I E S COLLEGE OF ENGINEERING

CHITTILAPPILLY, THRISSUR.

Kerala – 680 551

PH: (91) 0487 2309966, 2309967

www.iesce.org

E-mail: info@iescollegeofengineering.org

(Approved by AICTE & Affiliated to APJ Abdul Kalam Technological University.)

NAAC
NATIONAL ASSESSMENT AND ACCREDITATION COUNCIL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (DATA SCIENCE)

PBCST304 OBJECT ORIENTED PROGRAMMING (2024 SCHEME)

PRIYA P P

SEMESTER S3

OBJECT ORIENTED PROGRAMMING

(Common to CS/CA/CD/AM/CB/CN/CU/CG)

Course Code	**PBCST304**	**CIE Marks**	60
Teaching Hours/Week (L:T:P:R)	3:0:0:1	**ESE Marks**	40
Credits	4	**Exam Hours**	2 Hrs. 30 Min.
Prerequisites (if any)	None	**Course Type**	Theory

Course Objectives:

1. To teach the core object-oriented principles such as abstraction, encapsulation, inheritance, and polymorphism, robust error-handling using exception mechanisms to ensure program reliability.
2. To equip the learner to develop object oriented programs encompassing fundamental structures, environments, and the effective utilization of data types, arrays, strings, operators, and control statements for program flow in Java.
3. To enable the learner to design and develop event-driven graphical user interface (GUI) database applications using Swing and database connection components.

SYLLABUS

Module No.	Syllabus Description	Contact Hours
1	**Introduction to Java :-** Structure of a simple java program; Java programming Environment and Runtime Environment (Command Line & IDE); Java compiler; Java Virtual Machine; Primitive Data types and Wrapper Types; Casting and Autoboxing; Arrays; Strings; Vector class; Operators - Arithmetic, Bitwise, Relational, Boolean Logical, Assignment, Conditional (Ternary); Operator Precedence; Control Statements - Selection Statements, Iteration Statements and Jump Statements; Functions; Command Line Arguments; Variable Length Arguments; Classes; Abstract Classes; Interfaces. [***Use proper naming conventions***]	**10**

	OOP Concepts :- Data abstraction, encapsulation, inheritance, polymorphism, Procedural and object oriented programming paradigm; Microservices. **Object Oriented Programming in Java :-** Declaring Objects; Object Reference; Introduction to Methods; Constructors; Access Modifiers; ***this*** keyword.	
2	**Polymorphism :-** Method Overloading, Using Objects as Parameters, Returning Objects, Recursion. Static Members, Final Variables, Inner Classes. **Inheritance -** Super Class, Sub Class, Types of Inheritance, The ***super*** keyword, protected Members, Calling Order of Constructors. Method Overriding, Dynamic Method Dispatch, Using ***final*** with Inheritance.	**8**
3	**Packages and Interfaces –** Packages - Defining a Package, CLASSPATH, Access Protection, Importing Packages. **Interfaces -** Interfaces v/s Abstract classes, defining an interface, implementing interfaces, accessing implementations through interface references, extending interface(s). **Exception Handling -** Checked Exceptions, Unchecked Exceptions, ***try*** Block and ***catch*** Clause, Multiple catch Clauses, Nested ***try*** Statements, ***throw***, ***throws*** and ***finally***, Java Built-in Exceptions, Custom Exceptions. **Introduction to design patterns in Java :** Singleton and Adaptor.	**9**
4	**SOLID Principles in Java (https://www.javatpoint.com/solid-principles-java)** **Swings fundamentals** – Overview of AWT, Swing v/s AWT, Swing Key Features, Model View Controller (MVC), Swing Controls, Components and Containers, Swing Packages, Event Handling in Swings, Swing Layout Managers, Exploring Swings–JFrame, JLabel, The Swing Buttons, JTextField. **Event handling** – Event Handling Mechanisms, Delegation Event Model, Event Classes, Sources of Events, Event Listener Interfaces, Using the Delegation Event Model. **Developing Database Applications using JDBC** – JDBC overview, Types, Steps, Common JDBC Components, Connection Establishment, SQL	**10**

	Fundamentals [***For projects only***] - Creating and Executing basic SQL Queries, Working with Result Set, Performing CRUD Operations with JDBC.	

Suggestion on Project Topics

Student should Identify a topic to be implemented as project having the following nature

i. *It must accept a considerable amount of information from the user for processing.*
ii. *It must have a considerable amount of data to be stored permanently within the computer - as plain files / using databases..*
iii. *It must process the user provided data and the stored data to generate some output to be displayed to the user.*

SEMESTER S3

OBJECT ORIENTED PROGRAMMING

(Common to CS/CA/CD/AM/CB/CN/CU/CG)

Course Code	**PBCST304**	**CIE Marks**	60
Teaching Hours/Week (L:T:P:R)	3:0:0:1	**ESE Marks**	40
Credits	4	**Exam Hours**	2 Hrs. 30 Min.
Prerequisites (if any)	None	**Course Type**	Theory

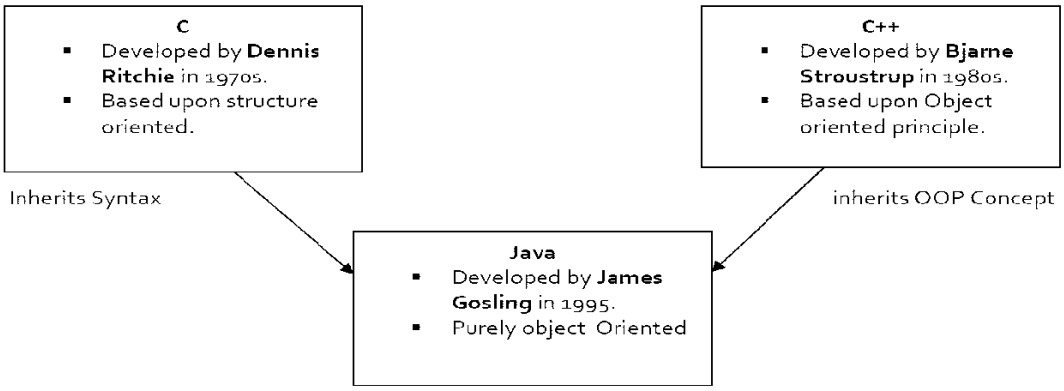
MODULE I

Introduction to Java:

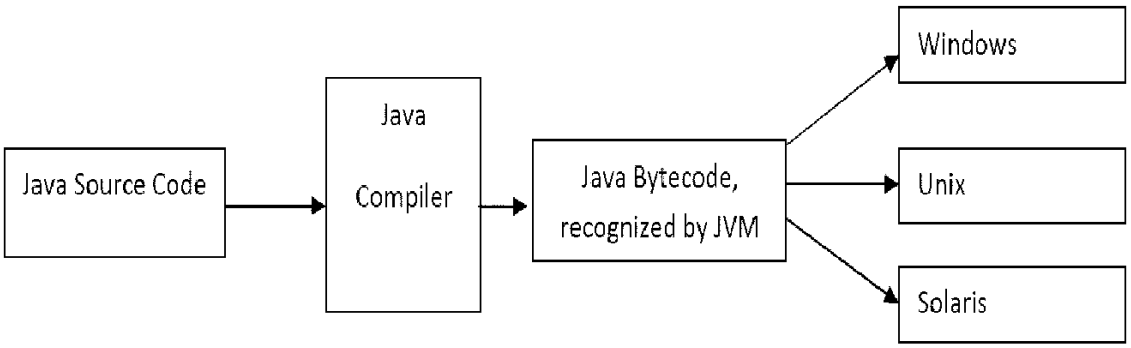
Structure of a simple java program; Java programming Environment and Runtime Environment Command Line & IDE); Java compiler; Java Virtual Machine; Primitive Data types and Wrapper Types; Casting and Autoboxing; Arrays; Strings; Vector class; Operators - Arithmetic, Bitwise, Relational, Boolean Logical, Assignment, Conditional (Ternary); Operator Precedence; Control Statements - Selection Statements, Iteration Statements and Jump Statements; Functions; Command Line Arguments; Variable Length Arguments; Classes; Abstract Classes; Interfaces. [Use proper naming conventions] OOP Concepts :-Data abstraction, encapsulation, inheritance, polymorphism, Procedural and object oriented programming paradigm; Microservices. Object Oriented Programming in Java :-Declaring Objects; Object Reference; Introduction to Methods; Constructors;Access Modifiers; this keyword.

Introduction to Java

- Java was developed in 1995 at Sun Microsystems.
- Java was developed by James Gosling.
- Initially it was called Oak, in honor of the tree outside James Gosling's window; its name was changed to Java because there was already a language called Oak.



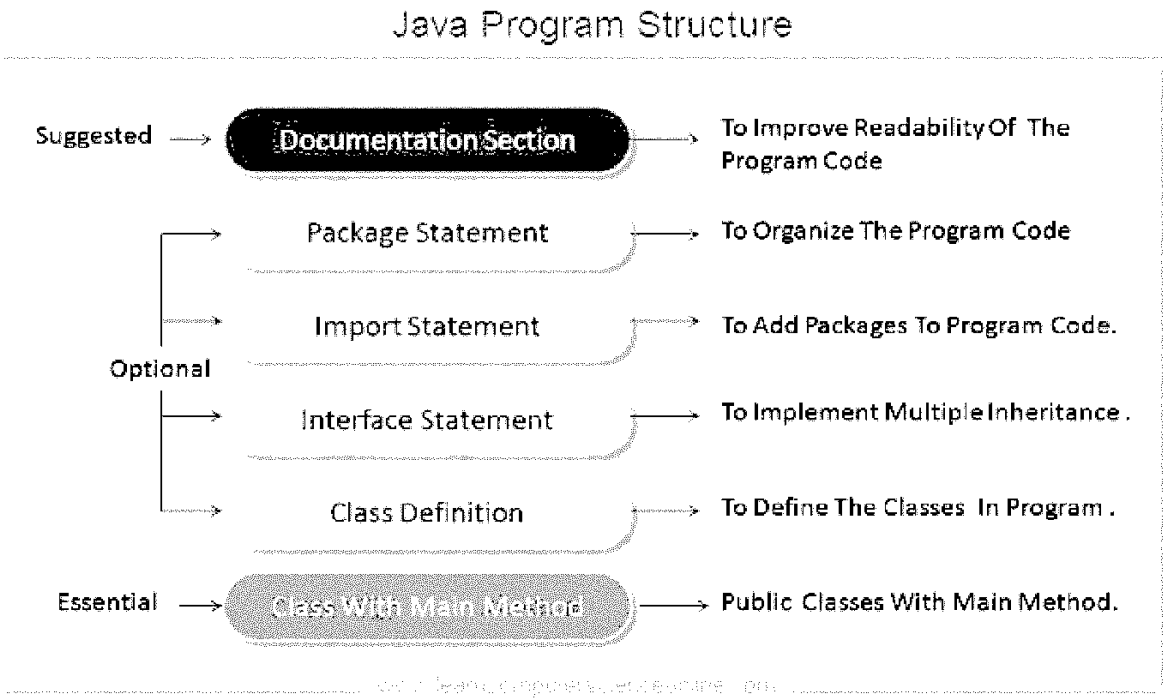
- Java is based upon the concept "Write once, run anywhere".
- The idea is that the same software should run on different kinds of computers, consumer gadgets, and other devices.
- JVM generates byte codes as a result of compilation.
- Java byte codes are a form of instructions understood by Java Virtual Machine and usually generated as a result of compiling Java language source code.



Structure of a simple java program

Java is a high-level, **object-oriented programming language** used to build web apps, mobile applications, and enterprise software systems. It is known for its **Write Once, Run Anywhere** capability, which means code written in Java can run on any device that supports the Java Virtual Machine (JVM).

Java syntax and structure is similar to C-based languages like C++ and C#. Its robustness, platform-independent compatibility, and strong memory management have made it a go-to language for developers worldwide.



Documentation Section

- It is used to improve the readability of the program.
- It comprises a comment line which gives the names program, the programmer's name and some other brief details.
- There are three types of comments that Java supports

Single line Comment

These begin with // and continue to the end of the line. They are typically used for short explanations or to temporarily disable a line of code.

```
// This is a single-line comment
int x = 10; // Initialize x to 10
```

Multi-line Comment

These start with /* and end with */, and can span multiple lines. They are used for longer explanations, to comment out blocks of code

```
/*
 * This is a multi-line comment.
 * It can span multiple lines.
 */
int y = 20;
/* Another way to use multi-line comments */
```

Documentation Comment

These begin with /** and end with */. They are used to generate API documentation using the Javadoc tool. They are placed before class, method, and field declarations to describe their purpose, parameters, return values, and exceptions.

```
/**
 * This class represents a person.
 */
public class Person {
    private String name;

    /**
     * Constructs a new Person object with the given name.
     * @param name the name of the person
     */
    public Person(String name) {
        this.name = name;
    }
```

Packages

Packages in Java serve as containers for organizing classes, interfaces, and sub-packages, creating namespaces to avoid naming conflicts and manage access control. They facilitate code

reusability and maintainability, especially in large projects. There are two types of packages in Java:

Built-in Packages:

These come with the Java Development Kit (JDK) and provide core functionalities, for example, java.lang, java.util, java.io, and java.net.

User-defined Packages:

Developers create these to organize their own classes and interfaces. Package names are typically written in lowercase, following a reverse domain name convention (e.g., com.example.project).

To create a package, the package keyword is used at the beginning of a Java source file:

```
package com.example.project;

public class MyClass {
    // Class implementation
}
```

```
import com.example.project.MyClass;

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
    }
}
```

To use classes from another package, the import keyword is used:

Import statement in Java is helpful to take a class or all classes visible for a program specified under a package, with the help of a single statement. It is pretty beneficial as the programmer do not require to write the entire class definition. Hence, it improves the readability of the program.

Syntax 1:

import package1[.package2].(*);

Here,

package1: Top-level package

package2: Subordinate-level package under package1

*: To import all the classes

Syntax 2:

import package1[.package2].(myClass);

Here,

package1: Top-level package

package2: Subordinate-level package under the top-level package

myClass: Import only myClass

Note: Either we can import a class or we can import all classes specified under the package.

To understand why we need to bring a class or classes into visibility, let us consider a Java program without the use of an import statement:

```
import packageName.ClassName;
import packageName.*;
```

Interface statement

- Interfaces are like a class that includes a group of method declarations.
- This is an optional section and can be used only when programmers want to implement multiple inheritances within a program.
- An interface is a lot similar to a class in Java but it contains only constants and method declarations.

Class definition

- A class is a collection of variables and methods that operate on the fields.
- Every program in Java will have at least one class with the main method.
- A Java program may contain multiple class definitions.
- Classes are the main and important elements of any Java program.
- These classes are used to plot the objects of the real world problem.

Syntax

```
class ClassName {
    // Fields
    dataType fieldName;
    // Constructor
    ClassName(parameters) {
        // Initialize fields
    }
    // Methods
    returnType methodName(parameters) {
        // Method body
    }
}
```

Example

```
class Car {
    // Fields
```

```
    String color;
    int year;
    // Constructor
    Car(String color, int year) {
        this.color = color;
        this.year = year;
    }
    // Method
    void displayInfo() {
        System.out.println("Car color: " + color + ", Year: " + year);
    }
}

public class Main {
    public static void main(String[] args) {
// Creating an object
        Car myCar = new Car("Red", 2020);
        myCar.displayInfo(); // Output: Car color: Red, Year: 2020
    }
}
```

Java programming Environment and Runtime Environment Command Line & IDE); Java compiler; Java Virtual Machine;

- Java is a recently developed, concurrent, class-based, object-oriented programming and runtime environment, consisting of:

  A programming language

  An API specification

  A virtual machine specification

Programming language

- It is a human-readable language which is generally considered **easy to read and write.**
- It is *class-based* and *object-oriented* in nature.
- Java is intended to be easy to learn and to teach.

- *The Java Language Specification (JLS)* defines how a confirming Java application must behave.

Application Programming Interface(API)

- API is a collection of prewritten packages, classes, and interfaces with their respective methods, fields and constructors.
- It is similar to a user interface, which facilitates interaction between humans and computers, an API serves as a software program interface facilitating interaction.
- In Java, most basic programming tasks are performed by the API's classes and packages, which are helpful in minimizing the number of lines written within pieces of code.

Virtual Machine Specification

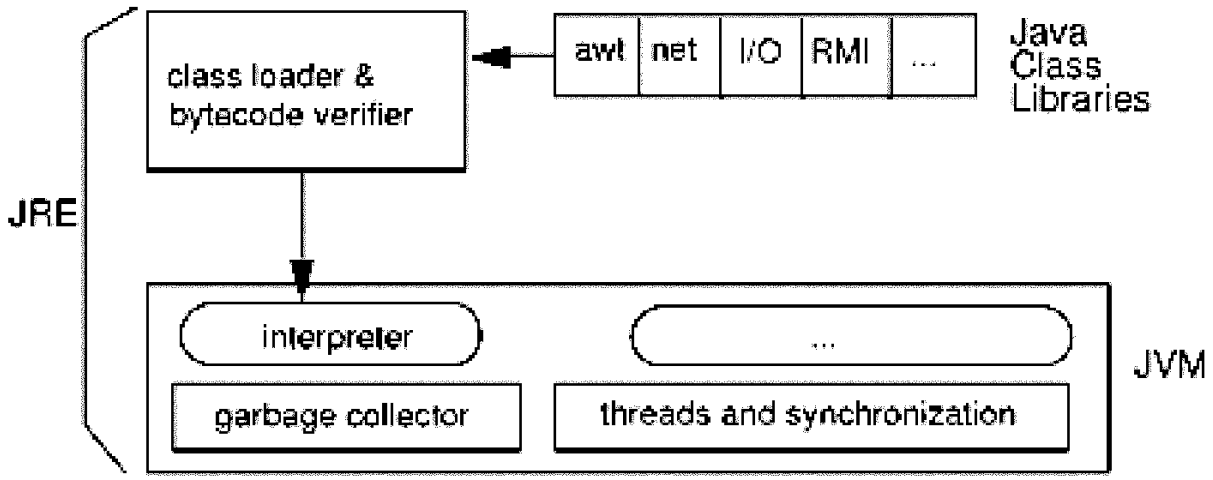
- The Java Virtual Machine is a program which provides the ***runtime environment*** to execute Java programs.
- Java programs cannot run if a supporting JVM is not available.
- JVM does not take Java source files as input. The java source code is first converted to ***bytecode*** by javac program.
- Javac takes in source files as input and outputs bytecode in the form of **class files** with .class extension.
- These class files are then interpreted (stepped through one at a time) by the ***JVM interpreter*** and the program is executed.

JVM makes the programmer's life easier by-

- providing a container for the Java program to run in
- creating a secure execution environment as compared to C/C++
- taking memory management out of the hands of the developer
- allowing class files from one platform to run on a different environment without any modification or recompilation

JRE Components

- The JRE is the software environment in which programs compiled for a typical JVM implementation can run.
- The runtime system includes:
    - Code necessary to run Java programs, dynamically link native methods, manage memory, and handle exceptions
    - Implementation of the JVM



Java Platform Components:

Java Development Kit (JDK)

- The Java Development Kit (JDK) is a software development environment used for developing and executing Java applications and applets
- It includes the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc) and other tools needed in Java development.
- JDK is only used by Java Developers.
- **JDK (Java Development Kit):** The complete set of tools for Java development. Explain it as the "toolbox for programmers."
- javac (Java Compiler)
- java (Java Launcher/Interpreter)
- javadoc (Documentation Generator)
- jar (Archiver)
- Other utility tools (e.g., jdb - debugger)

Development Platforms

There are four platforms of the Java programming language:

- Java Platform, Standard Edition (Java SE)
- Java Platform, Enterprise Edition (Java EE)

- Java Platform, Micro Edition (Java ME)
- Java FX

Each Java platform provides a virtual machine and an API

This allows applications written for that platform to run on any compatible system with all the advantages of the Java programming language: platform-independence, power, stability, ease-of-development, and security.

