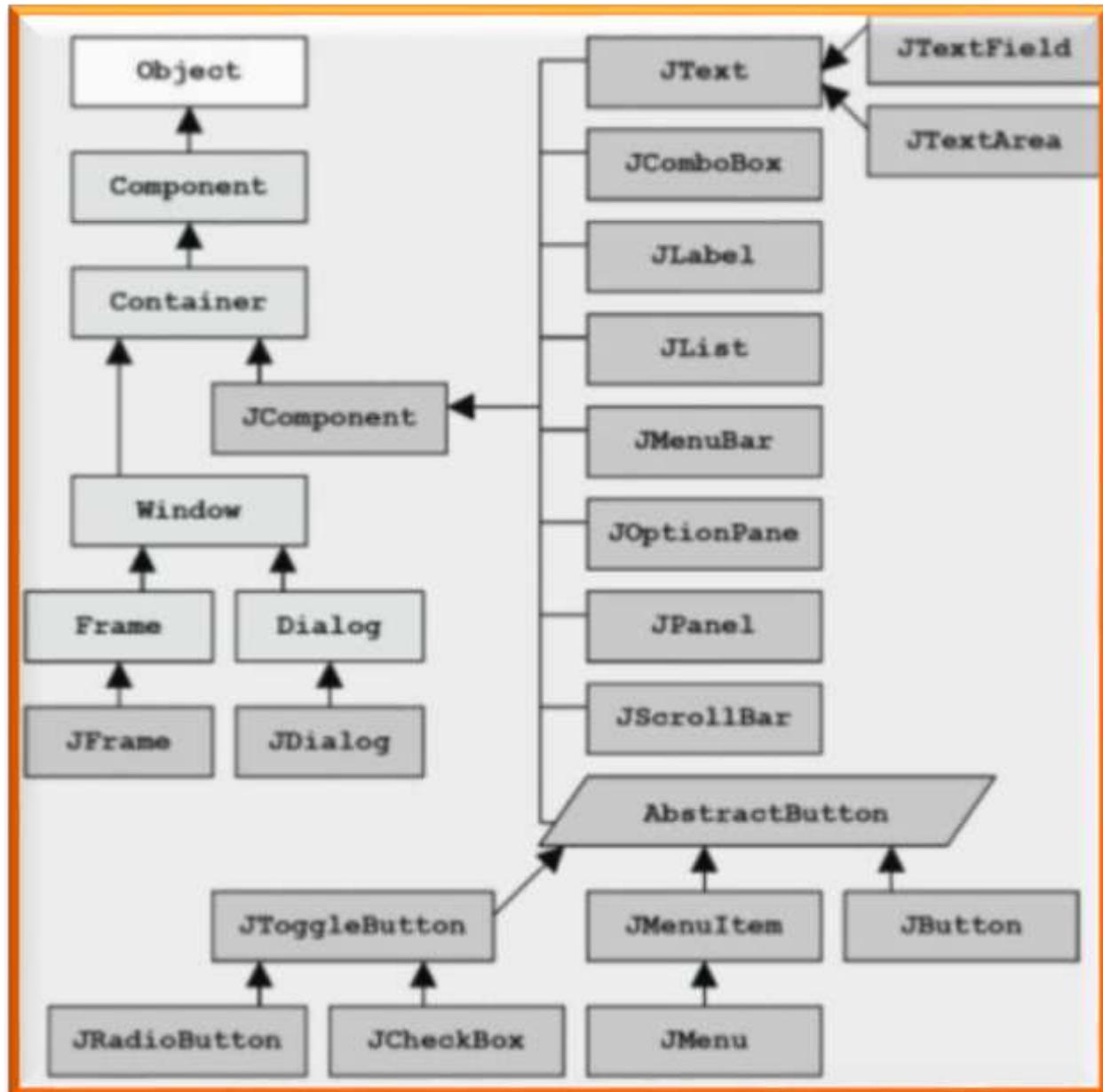
Swings

- Swing in Java is a **lightweight GUI toolkit**
- **Swing** in Java is a **Graphical User Interface (GUI) toolkit** that **includes the GUI components**.
- **Swing provides a rich set packages** to make **sophisticated GUI components** for Java applications.
- **Swing** is a **part of Java Foundation Classes(JFC)**, which is **an API for Java GUI**.