Java SE

- Java SE's API provides the core functionality of the Java programming language.
- computing platform in which we can execute software, and it can be used for development and deployment of portable code for desktop and server environments.

Java EE

- The Java EE platform is built on top of the Java SE platform.
- The Java EE platform provides an API and runtime environment for developing and running large-scale, multi-tiered, scalable, reliable, and secure network applications.

Java ME

- The Java ME platform provides an API and a small-footprint virtual machine for running Java programming language applications on small devices, like mobile phones.

Java FX

- Java FX technology is a platform for creating rich internet applications written in JavaFX Script.
- The applications built in JavaFX, can run on multiple platforms including Web, Mobile and Desktops.

JRE (Java Runtime Environment): Explain it as the "environment to *run* Java applications."

    - JVM
    - Java Class Libraries (APIs)

- **JVM (Java Virtual Machine):** The core component that executes Java bytecode. Emphasize its role in "Write Once, Run Anywhere."
    - Bytecode
    - Class Loader
    - Execution Engine
    - Runtime Data Areas
- **Setting up the Java Environment:**
    - **Installation:** Briefly discuss where to download JDK (Oracle/OpenJDK).
    - **Environment Variables:**

        - JAVA_HOME: Points to the JDK installation directory. Explain its importance for other tools and IDEs.
        - PATH: Adding the bin directory of JDK to the system's PATH. Explain why this is necessary to run javac and java from any directory.
    - **Verification:** java -version, javac -version
- **Command Line Compilation and Execution:**
    - **Steps:**
        1. Create a .java source file (e.g., HelloWorld.java) using a text editor.
        2. Compile using javac HelloWorld.java (explaining the .class file generation).
        3. Execute using java HelloWorld (explaining how JVM finds and runs the main method).
    - **Advantages:**
        - Fundamental understanding of Java compilation/execution.
        - Lightweight, no extra software.
        - Useful for scripting and server environments.
    - **Disadvantages:**
        - No code completion, syntax highlighting, debugging tools.
        - Error messages can be cryptic.
        - Not practical for large projects.

- **Integrated Development Environments (IDEs):**
    - **Definition:** What is an IDE? (Integrated tools for development).
    - **Why use an IDE?**
        - **Productivity Boosters:** Code completion (IntelliSense/Code completion), syntax highlighting, error checking on the fly.
        - **Debugging Tools:** Step-through execution, breakpoints, variable inspection.
        - **Project Management:** Organizing files, build automation.
        - **Refactoring:** Automated code restructuring.
        - **Version Control Integration:** (Brief mention)
    - **Popular Java IDEs:**
        - **IntelliJ IDEA (Community Edition):** Highly recommended for its smart features and user-friendliness.
        - **Eclipse:** Historically very popular, open-source.
        - NetBeans (brief mention).
    - **Basic IDE Usage (Demonstration Focus):**
        - Creating a new Java project.
        - Creating a new Java class.
        - Writing a simple main method.
        - Running the program directly from the IDE.
        - Briefly showing the project structure, output console.

Java Virtual Machine(JVM)

- JVM stands for Java Virtual Machine.
- JVM is the engine that drives the java code.
- The JVM is an abstract computing machine, having an instruction set that uses memory.
- The JVM is the cornerstone of the Java programming language.
- It is responsible for Java's cross-platform portability and the small size of its compiled code.
- Mostly in other Programming Languages, compiler produce code for a particular system but Java compiler produce Bytecode for a Java Virtual Machine.
- Bytecode is an intermediary language between Java source and the host system.
- It is the medium which compiles Java code to bytecode which gets interpreted on a different machine and hence it makes it Platform/Operating system independent.
- The Java compiler, javac, outputs bytecodes and puts them into a .class file.
- The JVM then interprets these bytecodes, which can then be executed by any JVM implementation, thus providing Java's cross-platform portability.
- As different computers with the different operating system have their own JVM, when we submit a .class file to any operating system, JVM interprets the bytecode into machine level language.
- JVM is the main component of Java architecture and it is the part of the JRE (Java Runtime Environment).
- A program of JVM is written into " C Programming Language" and JVM is Operating System dependent.
- JVM is responsible for allocating the necessary memory needed by the Java program.
- JVM is responsible for deallocating memory space.

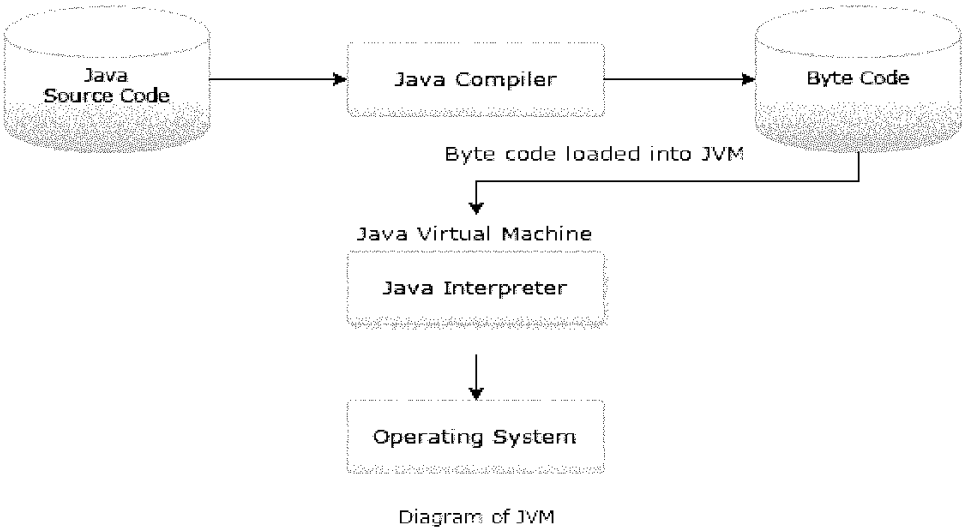
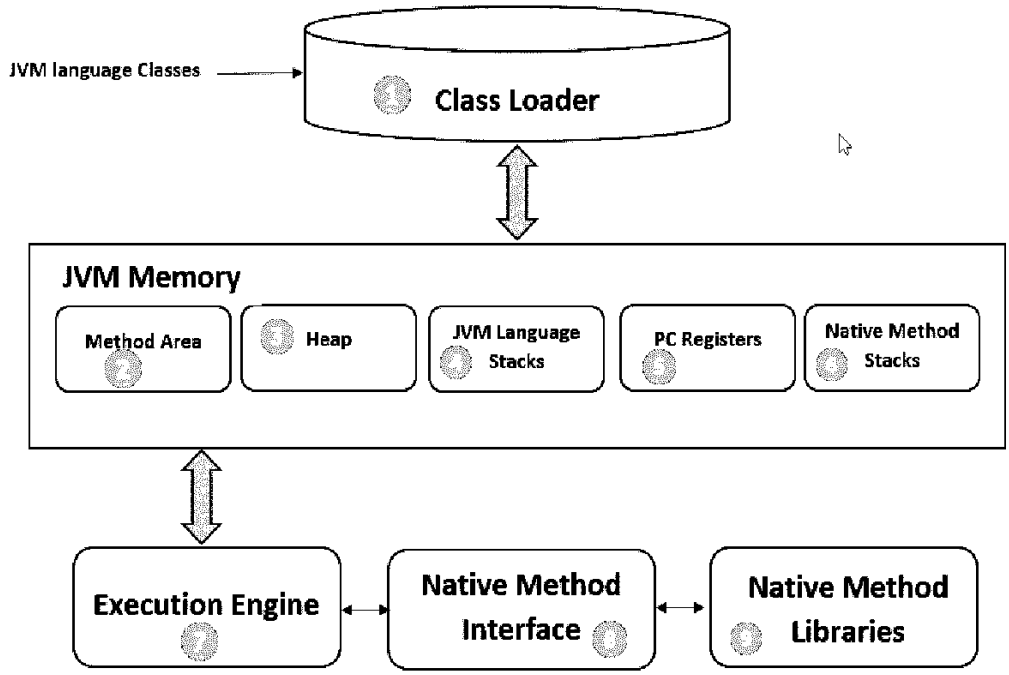


Diagram of JVM



Class Loader

- The class loader is a subsystem used for loading class files. It is mainly responsible for three activities.
    - Loading
    - Linking
    - Initialization

Loading:

- The Class loader reads the .class file, generates the corresponding binary data and saves it in the method area. For each .class file, JVM stores the following information in the method area.
    - Fully qualified name of the loaded class and its immediate parent class.
    - Whether .class file is related to Class or Interface
        - Modifier, Variables and Method information etc.
- After loading the .class file, JVM creates an object of type Class to represent this file in the heap memory.
- This object is of type Class predefined in java.lang package.
- This Class object can be used by the programmer for getting class level information like name of class, parent name, methods and variable information etc.
- To get this object reference we can use the getClass() method of Object class.

Linking:

- Performs verification, preparation

- Verification: It ensures the correctness of a .class file i.e. it checks whether this file is properly formatted and generated by a valid compiler or not. If verification fails, we get a run-time exception java.lang.VerifyError.
- Preparation: JVM allocates memory for class variables and initializes the memory to default values.

Initialization:

- In this phase, all static variables are assigned with their values defined in the code and static block (if any).

Method area:

- In the method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables.
- There is only one method area per JVM, and it is a shared resource.

Heap area:

- Information of all objects is stored in the heap area.
- There is also one Heap Area per JVM.
- It is also a shared resource.

Stack area:

- For every thread, JVM creates one run-time stack which is stored here.
- Every block of this stack is called activation record/stack frame which store methods calls.
- All local variables of that method are stored in their corresponding frame.
- After a thread terminates, its run-time stack will be destroyed by JVM.
- It is not a shared resource.

Execution Engine

- Execution engine executes the .class (bytecode).
- It reads the byte-code line by line, uses data and information present in various memory areas and executes instructions.
- It can be classified in three parts
    - Interpreter: It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.
- **Just-In-Time Compiler (JIT):** It is used to increase efficiency of interpreters.
- It compiles the entire bytecode and changes it to native code so whenever the interpreter sees repeated method calls, JIT provides direct native code for that part so re-interpretation is not required, thus efficiency is improved.

- Garbage Collector: It destroys un-referenced objects. For more on Garbage Collector

Java Native Interface (JNI)

- It is an interface which interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution.
- It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.

Native Method Libraries:

- It is a collection of the Native Libraries(C, C++) which are required by the Execution Engine.
- **PC Registers:**
  - Store address of current execution instruction of a thread. Obviously each thread has separate PC Registers.

Native method stacks:

- For every thread, a separate native stack is created. It stores native method information.

JAVA COMPILER

- being a platform independent programming language, doesn't work on one-step-compilation.
- it involves a two-step execution,
  - OS independent compiler;
  - a virtual machine (JVM) which is custom-built for every operating system.
- The source '.java' file is passed through the compiler, which encodes the source code into a machine independent encoding, known as Bytecode.
- The content of each class contained in the source file is stored in a separate '.class' file.
- While converting the source code into the bytecode, the compiler follows the following steps:
  - **Parse**: Reads a set of *.java source files and maps the resulting token sequence into AST (Abstract Syntax Tree)-Nodes.
  - **Enter**: Enter symbols for the definitions into the symbol table.
  - **Process annotations**: If Requested, processes annotations found in the specified compilation units.

- **Attribute**: Attributes the Syntax trees. This step includes name resolution, type checking and constant folding.
- **Flow**: Performs dataflow analysis on the trees from the previous step. This includes checks for assignments and reachability.
- **Desugar**: Rewrites the AST and translates away some syntactic sugar.
- **Generate**: Generates '.Class' files.

Execution

- The class files generated by the compiler are independent of the machine or the OS, which allows them to be run on any system.
- To run, the main class file (the class that contains the method main) is passed to the JVM, and then goes through three main stages before the final machine code is executed.

These stages are:

Class Loader

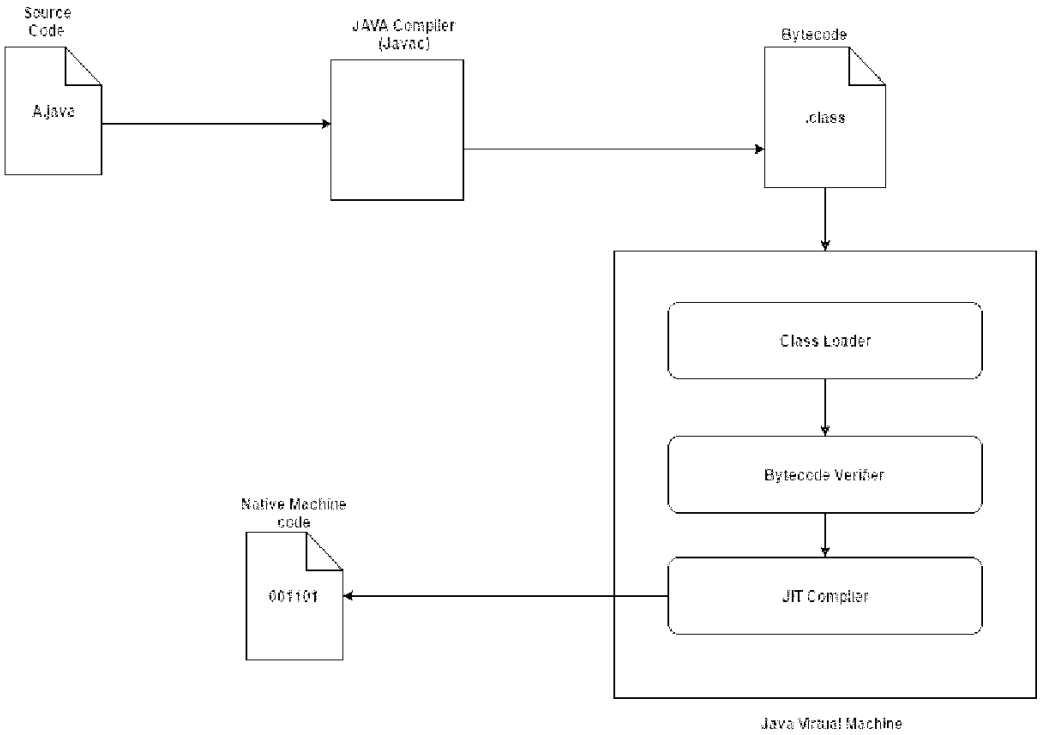
- The main class is loaded into the memory by passing its '.class' file to the JVM, through invoking the latter.
- All the other classes referenced in the program are loaded through the class loader.
- A class loader, itself an object, creates a flat name space of class bodies that are referenced by a string name.

Bytecode Verifier

- After the bytecode of a class is loaded by the class loader, it has to be inspected by the bytecode verifier,
- It checks that the instructions don't perform damaging actions.
- The following are some of the checks carried out:
  - Variables are initialized before they are used.
  - Method calls match the types of object references.
  - Rules for accessing private data and methods are not violated.
  - Local variable accesses fall within the runtime stack.
  - The run time stack does not overflow.
  - If any of the above checks fails, the verifier doesn't allow the class to be loaded.

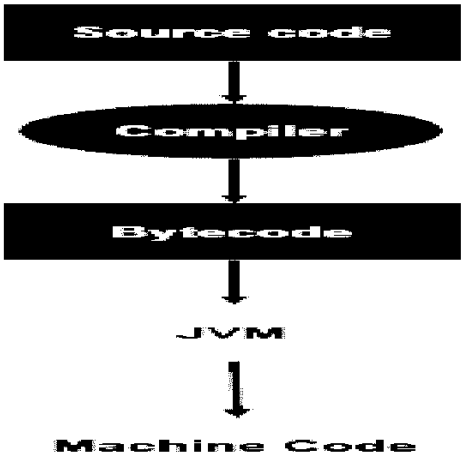
Just-In-Time Compiler

- This is the final stage encountered by the java program, and its job is to convert the loaded bytecode into machine code.
- The JIT Compiler is activated and enabled by default when a method is invoked in Java.
- When a method is compiled, Java Virtual Machine invokes the compiled code of the method directly without interpreting it.
- Hence, it does not require much memory usage and processor time.
- That basically speeds up the performance of the Java Native Application.



BYTE CODE

- Java program is compiled bytecode is generated
- Java bytecode is the machine code in the form of a .class file.
- A bytecode in Java is the instruction set for Java Virtual Machine and acts similar to an assembler.

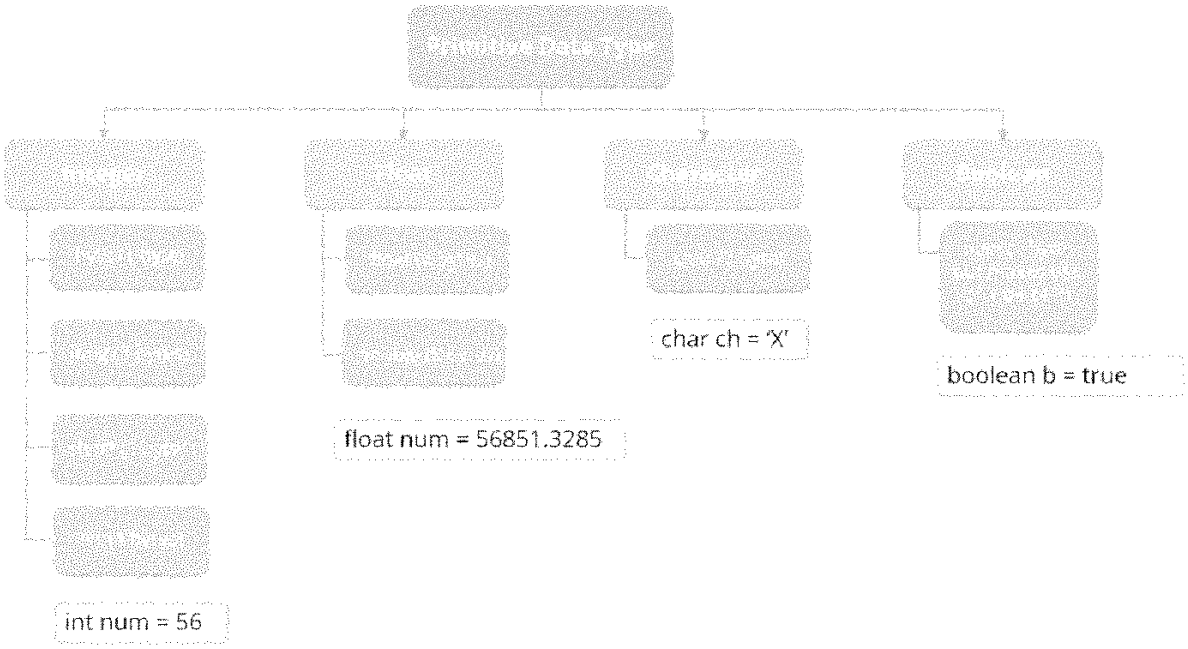


- When a Java program is executed, the compiler compiles that piece of code and a Bytecode is generated for each method in that program in the form of a .class file.
- We can run this bytecode on any other platform as well.
- The bytecode generated after the compilation is run by the Java virtual machine.
- Resources required for the execution are made available by the Java virtual machine for smooth execution which calls the processor to allocate the resources.
- Machine code is a set of instructions in machine language or binary which can be directly executed by the CPU.
- Bytecode is a non-runnable code generated by compiling a source code that relies on an interpreter to get executed.

Primitive Data types and Wrapper Types; Casting and Autoboxing; Arrays; Strings; Vector class

Primitive Data types

- A data type is used to represent different values which are stored in a variable.
- They are mainly classified into 4 different aspects
  - Integer
  - Float
  - Character
  - Boolean



Integers

- Java defines four integer types: **byte, short, int,** and **long**.
- All of these are signed, positive and negative values.
- The smallest integer type is **byte**.
- The most commonly used type is **int**.

Integers – byte

- The smallest integer type is byte.
- This is a signed 8-bit. Variables of type byte are especially useful
    - when you're working with a stream of data from a network or file.
    - when you're working with raw binary data that may not be directly compatible with Java's other built-in types.
- Byte variables are declared by use of the **byte** keyword.
- For example, the following declares two byte variables called b and c:
- **byte b, c;**

Integers – short

- short is a signed 16-bit type.
- It is probably the least used Java type.
- Example

- **short s;**

Integers – int

- The most commonly used integer type is **int**.
- In addition to other uses, variables of type int are commonly employed to control loops and to index arrays.
- Example:
  - **int a;**

Integers – long

- long is a signed 64-bit type and is useful for those occasions where an int type is not large enough to hold the desired value.
- Example:
- **long a;**

Floating point

- **Floating-point numbers**: This group includes **float** and **double**, which represent numbers with **fractional** precision.
- Float specifies a single precision value and double specifies a double precision value.

Floating point: float

- The type **float** specifies a single-precision value that uses 32 bits of storage.
- Variables of type **float** are useful when you need a fractional component, but don't require a large degree of precision.
- Example:
  - **float hightemp, lowtemp;**

Floating point: double

- Double precision, as denoted by the double keyword, uses 64 bits to store a value.
- When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, double is the best choice.
- Example :
  - **double pi, r, a;**

Characters

- The data type used to store characters is **char**.
- Java uses *Unicode* to represent characters

- Unicode requires 16 bits.
- char is a 16-bit type.
- Example:
  - **char letterA = 'A';**

BOOLEAN

- Java has a primitive type, called **boolean**, for logical values.
- It can have only one of two possible values, **true** or **false**.
- This is the type returned by all relational operators, as in the case of **a < b**.

Literals

- Any constant value which can be assigned to the variable is called literal/constant.
- their values cannot be changed once assigned.
- Literals can be assigned to any primitive type variable.

```
public class DataTypesExample {
    public static void main(String[] args) {
        // --- Primitive Data Types ---
        // 1. Integer Types
        byte myByte = 100; // 8-bit signed integer
        short myShort = 10000; // 16-bit signed integer
        int myInt = 100000; // 32-bit signed integer (most commonly used)
        long myLong = 10000000000L; // 64-bit signed integer (note the 'L' suffix for long literals)
        System.out.println("--- Primitive Integer Types ---");
        System.out.println("byte: " + myByte);
        System.out.println("short: " + myShort);
        System.out.println("int: " + myInt);
        System.out.println("long: " + myLong);
        System.out.println();
        // 2. Floating-Point Types
        float myFloat = 23.45f; // 32-bit floating-point (note the 'f' suffix for float literals)
```

```
        double myDouble = 123.456789; // 64-bit floating-point (default for decimal numbers)
        System.out.println("--- Primitive Floating-Point Types ---");
        System.out.println("float: " + myFloat);
        System.out.println("double: " + myDouble);
        System.out.println();
        // 3. Character Type
        char myChar = 'A'; // 16-bit Unicode character
        char unicodeChar = '\u0041'; // 'A' using Unicode representation
        System.out.println("--- Primitive Character Type ---");
        System.out.println("char: " + myChar);
        System.out.println("Unicode char: " + unicodeChar);
        System.out.println();
        // 4. Boolean Type
        boolean isJavaFun = true; // Represents true or false
        boolean isSunny = false;
        System.out.println("--- Primitive Boolean Type ---");
        System.out.println("Is Java fun? " + isJavaFun);
        System.out.println("Is it sunny? " + isSunny);
        System.out.println();
    }
}
```

output

```
--- Primitive Integer Types ---
byte: 100
short: 10000
int: 100000
long: 10000000000

--- Primitive Floating-Point Types ---
float: 23.45
double: 123.456789
```

--- Primitive Character Type ---

char: A

Unicode char: A

--- Primitive Boolean Type ---

Is Java fun? true

Is it sunny?false

In Java, **Wrapper Classes** are a set of classes that provide a way to use primitive data types (like int, char, boolean, etc.) as objects. Each primitive data type has a corresponding wrapper class.

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

Why Do We Need Wrapper Classes?

- Java is designed to be an object-oriented language. While primitive data types are efficient for storing simple values, there are many scenarios where you need to treat these values as objects. This is where wrapper classes come in.

Here are the primary reasons for using wrapper classes:

Java Collections Framework:

- The Java Collections Framework (e.g., ArrayList, HashMap, HashSet, Vector) can only store objects, not primitive data types. If you want to store a list of integers, you can't use ArrayList<int>. Instead, you use ArrayList<Integer>.
- Wrapper classes allow you to store primitive values within these collections by "wrapping" them in their corresponding object form.

Generics:

- Java's generics feature (like List<T>, Map<K, V>) works only with object types. You cannot use primitive types as type parameters for generics. For example, List<int> is invalid, but List<Integer> is valid.

Null Values:

- Primitive types cannot have a null value (e.g., an int always has a default value of 0, not null).
- Wrapper class objects, being objects, can be assigned null. This is crucial in situations where a value might be absent or unknown, such as when retrieving data from a database.

Serialization:

- If you need to serialize (convert an object into a byte stream for storage or transmission) primitive values, they must first be converted into objects. Wrapper classes facilitate this.

Synchronization in Multithreading:

- In multithreaded programming, synchronization mechanisms (like synchronized blocks) often operate on objects. If you need to synchronize access to a primitive value, you'd wrap it in its corresponding wrapper class.

Utility Methods:

- Wrapper classes provide useful utility methods for manipulating their primitive values. For example:

Integer.parseInt("123") converts a string to an int.

Double.toString(3.14) converts a double to a String.

Integer.MAX_VALUE, Integer.MIN_VALUE for constants.

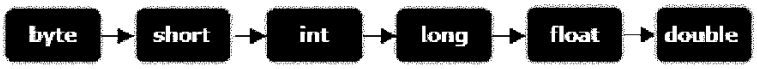
Casting and Autoboxing

- Converting one primitive data type into another is known as type casting (type conversion) in Java.
- If the two types are compatible, then Java will perform the conversions automatically.
- **You can cast the primitive data types in two ways namely,**
  - **Widening**
  - **Narrowing**

Widening

- Converting a lower datatype to a higher data type is known as widening conversion.
- This casting/conversion is done automatically therefore, it is known as implicit type casting.

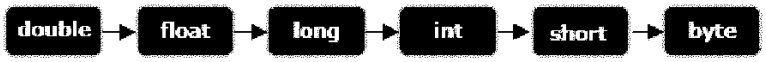
- In this case both data types should be compatible with each other.




```
public class WideningExample
{
public static void main(String
args[])
{
        char ch = 'C';
        int i=ch;
        System.out.println(i);
}
}
O/P: 67
```

Narrowing

- Converting a higher datatype to a lower datatype is known as narrowing.
- casting/conversion is not done automatically, you need to convert explicitly using the cast operator "( )"explicitly. Therefore, it is known as explicit type casting.
- In this case both datatypes need not be compatible with each other.



```
public class WideningExample
{
public static void main(String
        args[])
{
        int i=67;
         char ch = (char) i;
        System.out.println(ch);

}
}
O/P : c
```

Autoboxing :

- The automatic conversion of a primitive type to its corresponding wrapper class object.

int primitiveInt = 10;

Integer wrapperInt = primitiveInt; // Autoboxing: int to Integer

The Java compiler applies autoboxing in two primary scenarios:

1.Assignment: When a primitive value is assigned to a variable of its corresponding wrapper class.

```
int count = 50;
Integer countObject = count; // Autoboxing
```

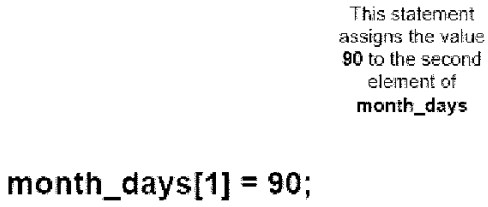
2.Method Invocation: When a primitive value is passed as an argument to a method that expects an object of the corresponding wrapper class.

```
public static void printIntegerObject(Integer num) {
   System.out.println("Received Integer object: " + num);
}
public static void main(String[] args) {
   int value = 123;
   printIntegerObject(value); // Autoboxing: int 'value' is converted to Integer
}
```

Arrays; Strings; Vector class

Arrays

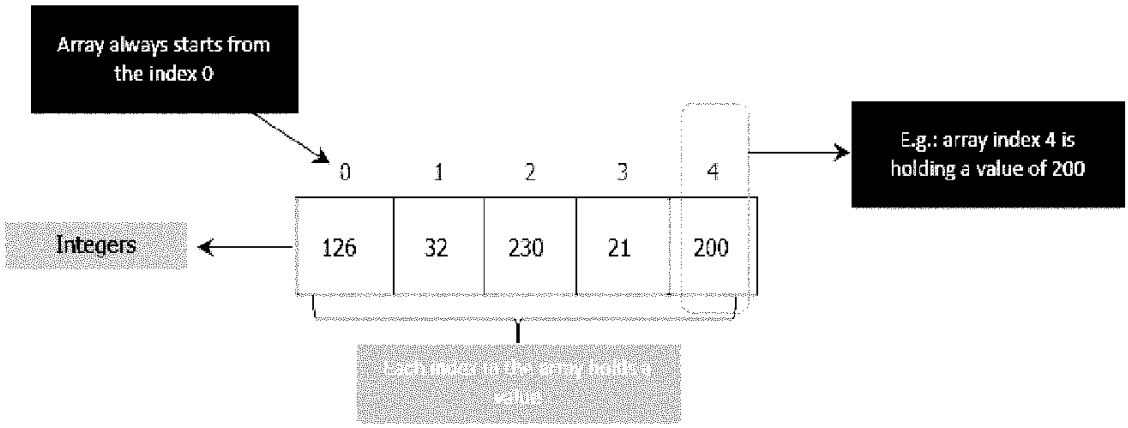
- An array is a group of same data typed variables that are referred to by a common name.
- An array is an ordered set of values grouped together under a single identifier.
- It is a data structure consisting of related data or a collection of data items.
- The members of an array are called **elements**.
- The values are stored in indexed locations within the array.
- The individual elements of the array are accessed by using the **name** of the array followed by the **index** value of the array element enclosed in square [] brackets.



- The index value is also known as **position**. It must be an integer or an integer expression.
- The **length** of an array is the number of elements that an array contains.
- The array element index starts with **0**. The last array element index number is one less than the length of the array.
- An array must be declared before it is used.
- Arrays of any type can be created and may be **One-dimensional** or **Multi-dimensional**.
- Arrays are dynamically allocated.

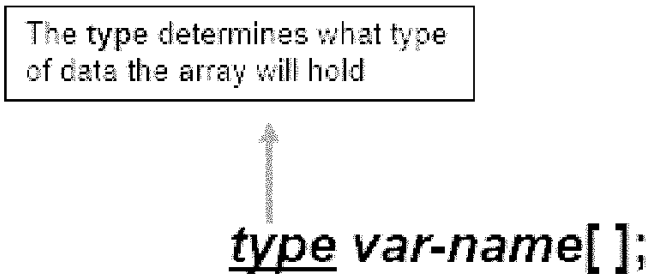
One-Dimensional Arrays

- A one-dimensional array is a list of same data typed variables. To create an array, we first create an array variable of the desired type. The type declares the data type of the elements in the array.
- **Syntax:** type arrayname[ ] ;
- **Eg:** int months[ ] ;



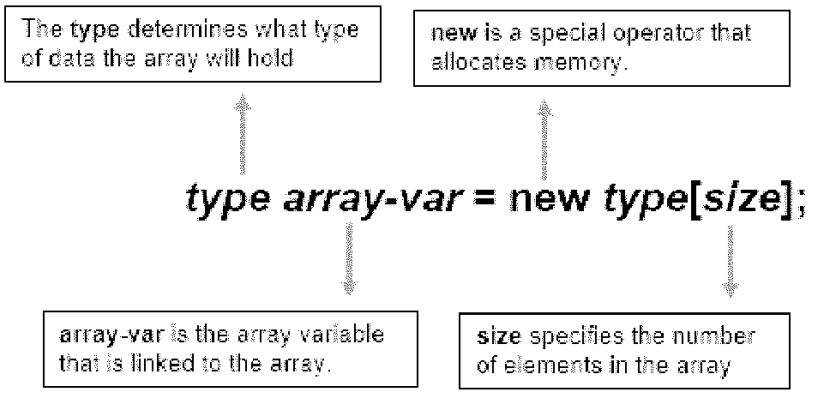
- Obtaining an array is a two-step process.
    - First, you must declare a variable of the desired array type
    - Second, you must allocate the memory that will hold the array, using new, and assign it to the array variable

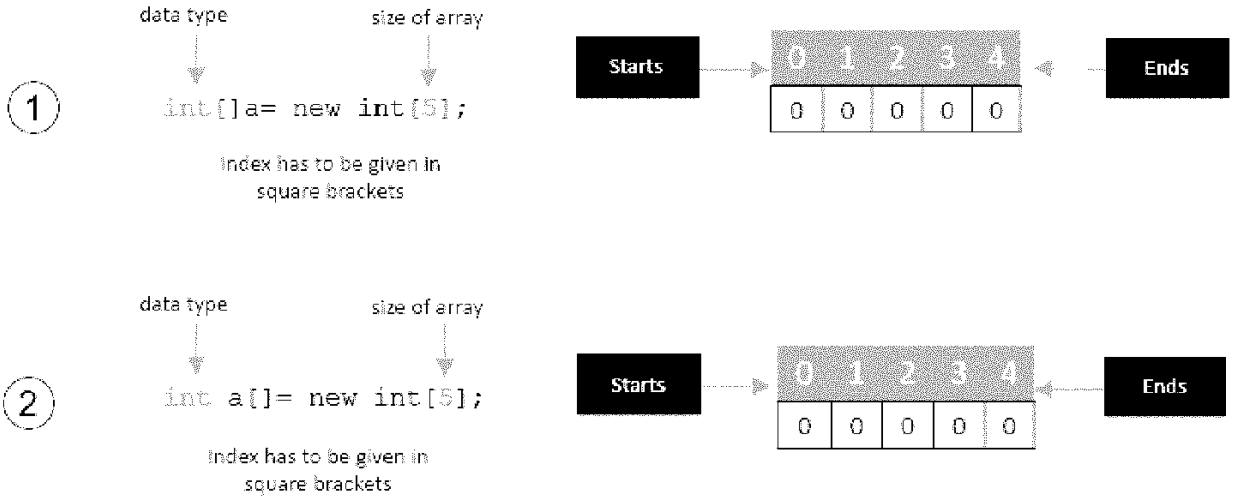
Array initialization-1



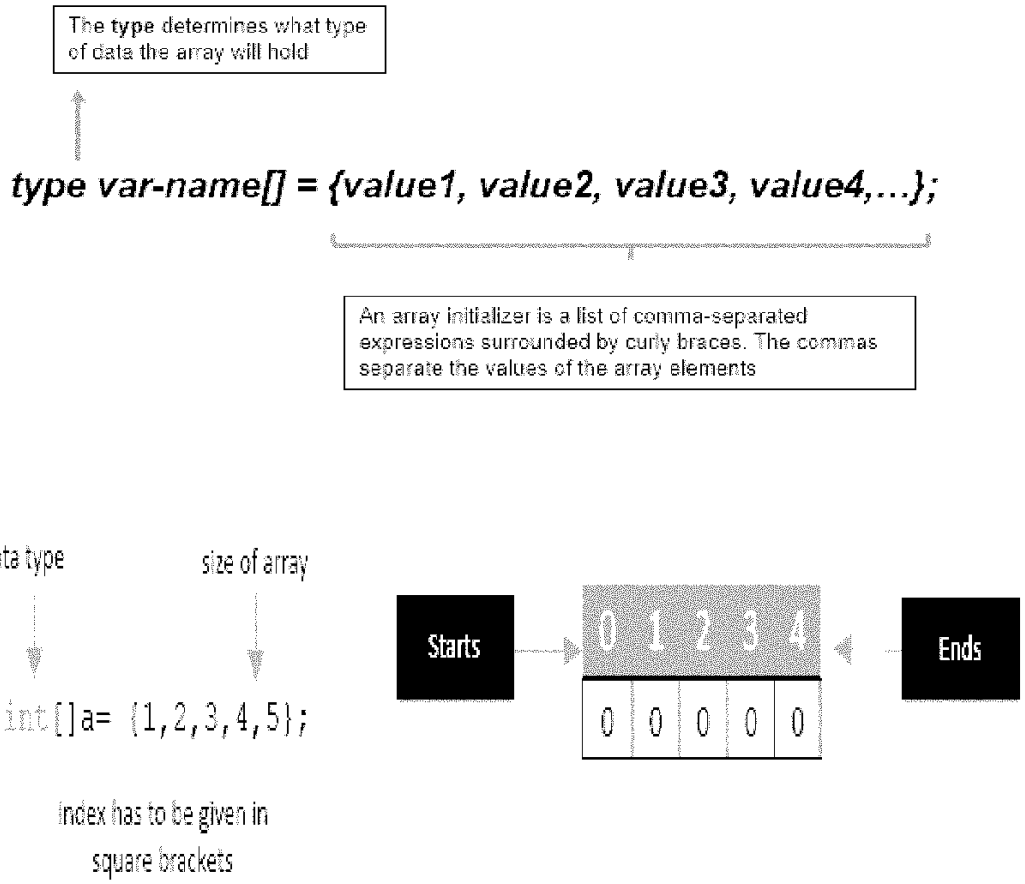
Example:- *int monthdays[];*

Array initialization-2





Array initialization-3

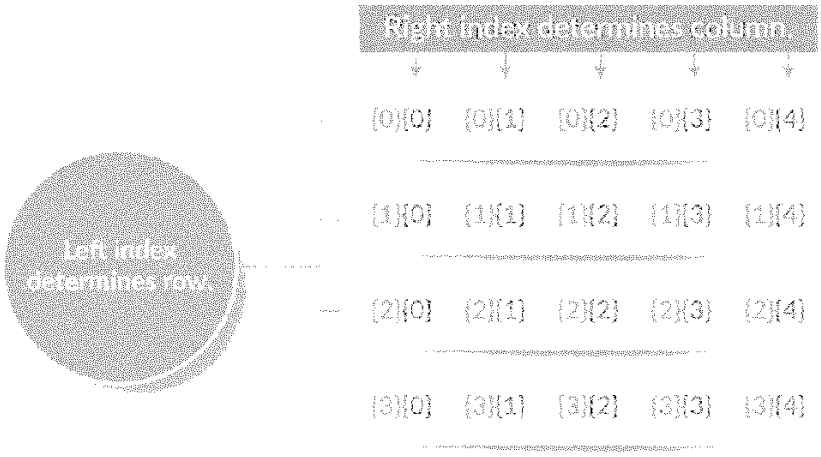


Multidimensional Array

- To declare Multidimensional Array, we have to specify each additional index using another set of square brackets.