Difference Between AWT and Swing

AWT	SWING
• Platform Dependent	• Platform Independent
• Does not follow MVC	• Follows MVC
• Lesser Components	• More powerful components
• Does not support pluggable look and feel	• Supports pluggable look and feel
• Heavyweight	• Lightweight

Swings



Swings

JFrame:

- JFrame is a **top-level container** that **represents the main window** of a **GUI** application. It **provides a title bar, and minimizes, maximizes, and closes buttons.**

JPanel:

- JPanel is a **container** that can **hold other components**. It is **commonly used to group related components together.**

JButton:

- JButton is a **component** that **represents a clickable button**. It is **commonly used to trigger actions** in a **GUI** application.

JLabel:

- JLabel is a **component** that **displays text or an image**. It is commonly used to **provide information** or to label **other components**.



Swings

JTextField:

- JTextField is a component that **allows the user to input text**. It is commonly **used to get input** from the user, such as a name or an address.

JCheckBox:

- JCheckBox is a component that represents a checkbox. **It is commonly used to get a binary input** from the user, **such as whether or not to enable** a feature.

JList:

- JList is a component that **represents a list of elements**. It is typically **used to display a list of options** from which the **user can select one or more items**.

JTable:

- JTable is a component that **represents a data table**. It is **typically used to present data in a tabular fashion**, such as a list of products or a list of orders.

Swings

JScrollPane:

- ✓ JScrollPane is a **component** that provides **scrolling functionality** to **other components**.
- ✓ It is **commonly used** to **add scrolling to a panel or a table**.

Creating a JFrame

- There are two ways to create a JFrame Window.
 - i. By instantiating JFrame class.
 - ii. By extending JFrame class.

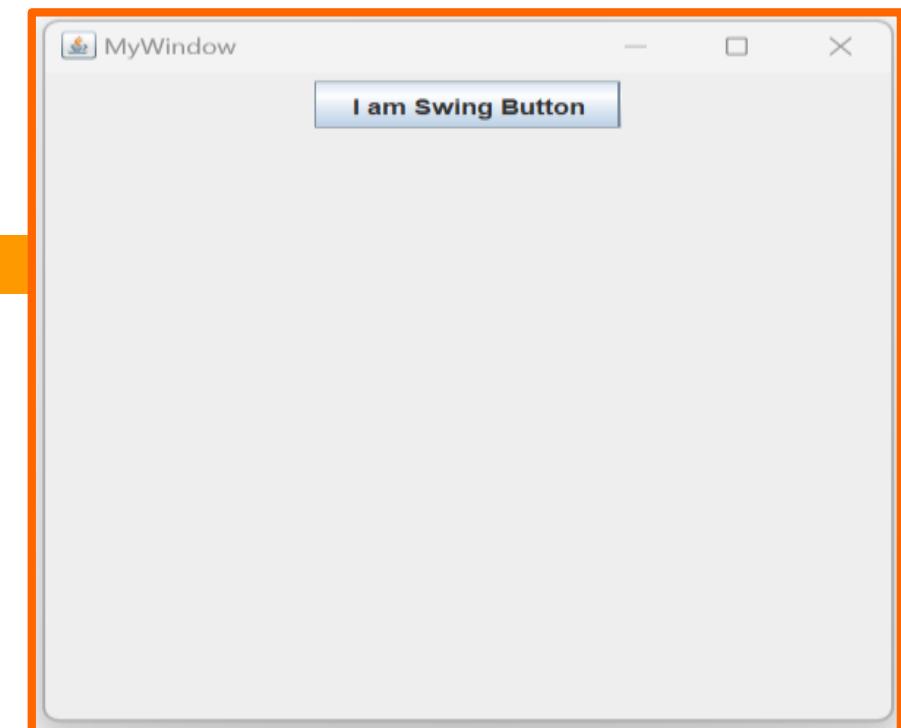
Creating JFrame Window by Instantiating JFrame class

//Creating JFrame Window by Instantiating JFrame class

```
import javax.swing.*;
import java.awt.*;
public class First
{
    First()
    {
        JFrame jf = new JFrame("MyWindow");
        JButton btn = new JButton("I am Swing Button");

        jf.add(btn);
        jf.setLayout(new FlowLayout());
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setSize(400, 400);

        jf.setVisible(true);
    }
    public static void main(String[] args)
    {
        First nf=new First();
    }
}
```