This allocates a **4** by **5** array and assigns it to **Mul**.

int Mul[ ][ ] = new int[4][5];




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

	0	1
0	1	4
1	4	5

2 x 2 dimensional int array

```
char [][]a= new char[3][2];
```

	0	1
0	s	a
1	g	v
2	v	d

Strings

- String is a sequence of characters.
- The Java platform provides the **String** class to create and manipulate strings.
- The *java.lang.String* class is used to create string object.
- String greeting = "Hello world!";
    - In this case, "Hello world!" is a *string literal*—a series of characters enclosed in double quotes.
    - Whenever it encounters a string literal in your code, the compiler creates a String object with its value—in this case, Hello world!.
- String can be created as objects by using the new keyword and a constructor.
- The String class has thirteen constructors that allow you to provide the initial value of the string using different sources, such as an array of characters:
- **String helloString = new String(helloArray);**

- char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };

Vector class

- Vector is like the *dynamic array* which can grow or shrink its size.
- we can store n-number of elements in it as there is no size limit.

Vector can be created as follows

Vector<Type> vector = new Vector<>();

Eg:

// create Integer type linked list Vector<Integer> vector= new Vector<>();

// create String type linked list Vector<String> vector= new Vector<>();

- Add Elements to Vector
  - add(element) - adds an element to vectors
  - add(index, element) - adds an element to the specified position
  - addAll(vector) - adds all elements of a vector to another vector

```
import java.util.Vector;
class Main
{
    public static void main(String[] args)
    {
            Vector<String> mammals= new Vector<>();          mammals.add("Dog");

            mammals.add("Horse");
            mammals.add(2, "Cat");
            System.out.println("Vector: " + mammals);
            Vector<String> animals = new Vector<>();
            animals.add("Crocodile");
            animals.addAll(mammals);
            System.out.println("New Vector: " + animals);
    }
}
```

Output

Vector: [Dog, Horse, Cat] New Vector: [Crocodile, Dog, Horse, Cat]

- Access Vector Elements
  - get(index) - returns an element specified by the index
  - iterator() - returns an iterator object to sequentially access vector elements

```
import java.util.Iterator;
import java.util.Vector;
class Main
 {
        public static void main(String[] args)
        {
                Vector<String> animals= new Vector<>();  animals.add("Dog");
                animals.add("Horse");
                animals.add("Cat");
                String element = animals.get(2);
                System.out.println("Element at index 2: " + element);
        Iterator<String> iterate = animals.iterator();  System.out.print("Vector: ");
                while(iterate.hasNext())
                {
                System.out.print(iterate.next()); System.out.print(", ");
} } }
```

Output: Element at index 2: Cat Vector: Dog, Horse, Cat,

OPERATOR

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulation.

Java operators can be divided into following categories:

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- conditional operator (Ternary)

Operator	Description	Example
+ (Addition)	Adds two operands	5 + 10 =15
- (Subtraction)	Subtract second operands from first. Also used to Concatenate two strings	10 - 5 =5
* (Multiplication)	Multiplies values on either side of the operator.	10 * 5 =50
/ (Division)	Divides left-hand operand by right-hand operand.	10 / 5 =2
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.	5 % 2 =1
++ (Increment)	Increases the value of operand by 1.	2++ gives 3
-- (Decrement)	Decreases the value of operand by 1.	3-- gives 2

```
class ArithmeticOperations {

     public static void main (String[] args){

          int answer = 2 + 2;
          System.out.println(answer);

          answer = answer - 1;
          System.out.println(answer);

          answer = answer * 2;
          System.out.println(answer);

          answer = answer / 2;
          System.out.println(answer);

          answer = answer + 8;
          System.out.println(answer);


          answer = answer % 7;
          System.out.println(answer);
     }
}
```

Output

```
4
3
6
3
11
4
```

```
class IncrementDecrementExample {
        public static void main(String args[]){

        int x= 5;
        System.out.println(x++);
        System.out.println(++x);
        System.out.println(x--);
        System.out.println(--x);
        }
}
```

Output

```
5
7
7
5
Press any key to continue . . .
```

++x is Change – Then - Use

```
class IncrementDecrementExample{

        public static void main(String args[]){
        int p=10;
        int q=10;
        System.out.println(p++ + ++p);//10+12=22
        System.out.println(q++ + q++);//10+11=21

        }
}
```

Output

```
22
21
Press any key to continue . . .
```

X++ is Use –Then - Change

Use of Modulus Operator

```
class ModulusOperator {
  public static void main(String args[]) {
    int     R = 42;
    double S = 62.25;

    System.out.println("R mod 10 = " + R % 10);
    System.out.println("S mod 10 = " + S % 10);
  }
}
```

Output

```
R mod 10 = 2
S mod 10 = 2.25
Press any key to continue . . .
```

Joining or Concatenate two strings

```
class AssignmentConcatination {
    public static void main(String[] args){

        String firstName = "Rahim";
        String lastName  = "Ramboo";

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

Output

```
RahimRamboo
Press any key to continue . . .
```

Boolean Operators

Operators	Descriptions	Examples
== (equal to)	This operator checks the value of two operands, if both are **equal**, then it returns true otherwise false.	(2 == 3) is not true.
!= (not equal to)	This operator checks the value of two operands, if both are **not equal**, then it returns true otherwise false.	(4 != 5) is true.
> (greater than)	This operator checks the value of two operands, if the left side of the operator is **greater**, then it returns true otherwise false.	(5 > 56) is not true.
< (less than)	This operator checks the value of two operands if the left side of the operator is **less**, then it returns true otherwise false.	(2 < 5) is true.
>= (greater than or equal to)	This operator checks the value of two operands if the left side of the operator is **greater or equal**, then it returns true otherwise false.	(12 >= 45) is not true.
<= (less than or equal to)	This operator checks the value of two operands if the left side of the operator is **less or equal**, then it returns true otherwise false.	(43 <= 43) is true.

```
public class RelationalOperator {

    public static void main(String args[]) {
        int p = 5;
        int q = 10;

        System.out.println("p == q = " + (p == q) );
        System.out.println("p != q = " + (p != q) );
        System.out.println("p > q = " + (p > q) );
        System.out.println("p < q = " + (p < q) );
        System.out.println("q >= p = " + (q >= p) );
        System.out.println("q <= p = " + (q <= p) );
    }
}
```

Output

```
p == q = false
p != q = true
p > q = false
p < q = true
q >= p = true
q <= p = false
```

Bitwise Operators

Operator	Description
& (bitwise and)	Bitwise AND operator give true result if both operands are true. otherwise, it gives a false result.
| (bitwise or)	Bitwise OR operator give true result if any of the operands is true.
^ (bitwise XOR)	Bitwise Exclusive-OR Operator returns a true result if both the operands are different. otherwise, it returns a false result.
~ (bitwise compliment)	Bitwise One's Complement Operator is unary Operator and it gives the result as an opposite bit.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.
>>> (zero fill right shift)	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.

```
class BitwiseAndOperator {
      public static void main(String[] args){

            int A = 10;
            int B = 3;
            int Y;
            Y = A & B;
            System.out.println(Y);

      }
}
```

Output

```
2
```

```
class BitwiseAndOperator {
    public static void main(String[] args){

        int A = 10;
        int B = 3;
        int Y;
        Y = A & B;
        System.out.println(Y);

    }
}
```

```
class BitwiseOrOperator {
    public static void main(String[] args){

        int A = 10;
        int B = 3;
        int Y;
        Y = A | B;
        System.out.println(Y);

    }
}
```

Output

```
2
```

Output

```
11
```

Logical Operators

Operator	Description	Example
&& (logical and)	If both the operands are non-zero, then the condition becomes true.	(0 && 1) is false
|| (logical or)	If any of the two operands are non-zero, then the condition becomes true.	(0 || 1) is true
! (logical not)	Logical NOT Operator Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(0 && 1) is true

```
public class LogicalOperatorDemo {
    public static void main(String args[]) {
        boolean b1 = true;
        boolean b2 = false;

        System.out.println("b1 && b2: " + (b1&&b2));
        System.out.println("b1 || b2: " + (b1||b2));
        System.out.println("!(b1 && b2): " + !(b1&&b2));
    }
}
```

Output:

```
b1 && b2: false
b1 || b2: true
!(b1 && b2): true
```

Assignment Operators

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x – 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3

conditional Operator / Ternary Operator (? :)

Expression1 ? Expression2 : Expression3

Expression ? value if true : value if false

```
public class ConditionalOperator {

    public static void main(String args[]) {
        int a, b;
        a = 20;
        b = (a == 1) ? 10: 25;
        System.out.println( "Value of b is : " +  b );
        b = (a == 20) ? 20: 30;
        System.out.println( "Value of b is : " + b );
    }
}
```

Output

```
Value of b is : 25
Value of b is : 20
```

```
public class TernaryOperatorDemo {

   public static void main(String args[]) {
        int num1, num2;
        num1 = 25;
        /* num1 is not equal to 10 that's why
         * the second value after colon is assigned
         * to the variable num2
         */
        num2 = (num1 == 10) ? 100: 200;
        System.out.println( "num2: "+num2);

        /* num1 is equal to 25 that's why
         * the first value is assigned
         * to the variable num2
         */
        num2 = (num1 == 25) ? 100: 200;
        System.out.println( "num2: "+num2);
   }
}
```

Output:

```
num2: 200
num2: 100
```

Operator Precedence

• Evaluate 2*x-3*y ?

(2x)-(3y) or 2(x-3y) which one is correct??????

• Evaluate A / B * C

A / (B * C) or (A / B) * C Which one is correct?????

To answer these questions satisfactorily one has to understand the priority or precedence of operations.

Priority	Operators	Description
1st	* / %	multiplication, division, modular division
2nd	+ -	addition, subtraction
3rd	=	assignment

Precedence order - When two operators share an operand the operator with the higher precedence goes first.

• Associativity - When an expression has two operators with the same precedence, the expression is evaluated according to its associativity

Precedence	Operator	Type	Associativity
15	() [] .	Parentheses Array subscript Member selection	Left to Right
14	++ --	Unary post-increment Unary post-decrement	Right to left
13	++ -- + - ! ~ (*type*)	Unary pre-increment Unary pre-decrement Unary plus Unary minus Unary logical negation Unary bitwise complement Unary type cast	Right to left
12	* / %	Multiplication Division Modulus	Left to right
11	+ -	Addition Subtraction	Left to right
10	<< >> >>>	Bitwise left shift Bitwise right shift with sign extension Bitwise right shift with zero extension	Left to right
9	< <= > >= instanceof	Relational less than Relational less than or equal Relational greater than Relational greater than or equal Type comparison (objects only)	Left to right
8	== !=	Relational is equal to Relational is not equal to	Left to right
7	&	Bitwise AND	Left to right
6	^	Bitwise exclusive OR	Left to right
5	|	Bitwise inclusive OR	Left to right
4	&&	Logical AND	Left to right
3	||	Logical OR	Left to right
2	? :	Ternary conditional	Right to left
1	= += -= *= /= %=	Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment	Right to left

Evaluate i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8

```
i = 6 / 4 + 4 / 4 + 8 - 2 + 5 / 8          operation: *
i = 1 + 4 / 4 + 8 - 2 + 5 / 8              operation: /
i = 1 + 1+ 8 - 2 + 5 / 8                   operation: /
i = 1 + 1+ 8 - 2 + 0                       operation: /
i = 2 + 8- 2 + 0                           operation: +
i = 10 - 2 + 0                             operation: +
i = 8 + 0                                  operation : -
i = 8                                      operation: +
```

Control Statements - Selection Statements, Iteration Statements and Jump Statements

- Selection statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable.
- Also called decision making statements
- Java supports various selection statements, like if, if-else and switch
- There are various types of if statements in java.
    - if statement
    - if-else statement
    - nested if statement
    - if-else-if ladder

- Use the if statement to specify a block of Java code to be executed if a condition is true.

Syntax

if (condition)

{

// block of code to be executed if the condition is true

}

```
class SampleIf
{
    public static void main(String args[])
    {
        int a=10;
        if (a > 0) {
            System.out.println("a is greater than 0");
        }
    }
}
```

Output:

```
a is greater than 0
```

if-else Statement

- If-else statement also tests the condition.
- It executes the if block if the condition is true otherwise the else block is executed.

Syntax

if (condition)

{

// block of code to be executed if the condition is true

}

else

{

// block of code to be executed if the condition is false

}

```
class SampleIfElse
{
    public static void main(String args[])
    {
        int a=10;
        if (a > 0) {
            System.out.println("a is greater than 0");
        }
        else
        {
            System.out.println("a is smaller than 0");
        }
    }
}
```

Output:

```
a is greater than 0
```

Nested if else Statement

```
if (condition) {
   if (condition)
   {
       // block of code to be executed if the condition is true
   }
   else
   {
      // block of code to be executed if the condition is false
   }
}
else
{
    if (condition) {
       // block of code to be executed if the condition is true
    }
    else
    {
      // block of code to be executed if the condition is false
    }
}
```

```
class SampleNestedIfElse
{
   public static void main(String args[])
   {
      int a=10,b=20,c=30;
      if (a>b)
      {
         if (a>c)
         {
             System.out.println("a is greatest.");
         }
         else
         {
              System.out.println("c is greatest.");
         }
      }
      else
      {
         if (b>c)
         {
             System.out.println("b is greatest.");
         }
         else
         {
             System.out.println("c is greatest.");
         }
      }
   }
}
```

Output:

c is greatest.

if else if ladder

Syntax

```
if (condition)
{
// block of code to be executed if the condition is true
}
else if (condition)
 {
// block of code to be executed if the condition is true
 }
else
{
// block of code to be executed if the condition is true
 }
```

```
class IfElseIfLadder {
     public static void main(String[] args){
        double score = 55;

        if (score >= 90.0)
        System.out.println('A');
        else if (score >= 80.0)
        System.out.println('B');
        else if (score >= 70.0)
        System.out.println('C');
        else if (score >= 60.0)
        System.out.println('D');
        else
      System.out.println('F');
    }
  }
```

Output

```
F
Press any key to continue . . .
```

```
class SampleLadderIfElse
{
   public static void main(String args[])
   {
      int a=10;
      if (a > 0) {
         System.out.println("a is +ve");
      }
      else if (a < 0) {
         System.out.println("a is -ve");
      }
      else {
         System.out.println("a is zero");
      }
   }
}
```

Output:

a is +ve

```
class SampleLadderIfElse
{
   public static void main(String args[])
   {
      int a=10;
      if (a > 0) {
         System.out.println("a is +ve");
      }
      else if (a < 0) {
         System.out.println("a is -ve");
      }
      else {
         System.out.println("a is zero");
      }
   }
}
```

Output:

a is +ve

If...Else & Ternary Operator – A comparison

```
int time = 20;
if (time < 18) {
   System.out.println("Good day.");
} else {
   System.out.println("Good evening.");
}
```

```
int time = 20;
String result = (time < 18) ? "Good day." : "Good evening.";
System.out.println(result);
```

switch case

- The if statement in java, makes selections based on a single true or false condition.

- But switch case have multiple choice for selection of the statements
- It is like if-else-if ladder statement
- How to Java switch works:
  - Matching each expression with case
  - Once it matches, execute all cases from where it matched.
  - Use break to exit from switch
  - Use default when expression does not match with any case.

Syntax

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

```
class SampleSwitch
{
   public static void main(String args[])
   {
        int day = 4;
        switch (day) {
          case 1:
                System.out.println("The day is Monday");
                break;
          case 2:
                System.out.println("The day is Tuesday");
                break;
          case 3:
                System.out.println("The day is Wednesday");
                break;
          case 4:
                System.out.println("The day is Thursday");
                break;
          case 5:
                System.out.println("The day is Friday");
                break;
          case 6:
                System.out.println("The day is Saturday");
                break;
          case 7:
                System.out.println("The day is Sunday");
                break;
          deafault:
                System.out.println("Please enter between 1 to 7.");
        }
    }
}
```

Output The day is Thursday

Why break is necessary in switch statement

- The break statement is used inside the switch to terminate a statement sequence.

- When a break statement is encountered, execution branches to the first line of code that follows the entire switch statement
- This has the effect of jumping out of the switch.
- The break statement is optional. If you omit the break, execution will continue on into the next case.

```
class NestedSwitchCase {
public static void main(String args[]) {
int count = 1;
int target = 1;
 switch(count) {
          case 1:
                  switch(target) { // nested switch
                          case 0:
                           System.out.println("target is zero inner switch");
                           break;
                          case 1: // no conflicts with outer switch
                           System.out.println("target is one inner switch");
                           break;
                          }
          break;

case 2:
     System.out.println("case 2 outer switch");
         }
 }
}
```

```
target is one inner switch
```

Iteration Statements (Loop)

- A loop can be used to tell a program to execute statements repeatedly
- A loop repeatedly executes the same set of instructions until a termination condition is met.

While Loop

- The while loop first checks the condition if the condition is true then control goes inside the loop body otherwise goes outside of the body.

Syntax

while (condition)

{

// code block to be executed

}

```
class WhileLoopExample
{
     public static void main(String args[])
     {
         int count = 0;
            while(count < 100){
            System.out.println("Welcome to atnyla!");
            count++;
         }
     }
}
```

```
Welcome to atnyla!
Welcome to atnyla!
Welcome to atnyla!
.....
.....
.....
Welcome to atnyla!
Welcome to atnyla!
Welcome to atnyla!
```

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

```
1
2
3
4
5
6
7
8
9
10
```

```
class WhileLoopSingleStatement {
    public static void main(String[] args){
        int count = 1;
        while (count <= 11)
         System.out.println("Number Count : " + count++);
    }
}
```

```
Number Count : 1
Number Count : 2
Number Count : 3
Number Count : 4
Number Count : 5
Number Count : 6
Number Count : 7
Number Count : 8
Number Count : 9
Number Count : 10
Number Count : 11
```

```
public class WhileInfiniteLoop {
public static void main(String[] args) {
     while(true){
          System.out.println("infinitive while loop");
     }
   }
}
```

```
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
.....
```

(Boolean Condition inside while loop)

```
class WhileLoopBoolean {
    public static void main(String[] args){
        boolean a = true;
        int count = 0 ;
        while (a)
            {
            System.out.println("Number Count : " + count);
            count++;
            if(count==5)
               a = false;
            }
    }
}
```

```
Number Count : 0
Number Count : 1
Number Count : 2
Number Count : 3
Number Count : 4
```

---

do...while loop

- A do while loop is a control flow statement that executes a block of code at least once, and then repeatedly executes the block, or not, depending on a given condition at the end of the block (in while).

Syntax

do {

// code block to be executed

} while (condition);

```
class DoWhile {
public static void main(String args[]) {
       int n = 0;
       do {
              System.out.println("Number " + n);
              n++;
          } while(n < 10);
   }
}
```

```
public class InfiniteDoWhileLoop {
public static void main(String[] args) {
    do{
       System.out.println("infinitive do while loop");
     }while(true);
  }
}
```

```
infinitive do while loop
infinitive do while loop
infinitive do while loop
infinitive do while loop
infinitive do while loop
infinitive do while loop
infinitive do while loop
infinitive do while loop
infinitive do while loop
.....
.....
.....
```

Difference Between while and do-while Loop

BASIS FOR COMPARISON	WHILE	DO-WHILE

General Form	`while ( condition) {`
	`statements; //body of loop`
	`}`
		`do{`
		`.`
		`statements; // body of loop.`
		`.`
		`} while( Condition );`

Controlling Condition	In 'while' loop the controlling condition appears at the start of the loop.	In 'do-while' loop the controlling condition appears at the end of the loop.
Iterations	The iterations do not occur if, the condition at the first iteration, appears false.	The iteration occurs at least once even if the condition is false at the first iteration.

for loop

- For Loop is used to execute a set of statements repeatedly until the condition is true.

Syntax

for (initialization; condition; increment/decrement)

{

// code block to be executed

}

Initialization : It executes at once.

Condition : This check will get true.

Increment/Decrement: This is for increment or decrement.

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

}
```

```
1
2
3
4
5
6
7
8
9
10
```

```
/*
Demonstrate the for loop.
Call this file "ForLoopExample.java".
*/

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

        for(int x = 15; x < 25; x = x + 1) {
            System.out.print("value of x : " + x );
            System.out.print("\n");
        }
        }

}
```

```
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
value of x : 20
value of x : 21
value of x : 22
value of x : 23
value of x : 24
```

For-each or Enhanced For Loop

- The for-each loop is used to traverse arrays or collections in java.
- It is easier to use than simple for loop because we don't need to increment value and use subscript notation.