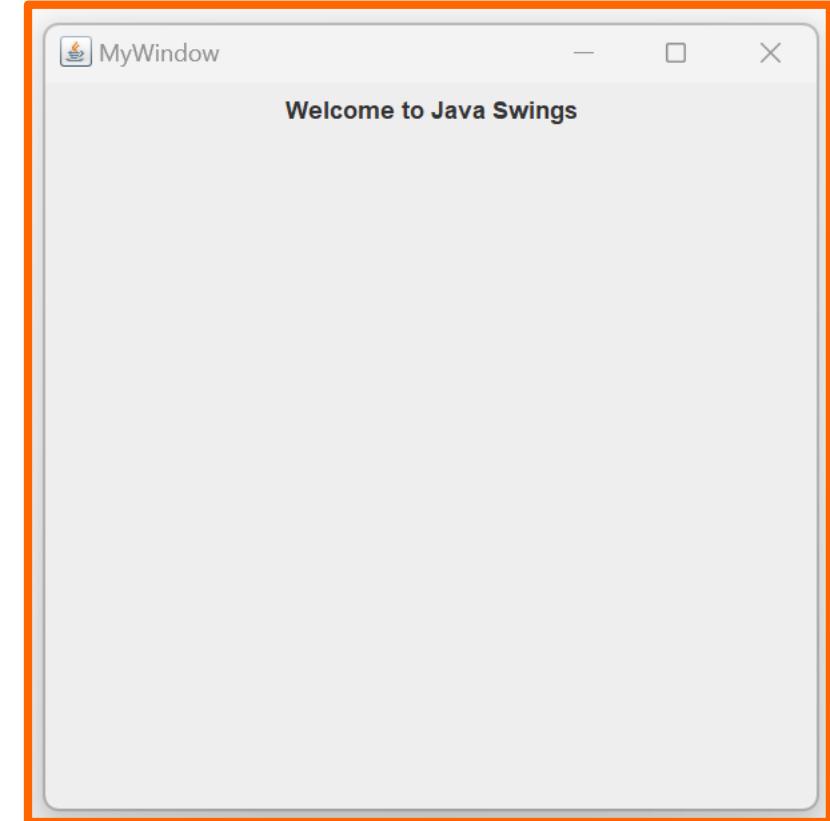


Creating JFrame Window by Extending JFrame class



```
//Creating JFrame window by extending JFrame class
import javax.swing.*; //importing swing package
import java.awt.*; //importing awt package
public class Second extends JFrame
{
    Second()
    {
        setTitle("MyWindow");
        JLabel lb = new JLabel("Welcome to Java Swings");
        add(lb);
        setLayout(new FlowLayout());
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(400, 400);
        setVisible(true);
    }

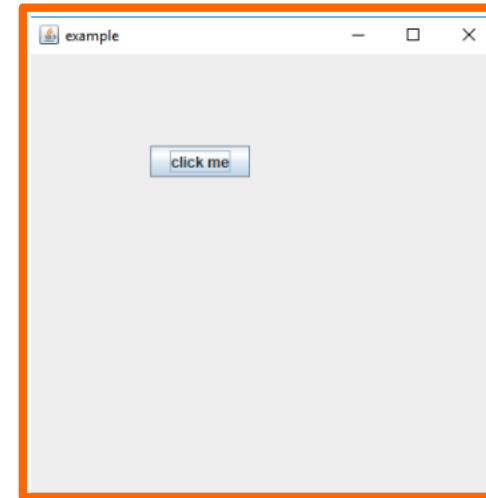
    public static void main(String[] args)
    {
        Second sf=new Second();
    }
}
```



JPanel Class

It inherits the JComponent class and **provides space for an application** which **can attach any other component**.