Syntax

for (type variableName : arrayName)

{

// code block to be executed

}

```
/*
Demonstrate the for each loop.
save file "ForEachExample.java".
*/

public class ForEachExample {
public static void main(String[] args) {
    int array[]={10,11,12,13,14};
    for(int i:array){
        System.out.println(i);
    }
  }
}
```

```
10
11
12
13
14
```

Labeled For Loop

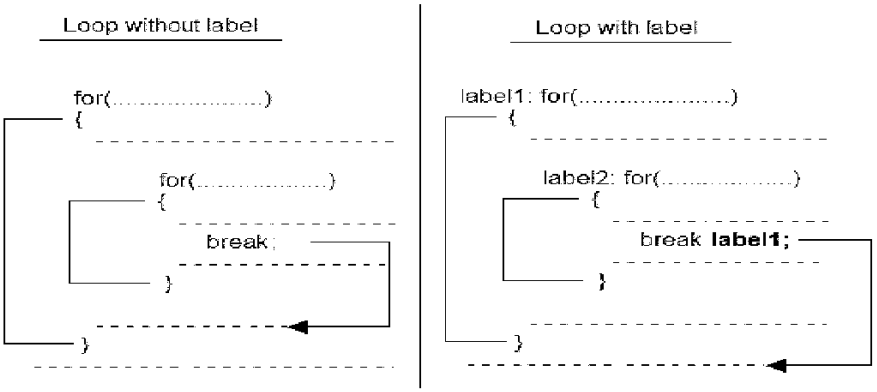
- According to the nested loop, if we put a break statement in the inner loop, the compiler will jump out from the inner loop and continue the outer loop again.
- What if we need to jump out from the outer loop using the break statement given inside the inner loop? The answer is, we should define labels along with the colon(:) sign before the loop.

Syntax

labelname:

for(initialization; condition; increment/decrement)

{ //code to be executed }




```
class WithoutLabelledLoop
{
      public static void main(String args[])
      {
            int i,j;
            for(i=1;i<=10;i++)
            {
                  System.out.println();
                  for(j=1;j<=10;j++)
                  {
                        System.out.print(j + " ");
                        if(j==5)
                        break;                  //Statement 1
                  }
            }
      }
}
```

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

```
class WithLabelledLoop
{
     public static void main(String args[])
     {
          int i,j;
          loop1:    for(i=1;i<=10;i++)
          {
               System.out.println();
               loop2:    for(j=1;j<=10;j++)
               {
                    System.out.print(j + " ");
                    if(j==5)
                         break loop1;       //Statement 1
               }
          }
     }
}
```

```
1 2 3 4 5
```

Java Break Statement

- The Java break statement 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 immediately terminated and the program control resumes at the next statement following the loop.
- In the case of an inner loop, it breaks only the inner loop.

```
class SampleBreak
{
   public static void main(String args[])
   {
         int num= 1;
         while (num <= 10) {
           System.out.println(num);
           if(num==5)
           {
               break;
           }
           num++;
         }
   }
}
```

Output

```
1
2
3
4
5
```

```
//Java Program to demonstrate the use of break statement
//inside the for loop.
public class BreakExample {
public static void main(String[] args) {
   //using for loop
   for(int i=1;i<=10;i++){
      if(i==5){
         //breaking the loop
         break;
      }
      System.out.println(i);
   }
}
}
```

Output:

```
1
2
3
4
```

```
//Java Program to illustrate the use of break statement
//inside an inner loop
public class BreakExample2 {
public static void main(String[] args) {
      //outer loop
      for(int i=1;i<=3;i++){
          //inner loop
          for(int j=1;j<=3;j++){
            if(i==2&&j==2){
               //using break statement inside the inner loop
               break;
            }
            System.out.println(i+" "+j);
          }
      }
}
}
```

Output:

```
1 1
1 2
1 3
2 1
3 1
3 2
3 3
```

```
//Java Program to demonstrate the use of break statement
//inside the Java do-while loop.
public class BreakDoWhileExample {
public static void main(String[] args) {
   //declaring variable
   int i=1;
   //do-while loop
   do{
      if(i==5){
        //using break statement
        i++;
        break;//it will break the loop
      }
      System.out.println(i);
      i++;
   }while(i<=10);
}
}
```

Output:

```
1
2
3
4
```

Java Continue Statement

- The Java continue statement is used to continue the loop
- The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately
- It continues the current flow of the program and skips the remaining code at the specified condition.
- In the case of an inner loop, it continues the inner loop only.

```
//Java Program to demonstrate the use of continue statement
//inside the for loop.
public class ContinueExample {
public static void main(String[] args) {
   //for loop
   for(int i=1;i<=10;i++){
      if(i==5){
        //using continue statement
        continue;//it will skip the rest statement
      }
      System.out.println(i);
   }
}
}
```

Output:

```
1
2
3
4
6
7
8
9
10
```

```
//Java Program to illustrate the use of continue statement
//inside an inner loop
public class ContinueExample2 {
public static void main(String[] args) {
        //outer loop
        for(int i=1;i<=3;i++){
            //inner loop
            for(int j=1;j<=3;j++){
              if(i==2&&j==2){
                //using continue statement inside inner loop
                continue;
              }
              System.out.println(i+" "+j);
            }
        }
}
}
```

Output:

```
1 1
1 2
1 3
2 1
2 3
3 1
3 2
3 3
```

```
//Java Program to demonstrate the use of continue statement
//inside the while loop.
public class ContinueWhileExample {
public static void main(String[] args) {
   //while loop
   int i=1;
   while(i<=10){
      if(i==5){
        //using continue statement
        i++;
        continue;//it will skip the rest statement
      }
      System.out.println(i);
      i++;
   }
}
}
```

Output:

```
1
2
3
4
6
7
8
9
10
```

Functions; Command Line Arguments; Variable Length Arguments;

Functions

In Java, **functions** are referred to as **methods**. A method is a block of code that performs a specific task, can be called (invoked) when needed, and optionally returns a value.

```
returnType methodName(parameters) {
   // method body
   // optional return statement
}
public class Example {
   // Method without return value
   void greet() {
      System.out.println("Hello, welcome!");
```

```
    }

    // Method with return value
    int add(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        Example obj = new Example();
        obj.greet();  // calling greet method
        int result = obj.add(5, 3);  // calling add method
        System.out.println("Sum: " + result);
    }
}
```

Types of Methods in Java

Type	Description
Instance Method	Belongs to an object; needs an object to call.
Static Method	Belongs to the class, can be called without an object.
Abstract Method	Declared without a body in an abstract class/interface.
Constructor	Special method used to initialize objects. (Not a typical method, but similar.)

Command Line Arguments

- Sometimes we want to pass information into a program when we run it. This is accomplished by passing command-line arguments to main( ).
- The main method can receive string arguments from the command line
- To access the command-line arguments inside a Java program is quite easy— they are stored as strings in a String array passed to the args parameter of main( ).

The first command-line argument is stored at args[0], the second at args[1], and so on.

```
// Display all command-line arguments.
class CmdLine {
public static void main(String args[]) {
        for(int i=0; i<args.length; i++)
        System.out.println("args[" + i + "]: " +args[i]);
        }
}
```

```
C:\Users\Hello World\Desktop\JAVA>javac CmdLine.java

C:\Users\Hello World\Desktop\JAVA>java CmdLine This is Commend-Line Argument Example
args[0]: This
args[1]: is
args[2]: Commend-Line
args[3]: Argument
args[4]: Example
```

Variable Length Arguments

- Varargs is a short name for variable arguments.
- In Java, an argument of a method can accept an arbitrary number of values. This argument that can accept variable number of values is called varargs.
- The syntax for implementing varargs is as follows:

accessModifier methodName(data type… arg){

// method body

}

- (three dots) is used in the formal parameter of a method.
- A method that takes a variable number of arguments is called a variable-arity method, or simply a varargs method.

```
class VarargExample
{
    public int sumNumber(int ... args)
    {
    System.out.println("argument
    length: " + args.length);

        int sum = 0;
        for(int x: args)
        {
            sum += x;
        }
        return sum;
    }

public static void main( String[] args )
{
    VarargExample ex = new
    VarargExample();
    int sum2 = ex.sumNumber(2, 4);
    System.out.println("sum2 = " +
    sum2);
    int sum3 = ex.sumNumber(1, 3, 5);
    System.out.println("sum3 = " +
    sum3);
    int sum4 = ex.sumNumber(1, 3, 5, 7);
    System.out.println("sum4 = " +
    sum4);
}}
```

Output

```
argument length: 2
sum2 = 6
argument length: 3
sum3 = 9
argument length: 4
sum4 = 16
```

Classes; Abstract Classes; Interfaces

Class

A **class** is a blueprint for creating objects. It can contain:

- Fields (variables)
- Methods (functions)
- Constructors
- Nested classes

```
public class Animal {
    String name;
    void speak() {
        System.out.println(name + " makes a sound.");
    }
}
```

Creating and Using Objects:

```
public class Main {
    public static void main(String[] args) {
        Animal dog = new Animal();
        dog.name = "Dog";
        dog.speak();  // Output: Dog makes a sound.
    }
}
```

Abstract Class

An **abstract class** cannot be instantiated. It can have:

- Abstract methods (without a body)
- Concrete methods (with a body)
- Fields and constructors

```
abstract class Animal {
    String name;
    abstract void makeSound();  // abstract method
    void sleep() {
        System.out.println("Sleeping...");
    }
```

```
}
class Dog extends Animal {
    void makeSound() {
        System.out.println("Bark");
    }
}
public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.makeSound();  // Bark
        d.sleep();      // Sleeping...
    }
}
```

Interface

An **interface** is a contract that classes can implement. It can contain:

- Abstract methods (implicitly public and abstract)
- Default methods (Java 8+)
- Static methods
- Constants (public static final)

```
interface Animal {
    void eat();  // implicitly public and abstract
}

class Cat implements Animal {
    public void eat() {
        System.out.println("Cat eats fish.");
    }
}
public class Main {
    public static void main(String[] args) {
        Cat c = new Cat();
        c.eat();  // Cat eats fish.
```

```
    }
}
```

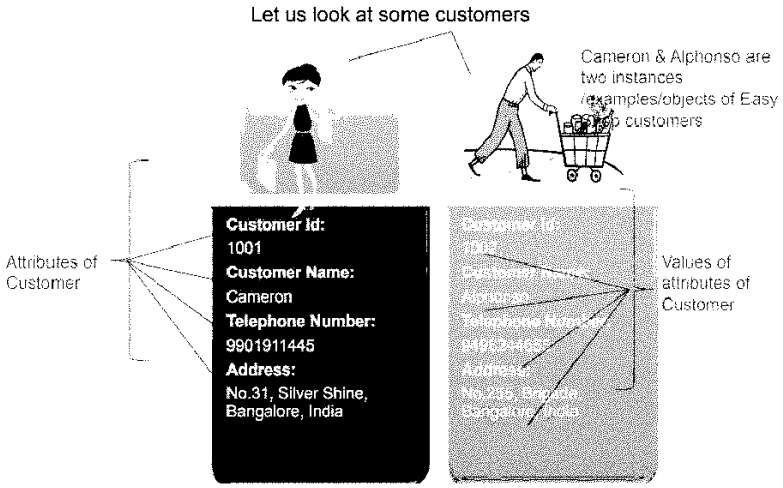
Feature	Class	Abstract Class	Interface
Can be instantiated	Yes	No	No
Can have constructors	Yes	Yes	No
Method types allowed	Any	Abstract + Concrete	Abstract + Default
Fields allowed	Any	Any	public static final only
Inheritance	Single only	Single only	Multiple allowed
Keyword	class	abstract class	interface
Implements	No	No	Yes (implements)
Extends	Yes	Yes (extends)	Yes (extends)

OOP Concepts :-
Data abstraction, encapsulation, inheritance, polymorphism, Procedural and object oriented programming paradigm; Microservices.

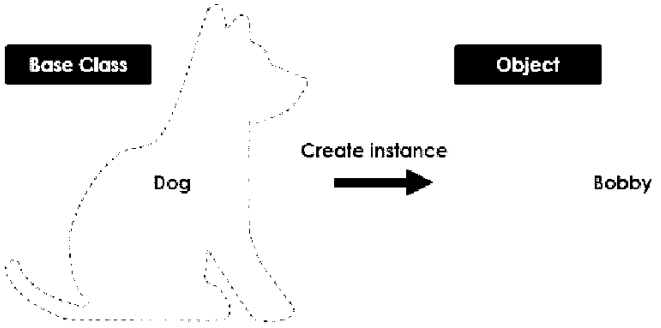
Basic object oriented concepts

Objects

- Object is a real world entity that has attributes(states) and behavior.
- Software objects are often used to model the real-world objects that you find in everyday life.



- Objects are basic run-time entities in an object oriented system
- Objects are instances of a class; these are user defined data types.
- Object take up space in memory and have an associated address
- When a program is executed the objects interact by sending messages to one another.
- Each object contains data and code to manipulate the data.
- Objects can interact without having to know details of each others data or code
- It is sufficient to know the type of message accepted and type of response returned by the objects.



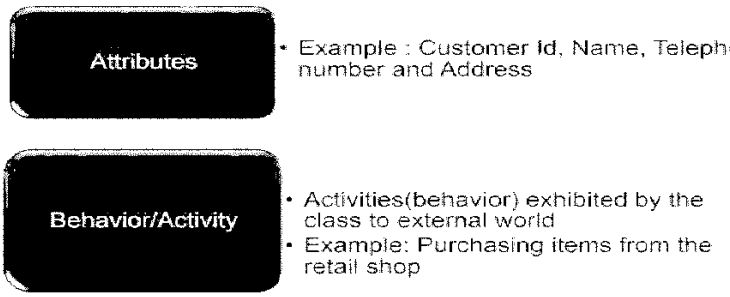

```
public class Puppy
{
  public static void main(String []args)
  {
  Puppy myPuppy = new Puppy();
  }
```

}

Classes

- Class is a blueprint of data and functions or methods.
- Class does not take any space.
- A Class is a user defined data-type which has data members and member functions
- Data members are the data variables and member functions are the functions used to manipulate these variables
- Together these data members and member functions define the properties and behavior of the objects in a class.
- **Objects are variable of type class**

- A class is a software design that describes the common attributes and activities (behavior) of objects




```
public class Dog
{
    String breed;   // Instance Variables
    int age;        // Instance Variables
    String color;   // Instance Variables
    void barking() // method 1
    {  }
    void hungry()   // method 2
    {  }
}
```

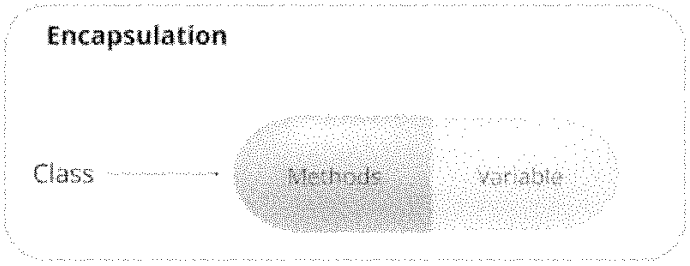
Abstraction

The abstraction mechanism allows us to represent a problem in a simpler way by considering only those aspects that are relevant to some purpose and omitting all other details that are irrelevant

Encapsulation

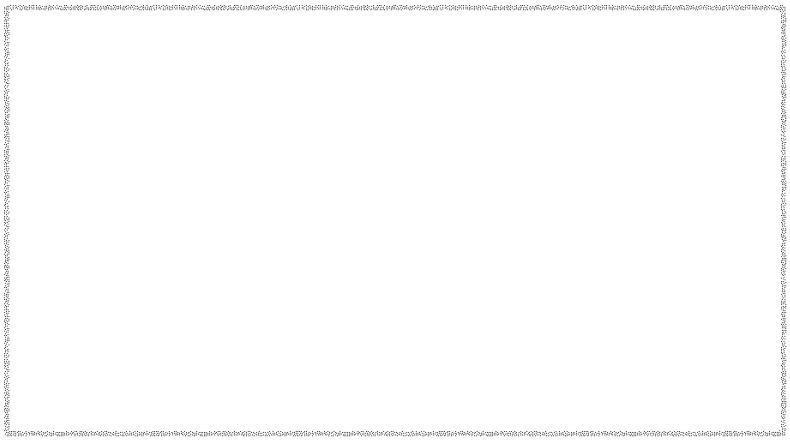
- Encapsulation is a mechanism where you bind your data and code together as a single unit.

- It also means to hide your data in order to make it safe from any modification.
- In a medical capsule, where the drug is always safe inside the capsule.
- Similarly, through encapsulation the methods and variables of a class are well hidden and safe.



Polymorphism

- Polymorphism means taking many forms, where 'poly' means many and 'morph' means forms.
- It is the ability of a variable, function or object to take on multiple forms
- polymorphism allows you define one interface or method and have multiple implementations.