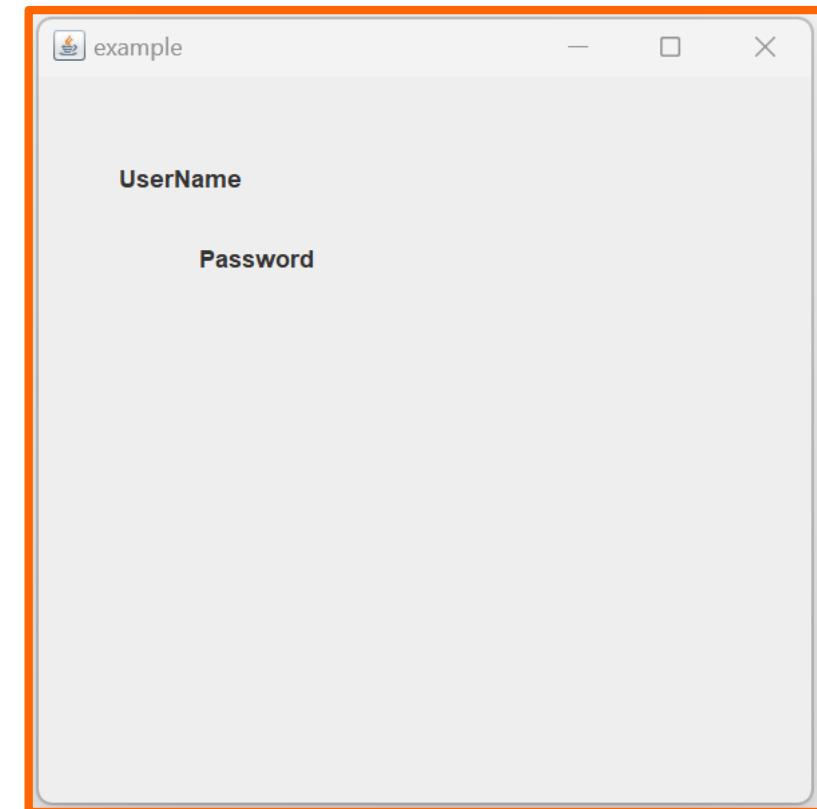
```
import java.awt.*;
import javax.swing.*;
public class Example
{
    Example()
    {
        JFrame jf = new JFrame("example");
        JPanel p = new JPanel();
        p.setBounds(40,70,200,200);
        JButton b = new JButton("click me");
        b.setBounds(60,50,80,40);
        p.add(b);
        jf.add(p);
        jf.setSize(400,400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setLayout(null);
        jf.setVisible(true);
    }
    public static void main(String args[])
    {
        new Example();
    }
}
```



JLabel

- It is used for **placing text** in a **container**.
- Only **Single line text is allowed** and the text **can not be changed**

```
import javax.swing.*;  
public class JLabExample  
{  
    public static void main(String args[])  
    {  
        JFrame a = new JFrame("example");  
        JLabel L1= new JLabel("UserName");  
        JLabel L2= new JLabel("Password");  
        L1.setBounds(40,40,90,20);  
        a.add(L1);  
        L2.setBounds(80,80,90,20);  
        a.add(L2);  
        a.setSize(400,400);  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        a.setLayout(null);  
        a.setVisible(true);  
    }  
}
```



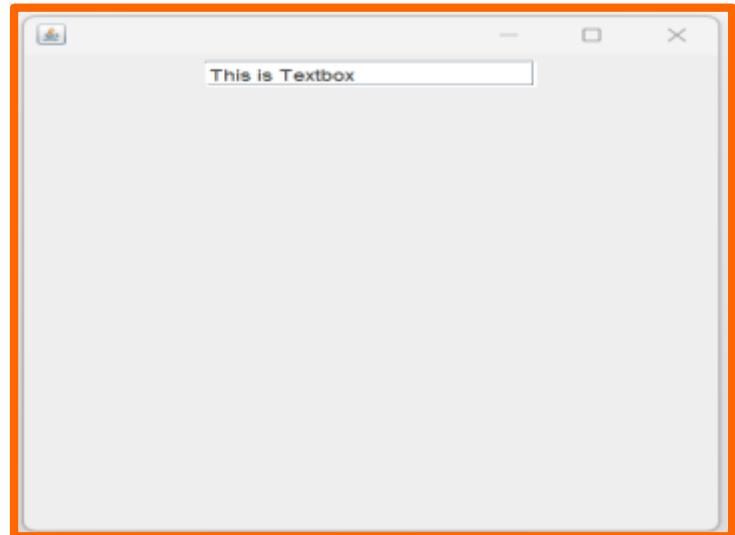
JTextField

- **JTextField** is used for taking input of single line of text.

It is most widely used text component.

- ✓ **JTextField(int cols)**
- ✓ **JTextField(String str, int cols)**
- ✓ **JTextField(String str)**

- *cols* represent the number of columns in text field.
- It is used to allow editing of single line text.

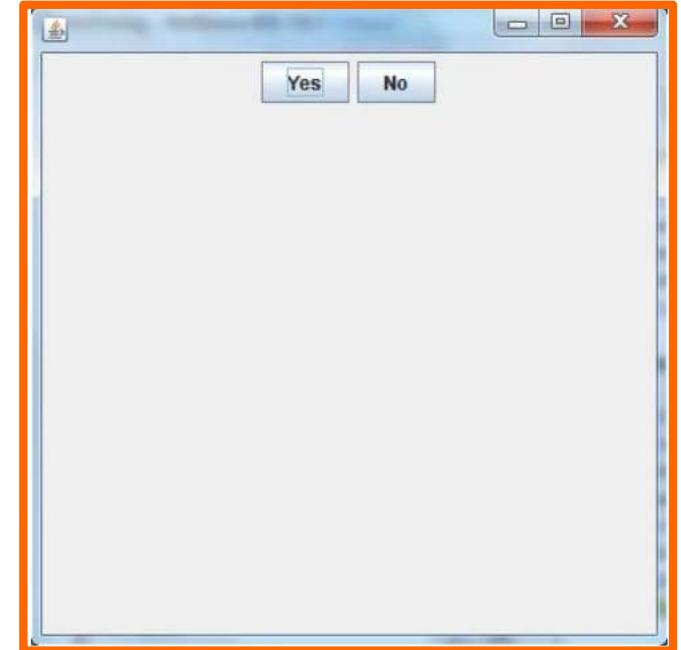


```
import javax.swing.*;  
import java.awt.event.*;  
import java.awt.*;  
public class MyTextField  
{  
    public MyTextField()  
{  
        JFrame jf = new JFrame();  
        JTextField jtf = new JTextField("This is Textbox",20);  
        jf.add(jtf);  
        jf.setLayout(new FlowLayout());  
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        jf.setSize(400, 400);  
        jf.setVisible(true);  
    }  
    public static void main(String[] args)  
{  
        new MyTextField();  
    }  
}
```

JButton

JButton class **provides functionality of a button**. It is **used to create button component**.

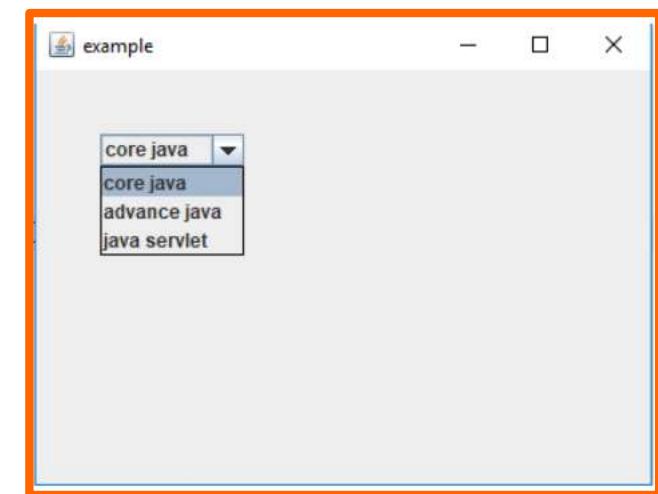
```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class testswing
{
    testswing()
    {
        JFrame jf = new JFrame("example");
        JButton bt1 = new JButton("Yes");
        JButton bt2 = new JButton("No");
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
        jf.setLayout(new FlowLayout());
        jf.setSize(400, 400);
        jf.add(bt1);
        jf.add(bt2);
        setVisible(true);
    }
    public static void main(String[] args)
    {
        testswing ts=new testswing();
    }
}
```



JComboBox

- It inherits the JComponent class and is used to show pop up menu of choices.
- Combo box is a combination of text fields and drop-down list.