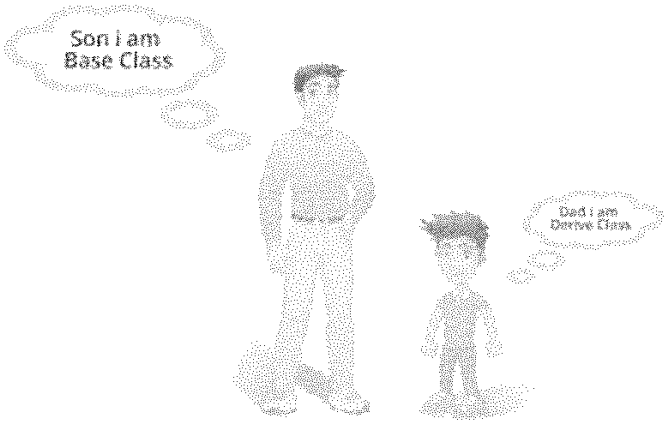


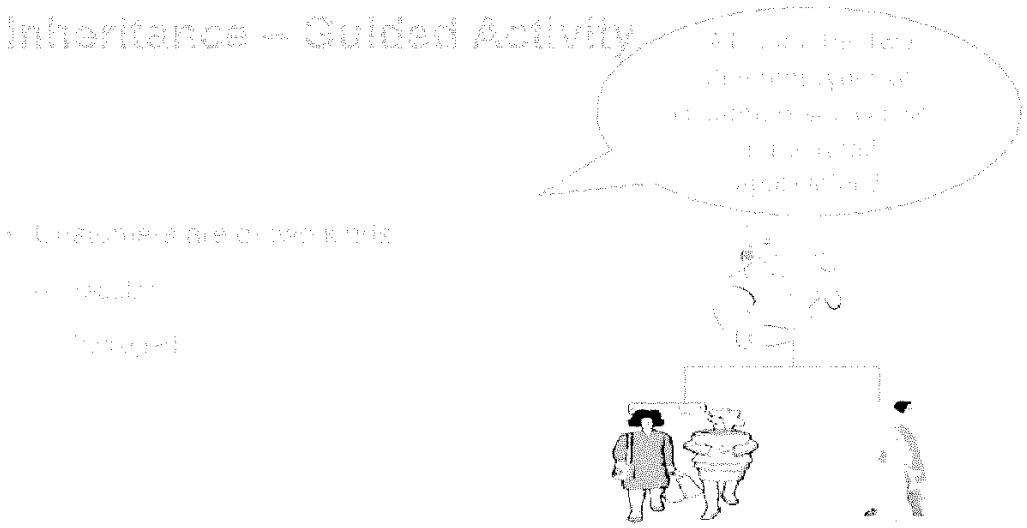
INHERITANCE

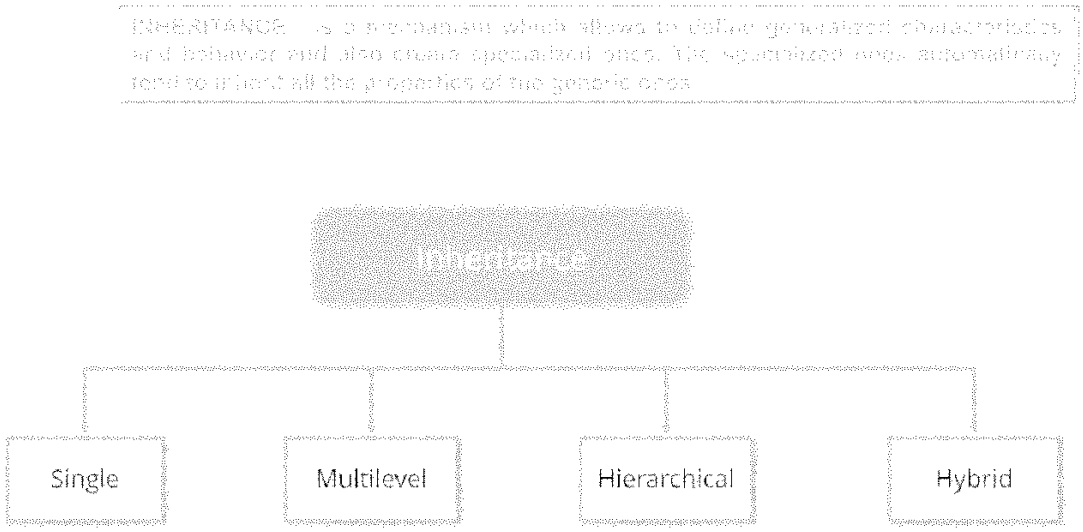
- Is the process by which objects of one class acquire the properties of objects of another class.
- Inheritance provides reusability.
- This means that we can add additional features to an existing class without modifying it.
- in Java, there are two classes:
    - Parent class (Super or Base class)

- Child class (Subclass or Derived class)

- A class which inherits the properties is known as Child Class whereas a class whose properties are inherited is known as Parent class.







Procedural and object oriented programming paradigm

Structure/Procedure oriented programming language

- In this type of language, large programs are divided into small programs called functions.
- Prime focus is on functions and procedures that operate on data
- Data moves freely around the systems from one function to another
- Program structure follows "Top Down Approach"
- Example: C, Pascal, ALGOL and Modula-2

Object oriented programming language

- In this type of language, programs are divided into objects
- Prime focus is on the data that is being operated and not on the functions or procedures
- Data is hidden and cannot be accessed by external functions
- Program structure follows "Bottom UP Approach"
- Example: C++, JAVA and C# (C sharp)

Procedure Oriented Programming (POP)	**Object Oriented Programming (OOP)**
• Importance is not given to data but to functions as well as sequence of actions to be done.	• Importance is given to the data rather than procedures or functions.
• Top Down approach in program design.	• Bottom Up approach in program design.
• Large programs are divided into smaller programs known as functions.	• Large programs are divided into classes and objects.
• POP does not have any access specifier.	• OOP has access specifiers named Public, Private,Protected, etc..
• Most functions use Global data for sharing that can be accessed freely from function to function in the system.	• Data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
• Adding data and function is difficult.	• Adding data and function is easy.
• Concepts like inheritance, polymorphism, abstraction,data encapsulation, access specifiers are missing.	• Concepts like inheritance, polymorphism, data encapsulation, abstraction, access specifier are available and can be used easily.
• Examples: C, Fortran, Pascal, etc…	• Examples: C++, Java, C#, etc…

Microservices Architecture in Java (and in general)

Microservices is an architectural style that structures an application as a collection of small, loosely coupled, independently deployable services. Each service is responsible for a specific business function and communicates with others through APIs (typically REST).

```
@RestController
@RequestMapping("/users")
public class UserController {
   @GetMapping("/{id}")
   public User getUser(@PathVariable String id) {
      return new User(id, "John Doe");
   }
}
```

Tools Often Used in Java Microservices

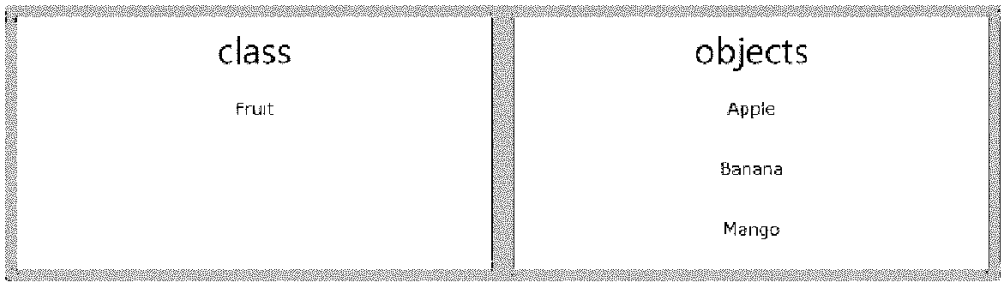
- **Spring Boot**: To build RESTful microservices quickly.
- **Spring Cloud**: For distributed systems components like config server, discovery, and load balancing.
- **Netflix OSS**: Tools like Eureka, Hystrix, Ribbon.
- **Docker**: For packaging microservices as containers.
- **Kubernetes**: For orchestration and scaling.

Object Oriented Programming in Java :-

Declaring Objects; Object Reference; Introduction to Methods; Constructors; Access Modifiers; this keyword.

Class Fundamentals

- The class is at the core of Java.
- It is the logical construct upon which the entire Java language is built
- Any concept you wish to implement in a Java program must be encapsulated within a class.
- Classes and objects are the two main aspects of object-oriented programming.
- It defines a new data type.
- Once defined, this new type can be used to create objects of that type.
- Class is a *template* for an object, and an object is an *instance* of a class.
- A Class is a "blueprint" for creating objects.



THE GENERAL FORM OF A CLASS

```
class classname
{
    type instance-variable1;
    // ...
    type instance-variableN;
    type methodname1 ( para
    meter-list)
    {
        // body of method
    }
    // ...
    type methodnameN (param
    eter-list)
    {
        // body of method
    }
}
```

- **Create a Class**
  - To create a class, use the keyword class

Example

```
/* A program
   that uses the
   Box class.
Call this file
   BoxDemo.java
*/
class Box
{
   double width;
   double height;
   double depth;
}
```

```
// This class
   declares an object
   of type Box.
class BoxDemo
{
   public static void
   main(String
   args[])
   {
Box mybox=new Box();
double vol;
//assign values to
   mybox's instance
   variables
mybox.width = 10;
mybox.height = 20;
mybox.depth = 15;
// compute volume of
   box
vol = mybox.width *
   mybox.height*
   mybox.depth;
System.out.println("Vo
   lume is " + vol);
 }
}
```

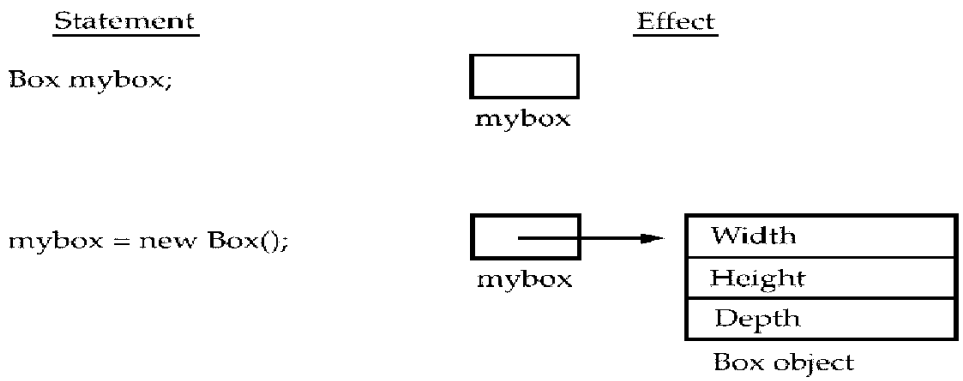
```
public class MyClass {
  int x = 5;
}
```

```
public class MyClass {
  int x = 5;

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

Declaring Objects

- Obtaining objects of a class is a two-step process
- First, you must declare a variable of the class type.
- It is simply a variable that can refer to an object.
- Second, you must acquire an actual, physical copy of the object and assign it to that variable.
- You can do this using the new operator
  - Box mybox; // declare reference to object
  - mybox = new Box(); // allocate a Box object

We can access attributes by creating an object of the class, and by using the dot syntax (.)



- The new operator dynamically allocates memory for an object.
- It has this general form:
  class-var = new classname ( );
- Here, class-var is a variable of the class type being created.
- The classname is the name of the class that is being instantiated.
- The class name followed by parentheses specifies the constructor for the class.
- A constructor defines what occurs when an object of a class is created.

Multiple Objects

You can create multiple objects of one class

```
public class MyClass {
  int x = 5;

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

Initialize the object through a reference variable

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

Output

101 sonoc

Modify Attributes

• We can also modify attribute values

Example

Set the value of x to 40

```
public class MyClass {
  int x;

  public static void main(String[] args) {
    MyClass myObj = new MyClass();
    myObj.x = 40;
    System.out.println(myObj.x);
  }
}
```

Output

40

Override existing values

Example

Change the value of x to 25

```
public class MyClass {
  int x = 10;

  public static void main(String[] args) {
    MyClass myObj = new MyClass();
    myObj.x = 25; // x is now 25
    System.out.println(myObj.x);
  }
}
```

Output

25

- If you don't want the ability to override existing values, declare the attribute as final
- The final keyword is useful when we want a variable to always store the same value, like PI (3.14159...)
- The final keyword is called a "modifier".

```
public class MyClass {
  final int x = 10;

  public static void main(String[] args) {
    MyClass myObj = new MyClass();
    myObj.x = 25; // will generate an error: cannot assign a value to a final variable
    System.out.println(myObj.x);
  }
}
```

Introducing Methods

- classes usually consist of two things: instance variables and methods.
- This is the general form of a method:

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

Create a method named myMethod() in MyClass
myMethod() prints a text (the action), when it is called.

```
public class MyClass {
  static void myMethod() {
    System.out.println("Hello World!");
  }
}
```

Adding a Method to the Box Class

```
// This program includes a method inside the box class.
class Box
{
    double width;
    double height;
    double depth;
    // display volume of a box
    void volume()
    {
        System.out.print("Volume is ");
        System.out.println(width * height * depth);
    }
}
```

```
class BoxDemo3
{
    public static void main(String args[])
    {
            Box mybox1 = new Box();
            Box mybox2 = new Box();
            // assign values to mybox1's instance variables
            mybox1.width = 10;
            mybox1.height = 20;
            mybox1.depth = 15;



   /* assign different values to mybox2's
   instance variables */
   mybox2.width = 3;
   mybox2.height = 6;
   mybox2.depth = 9;
   // display volume of first box
   mybox1.volume();
   // display volume of second box
   mybox2.volume();
   }
}
```

Returning a Value

```
// Now, volume() returns the volume of a box.
class Box
{
    double width;
    double height;
    double depth;
    // compute and return volume
    double volume()
    {
        return width * height * depth;
    }
}


 class BoxDemo4
 {
    public static void main(String args[])
    {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;
    // assign values to mybox1's instance variables
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;
```

```
    /* assign different values to mybox2's
    instance variables */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;
    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);
    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
    }
}
```

Adding a Method That Takes Parameters

```
// This program uses a parameterized method.
class Box
{
   double width;
   double height;
   double depth;
   // compute and return volume
   double volume()
   {
      return width * height * depth;
   }

   // sets dimensions of box
   void setDim(double w, double h, double d)
   {
      width = w;
      height = h;
      depth = d;
   }
}
```

```
class BoxDemo5
{
   public static void main(String args[])
   {
      Box mybox1 = new Box();
      Box mybox2 = new Box();
      double vol;
      // initialize each box
      mybox1.setDim(10, 20, 15);
      mybox2.setDim(3, 6, 9);


      // get volume of first box
      vol = mybox1.volume();
      System.out.println("Volume is " + vol);
      // get volume of second box
      vol = mybox2.volume();
      System.out.println("Volume is " + vol);
   }
}
```

- The dot (.) is used to access the object's attributes and methods.
- To call a method in Java, write the method name followed by a set of parentheses (), followed by a semicolon (;)

Constructors

- A constructor initializes an object immediately upon creation.
- It has the same name as the class in which it resides and is syntactically similar to a method.
- Once defined, the constructor is automatically called when the object is created.
- Constructors look a little strange because they have no return type, not even void.
- This is because the implicit return type of a class constructor is the class type itself.
- Types of Java constructors
  - Default Constructor
  - No-Args constructor
  - Parameterized constructor

```
public class MyClass{
    // Constructor
    MyClass(){
        System.out.println("BeginnersBook.com");
    }
    public static void main(String args[]){
        MyClass obj = new MyClass();
        ...
    }
}
```

New keyword creates the object of MyClass & invokes the constructor to initialize the created object.

```
/* Here, Box uses a constructor to initialize the dimensions of
   a box. */
class Box
{
   double width;
   double height;
   double depth;
   // This is the constructor for Box.
   Box()
   {
     width = 10;
     height = 10;
     depth = 10;
   }

    // compute and return volume
    double volume()
    {
      return width * height * depth;
    }
 }
 class BoxDemo6
 {
    public static void main(String args[])
    {
    // declare, allocate, and initialize Box  objects
    Box mybox1 = new Box();
    Box mybox2 = new Box();



    double vol;
    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);
    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
    }
 }
```

Parameterized Constructors

- While the Box( ) constructor in the preceding example does initialize a Box object, it is not
- very useful—all boxes have the same dimensions.
- What is needed is a way to construct Box objects of various dimensions.
- The easy solution is to add parameters to the constructor.

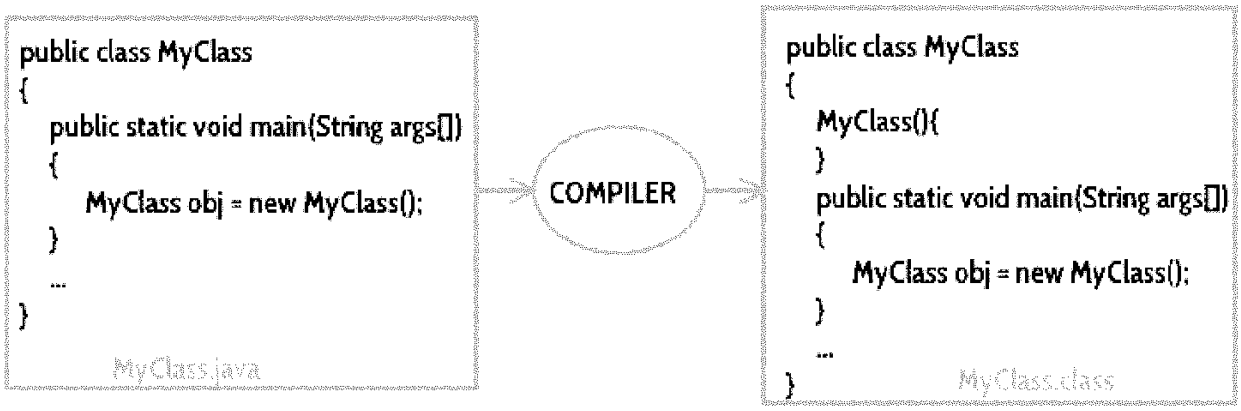
```
/* Here, Box uses a parameterized constructor to
initialize the dimensions of a box.
*/
class Box
{
   double width;
   double height;
   double depth;
   // This is the constructor for Box.
   Box(double w, double h, double d)
   {
     width = w;
     height = h;
     depth = d;
   }


  // compute and return volume
  double volume()
  {
    return width * height * depth;
  }
}

class BoxDemo7
{
   public static void main(String args[])
   {
     // declare, allocate, and initialize Box objects
     Box mybox1 = new Box(10, 20, 15);
     Box mybox2 = new Box(3, 6, 9);
     double vol;
     // get volume of first box
     vol = mybox1.volume();
     System.out.println("Volume is " + vol);
```

```
        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

Default constructor

- If you do not implement any constructor in your class, Java compiler inserts a default constructor into your code on your behalf.
- This constructor is known as the default constructor.
- You would not find it in your source code(the java file) as it would be inserted into the code during compilation and exists in .class file.

If you implement any constructor then you no longer receive a default constructor from the Java compiler.



- The purpose of a default constructor
  - The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

```
// Create a MyClass class
public class MyClass {
  int x;  // Create a class attribute

  // Create a class constructor for the MyClass class
  public MyClass() {
    x = 5;  // Set the initial value for the class attribute x
  }

  public static void main(String[] args) {
    MyClass myObj = new MyClass(); // Create an object of class MyClass (This will call the constructor)
    System.out.println(myObj.x); // Print the value of x
  }
}

// Outputs 5
```

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

Access Modifiers

Access Control Keywords

- The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.
- There are two types of modifiers in java:
  1. access modifiers and
  2. non-access modifiers.
- Java's access modifiers are public, private, and protected.
- protected applies only when inheritance is involved
- There are many non-access modifiers such as static, abstract, final, synchronized etc.
- Methods and Variables within a class are collectively known as members.
- Every member you declare has an access control, whether you explicitly type one or not.
- When no access modifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.
- The four access levels are –
  - Visible to the package, the default. No modifiers are needed.
  - Visible to the class only (private).
  - Visible to the world (public).
  - Visible to the package and all subclasses (protected).

Private

- The private access modifier is specified using the keyword private.
- The methods or data members declared as private are accessible only within the class in which they are declared.
- Any other class of same package will not be able to access these members.
- Private members are not visible within subclasses, and are not inherited.
- Classes or interfaces can not be declared as private.

```
//Java program to illustrate error while
//using class from different package with
//private modifier
package p1;

class A
{
   private void display()
    {
        System.out.println("GeeksforGeeks");
    }
}

class B
{
   public static void main(String args[])
      {
          A obj = new A();
          //trying to access private method of another class
          obj.display();
      }
}
```

Output:

```
error: display() has private access in A
                    obj.display();
```

- In this example, we will create two classes A and B within the same package p1. We will declare a method in class A as private and try to access this method from class B.

Default

- When we do not mention any access modifier, it is called default access modifier.
- The data members, class or methods which are not declared using any access modifiers i.e. having default access modifier are accessible only within the same package.
- This means that if we have a class with the default access modifier in a package, only those classes that are in this package can access this class.
- No other class outside this package can access this class.
- Similarly, if we have a default method or data member in a class, it would not be visible in the class of another package.
- In this example we have two classes, Test class is trying to access the default method of Addition class, since class Test belongs to a different package, this program would throw compilation error, because the scope of default modifier is limited to the same package in which it is declared.