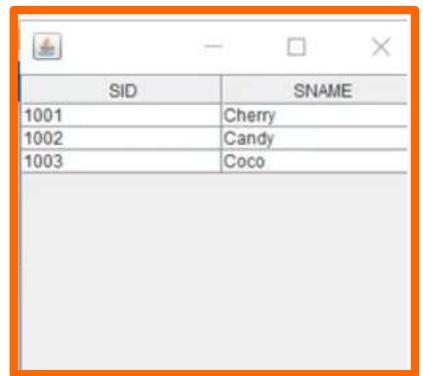
```
import javax.swing.*;  
public class Example  
{  
    Example()  
    {  
        JFrame a = new JFrame("example");  
        string courses[] = { "core java", "advance java", "java servlet" };  
        JComboBox c = new JComboBox(courses);  
        c.setBounds(40,40,90,20);  
        a.add(c);  
        a.setSize(400,400);  
        a.setLayout(null);  
        a.setVisible(true);  
    }  
    public static void main(String args[])  
    {  
        Example e=new Example();  
    }  
}
```



JTable

- It used to draw a table to display data.
- The **JTable** Contains 2 constructors:
 - i. **JTable()**
 - ii. **JTable(Object[][] rows, Object[] columns)**

```
import javax.swing.*;  
public class STableDemo1  
{  
    STableDemo1()  
    {  
        JFrame jf=new JFrame();  
        String table_data[][]={{ "1001","Cherry"}, {"1002","Candy"}, {"1003","Coco"}};  
        String table_column[]={ "SID", "SNAME" };  
        JTable jt=new JTable(table_data,table_column);  
        jt.setBounds(30,40,200,300);  
        JScrollPane tsp=new JScrollPane(jt);  
        jf.add(tsp);  
        jf.setSize(300,400);  
        jf.setVisible(true);  
    }  
    public static void main(String[] args)  
    {  
        new STableDemo1();  
    }  
}
```

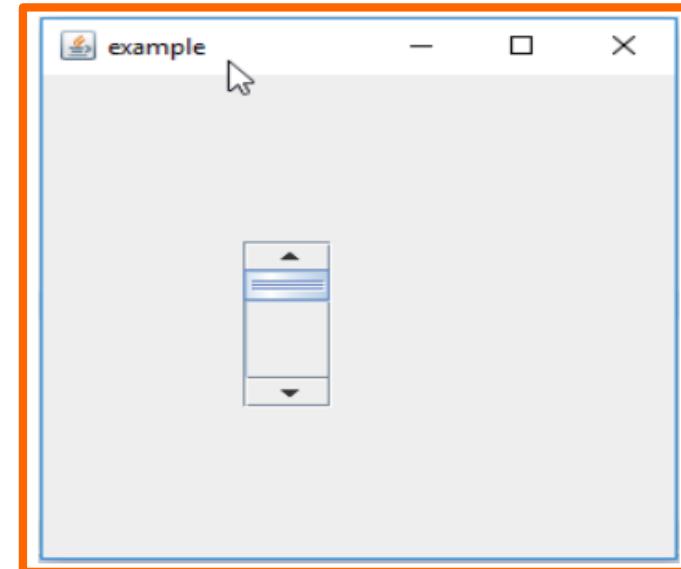


SID	SNAME
1001	Cherry
1002	Candy
1003	Coco

JScrollBar

- It is used to **add scroll bar, both horizontal and vertical.**

```
import javax.swing.*;  
class Example  
{  
    Example()  
    {  
        JFrame a = new JFrame("example");  
        JScrollBar b = new JScrollBar();  
        b.setBounds(90,90,40,90);  
        a.add(b);  
        a.setSize(300,300);  
        a.setLayout(null);  
        a.setVisible(true);  
    }  
    public static void main(String args[])  
    {  
        Example e=new Example();  
    }  
}
```

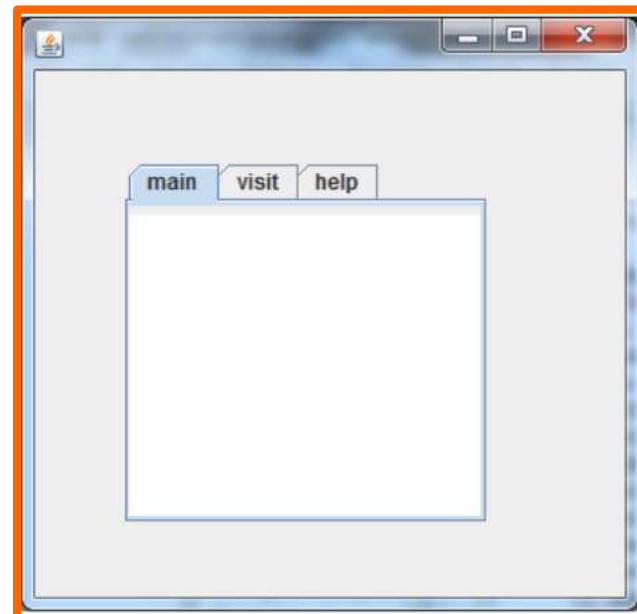


JTabbedPane

- It is used to switch between a group of components by clicking on a tab with a given title or icon.

```
import javax.swing.*;  
public class TabbedPaneExample  
{  
    TabbedPaneExample()  
{  
        JFrame 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.addTab("main",p1);  
        tp.addTab("visit",p2);  
        tp.addTab("help",p3);  
        f.add(tp);  
        f.setSize(400,400);  
        f.setLayout(null);  
        f.setVisible(true);  
    }  
}
```

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

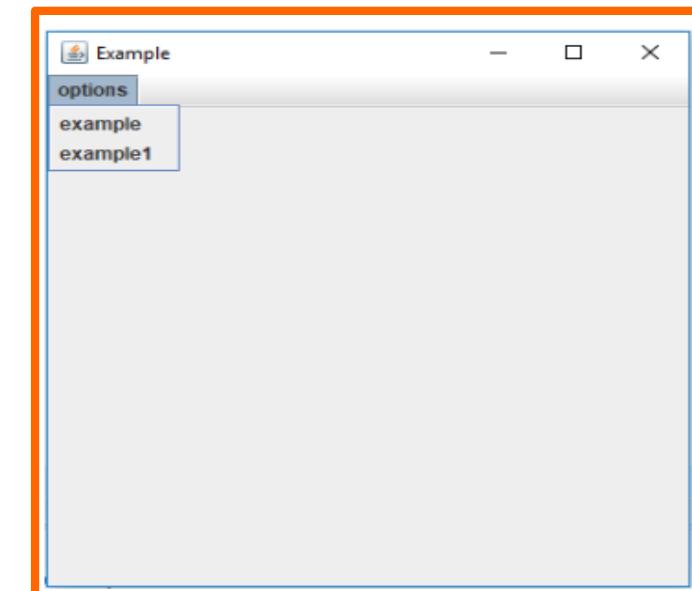


JMenu

- It inherits the JMenuItem class, and is a **pull down menu component** which is **displayed from the menu bar**.

```
import javax.swing.*;  
class Example  
{  
  
    Example()  
{  
        JFrame a = new JFrame("Example");  
        JMenu m = new JMenu("options");  
        JMenuBar mb = new JMenuBar();  
        JMenuItem a1 = new JMenuItem("example");  
        JMenuItem a2 = new JMenuItem("example1");  
        m.add(a1);  
        m.add(a2);  
        mb.add(m);  
        a.setJMenuBar(mb);  
        a.setSize(400,400);  
        a.setLayout(null);  
        a.setVisible(true);  
    }  
}
```