```
package abcpackage;

public class Addition {
   /* Since we didn't mention any access modifier here, it would
    * be considered as default.
    */
   int addTwoNumbers(int a, int b){
        return a+b;
   }
}

/* We are importing the abcpackage
 * but still we will get error because the
 * class we are trying to use has default access
 * modifier.
 */
import abcpackage.*;
public class Test {
   public static void main(String args[]){
        Addition obj = new Addition();
        /* It will throw error because we are trying to access
         * the default method in another package
         */
        obj.addTwoNumbers(10, 21);
   }
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The method addTwoNumbers(int, int) from the type Addition is not visible
at xyzpackage.Test.main(Test.java:12)
```

Protected

- The protected access modifier is specified using the keyword protected.
- Protected data members and methods are only accessible by the classes of the same package and the subclasses present in any package.
- You can also say that the protected access modifier is similar to default access modifier with one exception that it has visibility in subclasses.
- The protected access specifier cannot be applied to class and interfaces
- In this example the class Test which is present in another package is able to call the addTwoNumbers() method, which is declared protected. This is because the Test class

extends class Addition and the protected modifier allows the access of protected members in subclasses (in any packages).

```
package abcpackage;
public class Addition {

    protected int addTwoNumbers(int a, int b){
        return a+b;
    }
}
```

```
package xyzpackage;
import abcpackage.*;
class Test extends Addition{
   public static void main(String args[]){
        Test obj = new Test();
        System.out.println(obj.addTwoNumbers(11, 22));
   }
}
```

Output:

33

Public

- The public access modifier is specified using the keyword public.
- The public access modifier has the widest scope among all other access modifiers.
- Classes, methods, and data members declared as public can be accessed from any class in the Java program, whether these classes are in the same package or in another package.
- There is no restriction on the scope of public data members.

Let's take the same example that we have seen above but this time the method addTwoNumbers() has a public modifier and class Test is able to access this method without even extending the Addition class. This is because the public modifier has visibility everywhere.

```
package abcpackage;

public class Addition {

    public int addTwoNumbers(int a, int b){
        return a+b;
    }
}
```

```
package xyzpackage;
import abcpackage.*;
class Test{
   public static void main(String args[]){
      Addition obj = new Addition();
      System.out.println(obj.addTwoNumbers(100, 1));
   }
}
```

Output:

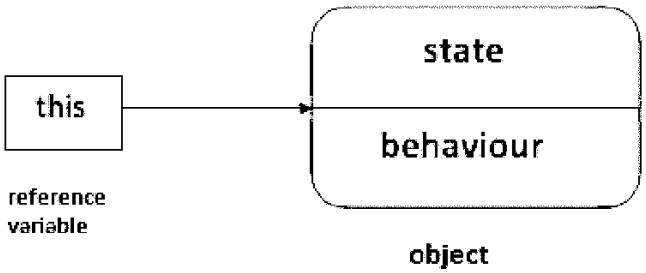
```
101
```

The scope of access modifiers in tabular form

	Class	Package	Subclass (same package)	Subclass (diff package)	Outside Class
public	Yes	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	Yes	No
default	Yes	Yes	Yes	No	No
private	Yes	No	No	No	No

This keyword

- Sometimes a method will need to refer to the object that invoked it.
- To allow this, Java defines this keyword. this can be used inside any method to refer to the current object.



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

```
// A redundant use of this.
Box(double w, double h, double d)
{
   this.width = w;
   this.height = h;
   this.depth = d;
}
```

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

```
class TestThis2
{
    public static void main(String args[])
    {

       Student s1=new Student(111,"an
    kit",5000f);

       Student s2=new Student(112,"su
    mit",6000f);
       s1.display();
       s2.display();
    }
}
```

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

```
class A{
void m(){System.out.println("hello m");}
void n(){
System.out.println("hello n");
//m();//same as this.m()
this.m();
}
}
class TestThis4{
public static void main(String args[]){
A a=new A();
a.n();
}}
```

Output

```
hello n
hello m
```

Short Questions

1. What are the essential components of a simple Java program (e.g., class declaration, `main` method)?
2. Briefly differentiate between the Java Runtime Environment (JRE) and the Java Virtual Machine (JVM).
3. What is the primary function of the Java compiler (`javac`)?
4. List any three primitive data types in Java and their typical uses.
5. What are wrapper types in Java? Give an example for an `int` primitive.
6. When would you use explicit type casting in Java? Provide a simple example.
7. What is autoboxing in Java?
8. How do you declare and initialize a one-dimensional array in Java?
9. Are `String` objects mutable in Java? Explain briefly.
10. What is the `Vector` class in Java? Mention one of its key characteristics.
11. List any three arithmetic operators in Java.
12. What do relational operators do? Give an example.
13. Name two boolean logical operators.
14. What is the purpose of the assignment operator (=)?
15. Explain the syntax and basic usage of the conditional (ternary) operator.
16. Write a simple `if-else` statement to check if a number is positive or negative.
17. How do you write a basic `for` loop to iterate from 1 to 5?
18. What is the purpose of the `break` statement in a loop?
19. What is a "function" (or method) in Java?
20. How can you access command-line arguments in a Java program?
21. Define what a "class" is in object-oriented programming.
22. Briefly explain data abstraction in OOP.
23. What is encapsulation?
24. What is the primary use of the `this` keyword in Java?
25. What is the visibility of members declared with the `public` access modifier?

Essay Questions

1. Explain the roles of the Java Development Kit (JDK), Java Runtime Environment (JRE), and Java Virtual Machine (JVM) in the Java ecosystem.

2. Compare and contrast developing Java programs using an Integrated Development Environment (IDE) versus using the command line. Mention advantages and disadvantages of each.
3. Differentiate between primitive data types and their corresponding wrapper types. When would you prefer using wrapper types over primitives?
4. Explain autoboxing and unboxing with a clear code example demonstrating both concepts.
5. Explain multi-dimensional arrays in Java. Declare and initialize a 2D array and print its elements.
6. Discuss different categories of operators in Java (e.g., arithmetic, logical, relational). Provide an example demonstrating operator precedence.
7. Explain the working of `switch` statements in Java. Provide a code snippet for a simple `switch` case. When would you prefer to switch over `if-else if`?
8. Explain the `while` loop and `do-while` loop with examples. Discuss when to use each.
9. Explain the `break` and `continue` statements with suitable examples in the context of loops.
10. Write a Java program that defines a method `calculateArea` that takes two `double` parameters (length, width) and returns their product. Demonstrate how to call this method from `main`.
11. Write a Java program that accepts two integer command-line arguments, calculates their sum, and prints the result. Include error handling for invalid input.
12. Explain the concept of variable-length arguments (varargs) in Java. Provide a method signature and an example usage of varargs.
13. Define "class" and "object". Write a simple Java class `Car` with fields like `make`, `model`, and `year`. Then, demonstrate how to declare and instantiate an object of `Car` and access its fields.
14. What is a constructor in Java? Explain the purpose of a default constructor and a parameterized constructor with an example.
15. Explain the `private` and `default` (package-private) access modifiers with examples, discussing their scope and visibility.
16. Differentiate between the procedural programming paradigm and the object-oriented programming paradigm. Highlight the advantages of OOP.
17. What are microservices in software architecture? How do they relate to the concept of modularity in programming?
18. Discuss the complete lifecycle of a Java program from source code creation to execution. Include the roles of the compiler, JVM, JRE, and the operating system. You may use a diagram to illustrate the flow.
19. Elaborate on explicit and implicit type casting in Java. Provide scenarios where explicit casting is necessary. Then, discuss the performance implications and potential pitfalls of excessive autoboxing/unboxing.
20. Compare and contrast Java arrays with the `Vector` class. Discuss their similarities, differences (fixed size vs. dynamic, synchronization), and when you would choose one over the other.
21. Explain all three categories of control statements in Java (Selection, Iteration, Jump) with a detailed example for at least two statements from each category. Discuss best practices for using them effectively.
22. Explain method overloading in Java. Write a class `Calculator` with overloaded methods for `add` (taking different numbers/types of arguments). Discuss the importance of clear method naming and parameter conventions.

23. Design a `Student` class with fields (`studentId`, `name`, `grades`). Include a parameterized constructor, a method to calculate the average grade, and a method to display student details. Demonstrate creating multiple `Student` objects and using their methods.
24. Compare and contrast abstract classes and interfaces in Java. Discuss their similarities, key differences, and when you would choose to use one over the other. Provide a small example for each.
25. Provide a comprehensive explanation of all four access modifiers in Java (`public`, `private`, `protected`, `default`). Give specific examples for each modifier to illustrate their scope and impact on accessibility within classes, packages, and inheritance hierarchies.
26. Explain the four core principles of Object-Oriented Programming (Data Abstraction, Encapsulation, Inheritance, Polymorphism) in detail. For each principle, provide a real-world analogy and a corresponding small Java code snippet to illustrate its application.
27. Explain the role of constructors in object initialization. Discuss how the `this` keyword is used to refer to current class instance variables, to call other constructors within the same class (`this()`), and to pass the current object as an argument. Provide code examples for each usage.

Module 2

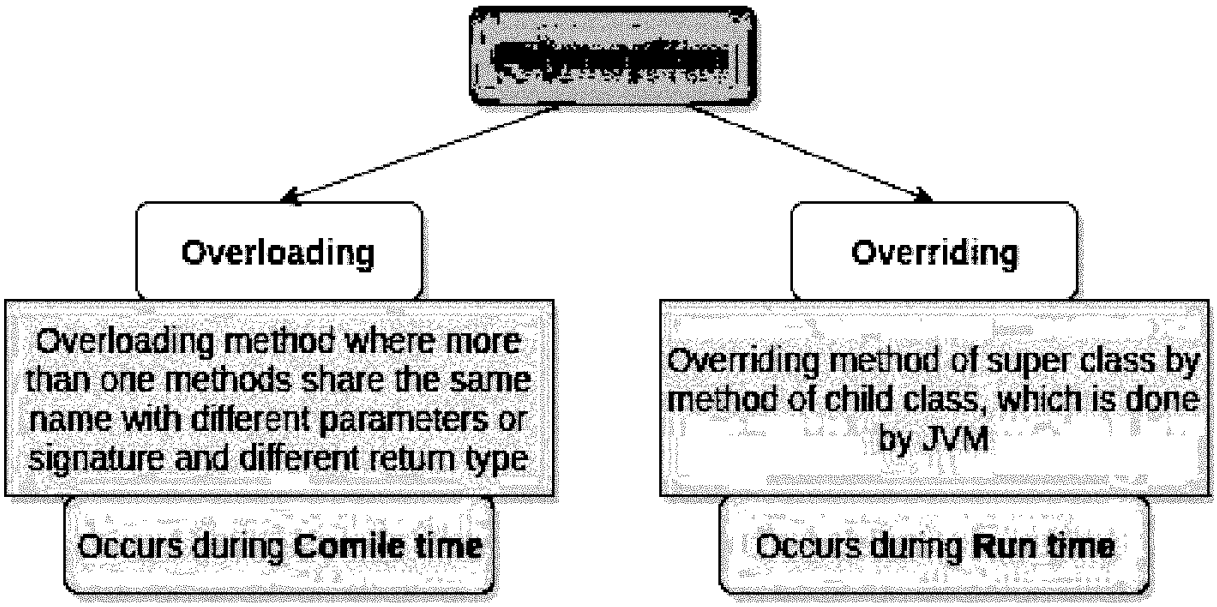
Polymorphism :

Method Overloading, Using Objects as Parameters, Returning Objects, Recursion. Static Members, Final Variables, Inner Classes.

Inheritance - Super Class, Sub Class, Types of Inheritance, The super keyword, protected Members, Calling Order of Constructors. Method Overriding, Dynamic Method Dispatch, Using final with Inheritance.

Polymorphism :-

Polymorphism, a core concept in object-oriented programming (OOP), allows objects to take on different forms or behaviors. In Java, it enables a single action to behave differently based on the object performing it. This feature promotes flexibility, code reusability, and scalability in applications.



Method Overloading

- With method overloading, multiple methods can have the same name with different parameters
- Method overloading is one of the ways that Java supports polymorphism.
- There are different ways to overload the method in java

- By changing number of arguments
- By changing the data type
- methods may have different return types, which is insufficient to distinguish two versions of a method

- **Advantage of method overloading**
  - The main advantage of this is cleanliness of code.
  - Method overloading increases the readability of the program.
  - Flexibility

```
int myMethod(int x)
float myMethod(float x)
double myMethod(double x, double y)
```

Example

```
class OverloadDemo
{
        void test()
        {
                System.out.println("Noparameters");
        }
// Overload test for one integer parameter.
        void test(int a)
        {
                System.out.println("a: " + a);
        }
// Overload test for two integer parameters.
        void test(int a, int b)
        {
                System.out.println("a and b: " + a + " " + b);
        }
        // Overload test for a double parameter
        double test(double a)
        {
```

```
            System.out.println("double a: " + a);
            return a*a;
        }
}
class Overload
{
        public static void main(String args[])
        {
            OverloadDemo ob = new OverloadDemo();
            double result;
            // call all versions of test()
            ob.test();
            ob.test(10);
            ob.test(10, 20);
            result = ob.test(123.25);
            System.out.println("Result of ob.test(123.25): " + result);
        }
}
```

Example - Consider the following example, which have two methods that add numbers of different type

```
static int plusMethodInt(int x, int y) {
  return x + y;
}

static double plusMethodDouble(double x, double y) {
  return x + y;
}

public static void main(String[] args) {
  int myNum1 = plusMethodInt(8, 5);
  double myNum2 = plusMethodDouble(4.3, 6.26);
  System.out.println("int: " + myNum1);
  System.out.println("double: " + myNum2);
}
```

Instead of defining two methods that should do the same thing, it is better to overload one.

```
static int plusMethod(int x, int y) {
  return x + y;
}

static double plusMethod(double x, double y) {
  return x + y;
}

public static void main(String[] args) {
  int myNum1 = plusMethod(8, 5);
  double myNum2 = plusMethod(4.3, 6.26);
  System.out.println("int: " + myNum1);
  System.out.println("double: " + myNum2);
}
```

Valid/invalid cases of method overloading

Example 1:

int mymethod(int a, int b, float c)

int mymethod(int var1, int var2, float var3)

- Result: **Compile time error**. Argument lists are exactly the same. Both methods have the same number, data types and same sequence of data types.

Example 2:

float mymethod(int a, float b)

float mymethod(float var1, int var2)

- Result: **Perfectly fine**. Valid case of overloading. Sequence of the data types of parameters are different, the first method is having (int, float) and second is having (float, int).

Example 3:

float mymethod(int a, float b)

float mymethod(float var1, int var2)

- Result: **Perfectly fine**. Valid case of overloading. Sequence of the data types of parameters are different, the first method is having (int, float) and second is having (float, int).

Overloading Constructors

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

```
public class Demo {
  Demo( ) {
  ..
  }
  Demo(String s) {
  ...
  }
  Demo(int i) {
  ...
  }
.....
}
```

Three overloaded constructors - They must have different Parameter list

Overloading Constructors

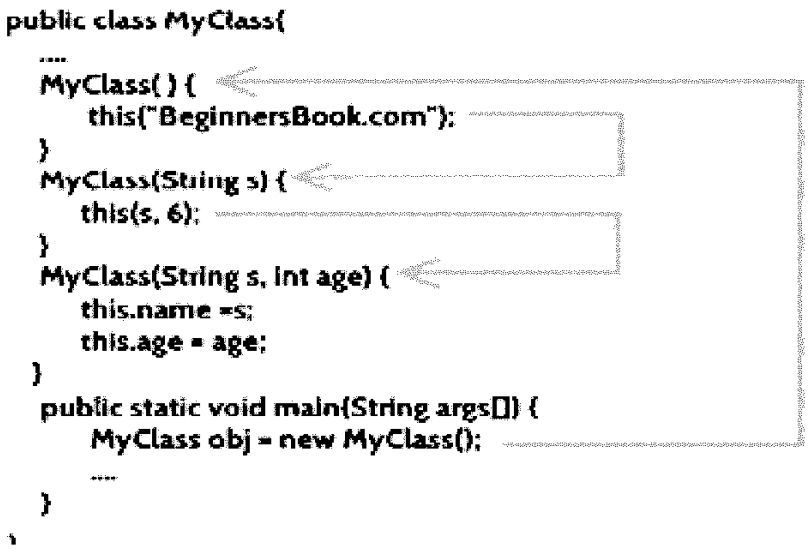
```
// constructor used
   when all dimensions
   specified
Box(double w, double h,
   double d)
{
   width = w;
   height = h;
   depth = d;
}
Box mybox1 = new Box(10, 20, 15);
```

```
// constructor used
   when no dimensions
   specified
Box()
{
   width = -1;
// use -1 to indicate an
   uninitialized box
   height = -1; //
   depth = -1; //
}
Box mybox2 = new Box();
```

```
// constructor used
   when cube is
   created
Box(double len)
{
   width = height =
   depth = len;
}
Box mybox2 = new Box(7);
```

Constructor Chaining

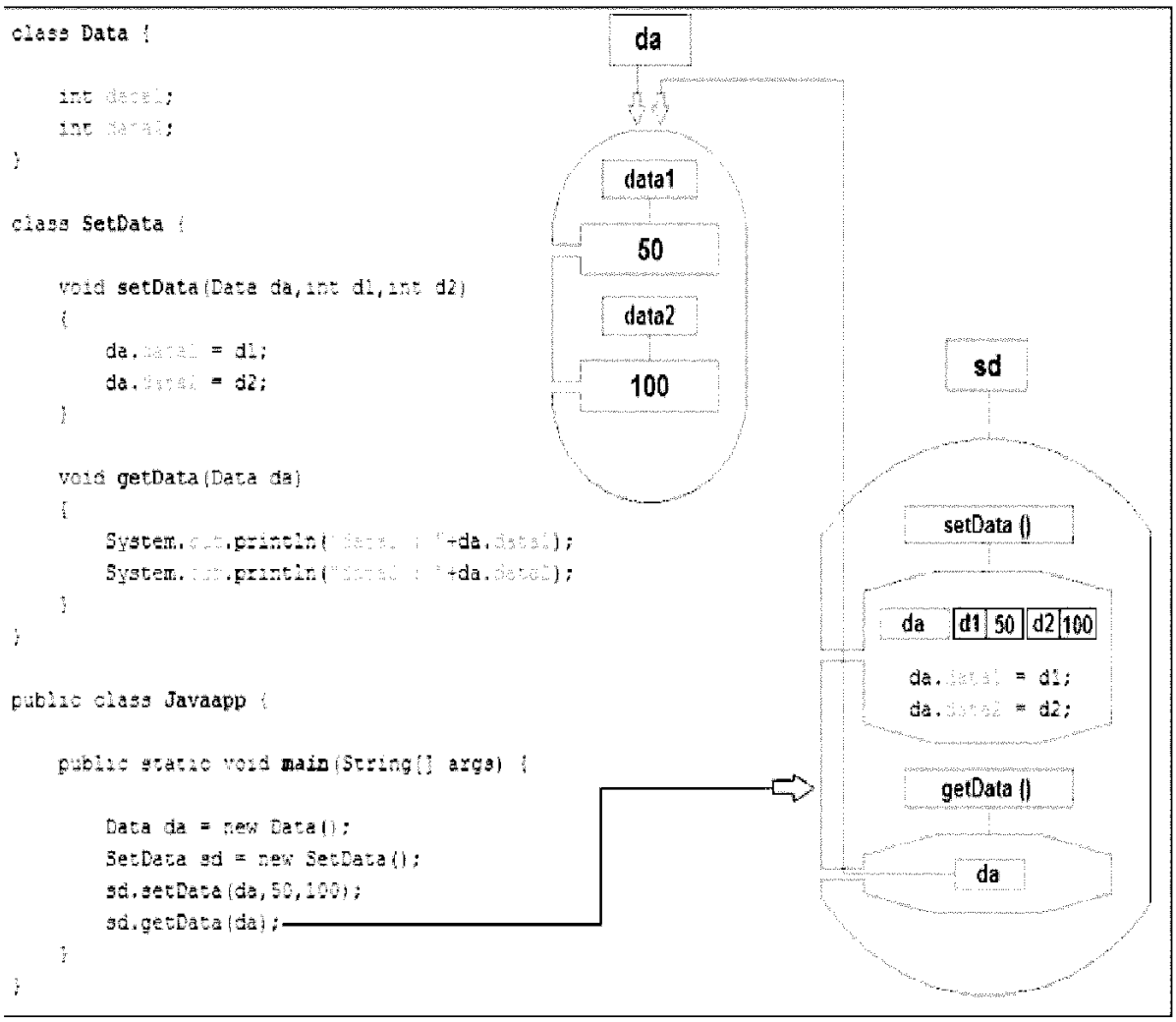
When A constructor calls another constructor of the same class then this is called constructor chaining.



constructors with fewer arguments should call those with more.

Using Objects as Parameters

- A method can take an object as a parameter.
- when we pass an object to a method, the objects are passed by call-by-reference.
- Java does this interesting thing that's sort of a hybrid between pass-by-value and pass-by-reference.
- While creating a variable of a class type, we only create a reference to an object.
- Thus, when we pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument.
- This effectively means that objects act as if they are passed to methods by use of call-by-reference.
- Changes to the object inside the method do reflect in the object used as an argument.