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



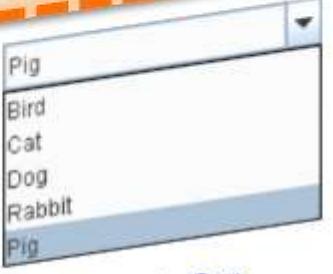
A Visual Guide to Swing Components



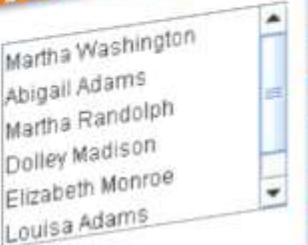
JButton



JCheckBox



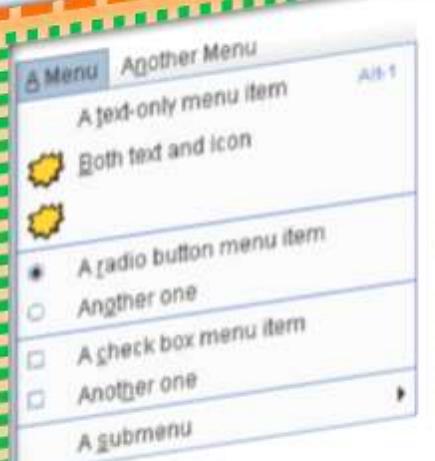
JComboBox



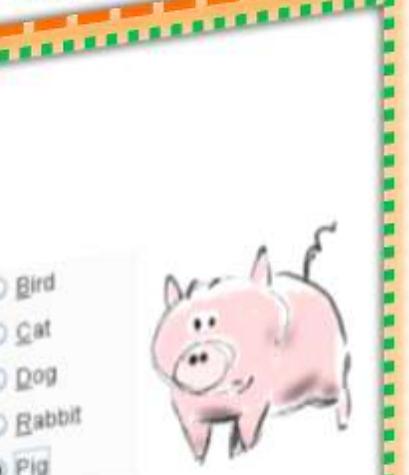
JList



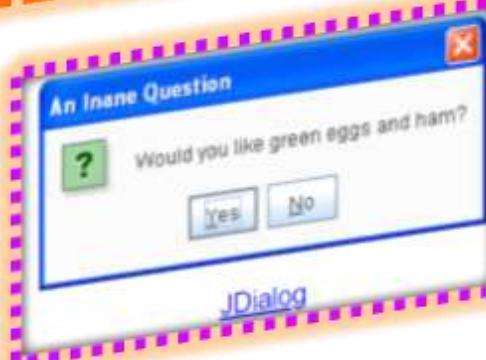
JScrollPane



JMenu



JRadioButton



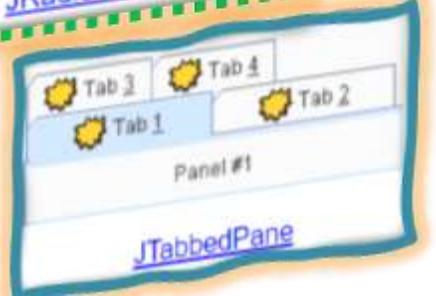
JDialog



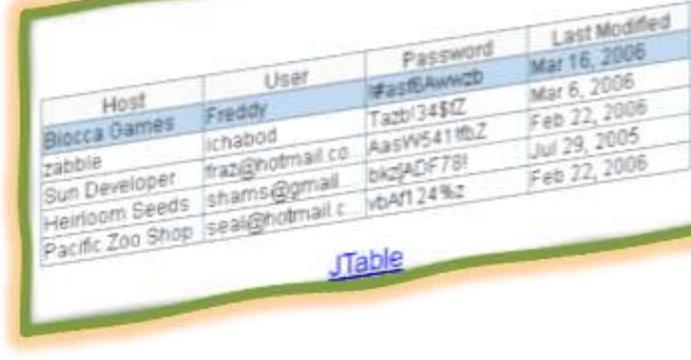
JTextField



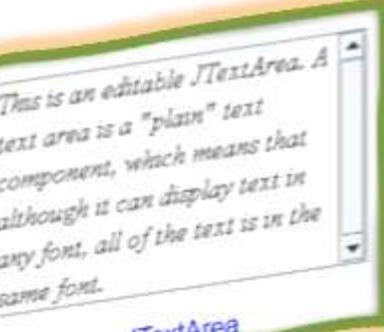
JPasswordField



JTabbedPane



JTable



JTextArea

This is an editable JTextArea. A text area is a "plain" text component, which means that although it can display text in any font, all of the text is in the same font.

Host	User	Password	Last Modified
Blocca Games	Freddy	#astfUwicb	Mar 16, 2006
zabbie	Ichabod	Tazbl34\$1Z	Mar 6, 2006
Sun Developer	frazij@hotmail.co	AasW541tbZ	Feb 22, 2006
Heirloom Seeds	shams@gmail.c	bkgADF78!	Jul 29, 2005
Pacific Zoo Shop	sean@hotmail.c	vbAf124%z	Feb 22, 2006