```
// Objects may be passed to methods
class Test
{
    int a, b;
    Test(int i, int j)
    {
        a = i;
        b = j;
    }
// return true if o is equal to the invoking object
boolean equalTo(Test o)
{
    if(o.a == a && o.b == b) return true;
    else return false;
}
}
```

```
class PassOb
{
    public static void main(String args[])
    {
        Test ob1 = new Test(100, 22);
        Test ob2 = new Test(100, 22);
        Test ob3 = new Test(-1, -1);
        System.out.println("ob1 == ob2: "
    + ob1.equalTo(ob2));
        System.out.println("ob1 == ob3: "
    + ob1.equalTo(ob3));
    }
}
```

USING OBJECT AS A PARAMETER / ARGUMENT

```
class Operation2{
 int data=50;

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


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

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

 }
}
```

```
Output:before change 50
       after change 150
```

Returning Objects

- In java, a method can return any type of data, including objects

// Java program to demonstrate returning of objects

```
class ObjectReturnDemo
{
    int a;
    ObjectReturnDemo(int i)
    {
            a = i;
    }
    // This method returns an object
    ObjectReturnDemo incrByTen()
    {
            ObjectReturnDemo temp = new ObjectReturnDemo(a+10);
            return temp;
    }
}
```

```
public class Test
{
    public static void main(String args[])
    {
            ObjectReturnDemo ob1 = new ObjectReturnDemo(2);
            ObjectReturnDemo ob2;
            ob2 = ob1.incrByTen();
            System.out.println("ob1.a: " + ob1.a);
            System.out.println("ob2.a: " + ob2.a);
    }
}
```

Recursion

- Recursion is the technique of making a method call itself.
- This technique provides a way to break complicated problems down into simple problems which are easier to solve.

Syntax

```
returntype methodname(){
//code to be executed
methodname();//calling same method
}
```

```
public class MyClass {
  public static void main(String[] args) {
    int result = sum(10);
    System.out.println(result);
  }
  public static int sum(int k) {
    if (k > 0) {
      return k + sum(k - 1);
    } else {
      return 0;
    }
  }
}
```

Working

```
10 + sum(9)
10 + ( 9 + sum(8) )
10 + (9 + ( 8 + sum(7) ) )
...
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + sum(0)
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0
```

Example

Use recursion to add all of the numbers up to 10.

```
public class RecursionExample3 {
   static int factorial(int n){
       if (n == 1)
        return 1;
       else
        return(n * factorial(n-1));
   }

public static void main(String[] args) {
System.out.println("Factorial of 5 is: "+factorial(5));
}
}
```

Working

```
factorial(5)
   factorial(4)
      factorial(3)
         factorial(2)
            factorial(1)
               return 1
            return 2*1 = 2
         return 3*2 = 6
      return 4*6 = 24
   return 5*24 = 120
```

Example

Factorial of a number

Static Members

Java static keyword

- The static keyword in java is used for memory management mainly.
- We can apply a Java static keyword with variables, methods, blocks and nested class.
- The static can be:
  - variable
  - method
  - block

Java static variable

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

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

Advantage of static variable

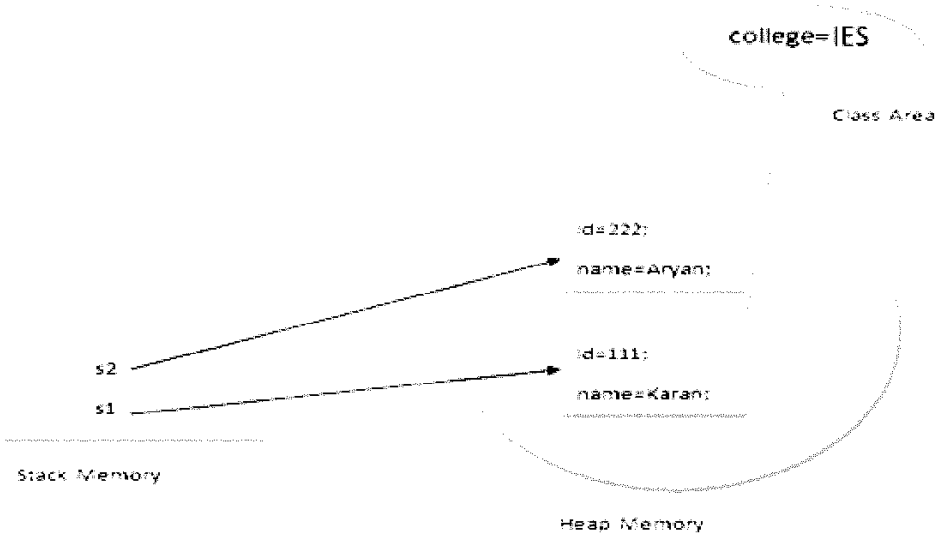
- It makes your program memory efficient (i.e it saves memory).

```
class Student
{
    int rollno;
    String name;
    String college="IES";
}
```

Advantage of static variable

- Suppose there are 500 students in my college, now all instance data members will get memory each time an object is created.
- All students have its unique rollno and name so instance data member is good.
- Here, college refers to the common property of all objects.
- If we make it static,this field will get memory only once.

```
//Program of static variable
 class Student
{
        int rollno;
        String name;
        static String college ="IES";
        Student(int r,String n)
        {
             rollno = r;
             name = n;
        }          void display ()
        {
                   System.out.println(rollno+" "+name+" "+college);
        }
        public static void main(String args[])
        {
                   Student s1 = new Student(111,"Karan");
                   Student s2 = new Student(222,"Aryan");
                   s1.display();
                   s2.display();
        }
}
```



Program of counter without static variable

```
class Counter
{
    int count=0;//will get memory when instance is created
    Counter()
    {
        count++;
        System.out.println(count);
    }
    public static void main(String args[])
    {
        Counter c1=new Counter();
        Counter c2=new Counter();
        Counter c3=new Counter();
   }
}
```

Output: 1
1
1

Java static method

- If you apply a static keyword with any method, it is known as a static method.
  - A static method belongs to the class rather than the object of a class.
  - A static method can be invoked without the need for creating an instance of a class.
  - static method can access static data members and can change the value of it.

Java static method

```
class Student
{
   int rollno;
   String name;
   static String college = "IES";
   static void change()
   {
       college = "IESCE";
   }
```

```
    Student(int r, String n)
    {
        rollno = r;
        name = n;
    }
void display ()
        {
        System.out.println(rollno+" “
                +name+" "+college);
        }
   public static void main(String args[])
   {
   Student.change();
   Student s1 = new Student (111,"Karan");
   Student s2 = new Student (222,"Aryan");
   Student s3 = new Student (333,"Sonoo");
   s1.display();
   s2.display();
   s3.display();
   }
}
```

Restrictions for static method

- There are two main restrictions for the static method. They are:
    - The static method cannot use non-static data members or call non-static methods directly.
    - this and super cannot be used in a static context.

Java static block

- Is used to initialize the static data member.
- It is executed before the main method at the time of classloading.

```
class A2{
  static{System.out.println("static block is invoked");}
```

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

Inner Classes

- In Java, it is also possible to nest classes (a class within a class).
- The purpose of nested classes is to group classes that belong together, which makes your code more readable and maintainable.
- To access the inner class, create an object of the outer class, and then create an object of the inner class

```
class OuterClass {
  int x = 10;

  class InnerClass {
    int y = 5;
  }
}

public class MyMainClass {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}

// Outputs 15 (5 + 10)
```

```
class OuterClass {
  int x = 10;

  class InnerClass {
    public int myInnerMethod() {
      return x;
    }
  }
}

public class MyMainClass {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.myInnerMethod());
  }
}

// Outputs 10
```

- Access Outer Class From Inner Class
- One advantage of inner classes is that they can access attributes and methods of the outer class

Inheritance

- Inheritance is a mechanism in which one class acquires the property of another class.
- For example, a child inherits the traits of his/her parents.
- With inheritance, we can reuse the fields and methods of the existing class.
- Reusability is an important concept of OOPs.
- In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.
- The meaning of "extends" is to increase the functionality.
- Inheritance represents the IS-A relationship which is also known as a parent-child relationship.
- **Super Class/Parent Class**:
  - Superclass is the class from where a subclass inherits the features.
  - It is also called a base class or a parent class.
- **Sub Class/Child Class**:
  - Subclass is a class which inherits the other class.
  - It is also called a derived class, extended class, or child class.
- The keyword used for inheritance is **extends**.
- Syntax :

class derived-class **extends** base-class

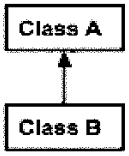
{

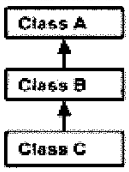
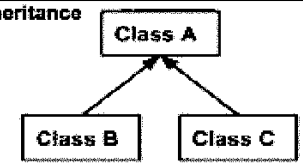
//methods and fields

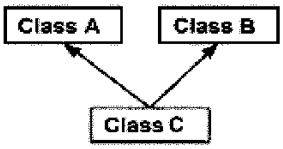
}

```
class Doctor
{
  void DoctorDetails()
   {
    System.out.println("Doctor Details...");
   }
}
class Surgeon extends Doctor
{
  void SurgeonDetails()
   {
    System.out.println("Surgen Detail...");
  }
}
```

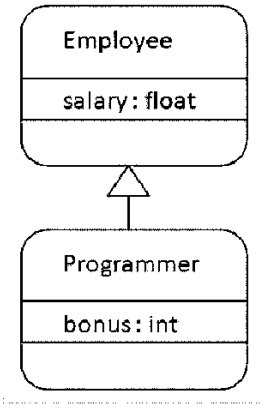
```
public class Hospital
{
  public static void main(String args[])
  {
    Surgeon s = new Surgeon();
    s.DoctorDetails();
    s.SurgeonDetails();
  }
}
```

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

Multi Level Inheritance 	public class A { .....} public class B extends A {.....} public class C extends B {..... }
Hierarchical Inheritance 	public class A { .....} public class B extends A {.....} public class C extends A {..... }

Multiple Inheritance
 
	public class A { .....}
	public class B {.....}
	public class C extends A,B {
	.....
	} // Java does not support multiple inheritance

- Programmer is the subclass (child class)
- Employee is the superclass (Parent class)
- The relationship between the two classes is Programmer IS-A Employee
- It means that Programmer is a type of Employee.



```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 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
```

- Programmer object can access the attribute of its own class as well as of Employee class i.e. code reusability.

The **Keyword super**

- The **super** keyword refers to superclass (parent) objects.
- It is used to call superclass methods, and to access the superclass constructor.
- The most common use of the super keyword is to eliminate the confusion between superclasses and subclasses that have methods with the same name.
- super can be used to refer to an immediate parent class instance variable.
- super can be used to invoke the immediate parent class method.
- super() can be used to invoke immediate parent class constructors.
- This scenario occurs when a derived class and base class have the same data members.
- In that case there is a possibility of ambiguity for the JVM.

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

output

```
black
white
```

Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

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

Output

```
eating...
barking...
```

The super keyword can also be used to invoke(call) parent class method.

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

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

Output

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

Java Interface

An interface in Java is a blueprint of a class. It is used to achieve abstraction and multiple inheritance in Java. An interface can only have:

- Abstract methods (until Java 7)
- Default and static methods (from Java 8)
- Private methods (from Java 9)

```
interface Animal {
    void sound(); // abstract method
}

class Dog implements Animal {
    public void sound() {
        System.out.println("Dog barks");
    }
}
```

Multiple Interfaces

Java does not support multiple inheritance with classes, but supports it with interfaces.

```
interface A {
    void show();
}

interface B {
    void display();
}

class C implements A, B {
    public void show() {
        System.out.println("Show from A");
    }

    public void display() {
        System.out.println("Display from B");
    }
}
```

Final Keyword

- The final keyword in java is used to restrict the user.
- The java final keyword can be used in many contexts. Final can be:
    1. Variable
    2. Method
    3. Class

Final Variables

- final variables are nothing but constants.
- We cannot change the value of a final variable once it is initialized.
- *It is considered as a good practice to have constant names in UPPERCASE(CAPS).*

```
class Demo
{
  final int MAX_VALUE=99;
  void myMethod()
  {
    MAX_VALUE=101;
  }
  public static void main(String args[])
  {
    Demo obj=new  Demo();
    obj.myMethod();
  }
}
```

Output

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

The final field Demo.MAX_VALUE cannot be assigned at beginnersbook.com.Demo.myMethod(Details.java:6)

at beginnersbook.com.Demo.main(Details.java:10)

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

Output:Compile Time Error

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

- **Blank final variable**
  - A final variable that is not initialized at the time of declaration is known as blank final variable.
  - We must initialize the blank final variable in constructor of the class otherwise it will throw a compilation error

```
class Demo
{
      final int MAX_VALUE; //Blank final  variable
       Demo()
      {
        MAX_VALUE=100; //It must be initialized in constructor
      }
       void myMethod(){
       System.out.println(MAX_VALUE);
  }
  public static void main(String args[])
 {
    Demo obj=new  Demo();
    obj.myMethod();
  }
}
```

Output:100

- Uninitialized static final variable
  - A static final variable that is not initialized during declaration can only be initialized in static block.

```
class Example
{
  //static blank final variable
  static final int ROLL_NO;
  static
  {
    ROLL_NO=1230;
  }
  public static void main(String args[])
  {
    System.out.println(Example.ROLL
    _NO);
  }
}
```

Output:1230

using final with Inheritance

final method

- A final method cannot be overridden.
- This means even though a subclass can call the final method of the parent class without any issues but it cannot override it.

Example

```
class XYZ
{
      final void demo()
      {
            System.out.println("XYZ Class
      Method");
       }
}
 class ABC extends XYZ
{
      public static void main(String args[])
      {
            ABC obj= new ABC();
            obj.demo();
       }
}
```

Output:
XYZ Class Method

```
class Bike{
  final void run(){System.out.println("running");}
}

class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}

   public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
   }
}
```

Output:Compile Time Error

final class

- We cannot extend a final class. Consider the below example:

```
final class XYZ
{
}
 class ABC extends XYZ
{
  void demo()
 {
    System.out.println("My
     Method");
 }
public static void main(String args[])
{
    ABC obj= new ABC();
    obj.demo();
  }
}
```

Output Error:

The type ABC cannot subclass the final class XYZ

```
final class Bike{}

class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

```
Output:Compile Time Error
```

final method can be inherited but cannot override it

```
class Bike{
  final void run(){System.out.println("running...");}
}
class Honda2 extends Bike{
   public static void main(String args[]){
    new Honda2().run();
   }
}
```

```
Output:running...
```

Example

```
class XYZ
{
  final void demo()
 {
    System.out.println("XYZ Class Method");
  }
}
 class ABC extends XYZ
{
  void demo()
{
    System.out.println("ABC Class Method");
  }
      public static void main(String args[])
{
    ABC obj= new ABC();
    obj.demo();
  }
}
```

The program would throw a compilation error

- A constructor cannot be declared as final.
- Local final variables must be initialized during declaration.
- We cannot change the value of a final variable.
- A final method cannot be overridden.
- A final class will not be inherited.
- If method parameters are declared final then the value of these parameters cannot be changed.
- final, finally and finalize are three different terms. finally is used in exception handling and finalize is a method that is called by JVM during garbage collection.

Protected Members

- The private members of a class cannot be directly accessed outside the class.
- Only methods of that class can access the private members directly.
- sometimes it may be necessary for a subclass to access a private member of a superclass.
- This can be done by declaring the member as **protected**.

Modifier	Class	Subclass	World
public	Y	Y	Y
protected	Y	Y	N
private	Y	N	N

Access Control and Inheritance

- The following rules for inherited methods are enforced –
  - Methods declared public in a superclass also must be public in all subclasses.
  - Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.
  - Methods declared private are not inherited at all, so there is no rule for them.

Calling Order of Constructors

Constructor of base class with no argument gets automatically called in derived class constructor.

```
class Base
{
    Base()
    {
    System.out.println("Base
    Class Constructor Called ");
    }
}
class Derived extends Base
{
  Derived()
  {
   System.out.println("Derived
     Class Constructor Called ");
  }
}
```

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

O/P

Base Class Constructor Called
Derived Class Constructor Called

```
class A {
 A() {
        System.out.println("Inside A's constructor.");
    }
}
// Create a subclass by extending class A.
class B extends A {
 B() {
        System.out.println("Inside B's constructor.");
    }
}
// Create another subclass by extending B.
class C extends B {
 C() {
        System.out.println("Inside C's constructor.");
    }
}
public class Main{
   public static void main(String args[])
    {
      C c = new C();
    }
}
```

Output

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

if we want to call parameterized constructor of base class, then we can call it using super().
base class constructor call must be the first line in derived class constructor.

```
class Base
{
    int x;
    Base(int _x)
    {
      x = _x;
    }
}
class Derived extends Base
{
    int y;
    Derived(int _x, int _y)
    {
        super(_x);
        y = _y;
    }
    void Display()
    {
      System.out.println("x = "+x+", y = "+y);
    }
}

public class Main
{
  public static void main(String[] args)
  {
    Derived d = new Derived(10, 20);
    d.Display();
  }
}
```

Output:
x = 10, y = 20

Method Overriding

- Declaring a method in a subclass which is already present in the parent class is known as method overriding.
- Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class.
- method in parent class is called overridden method and the method in child class is called overriding method.
- class can give its own specific implementation to a inherited method without even modifying the parent class code
- The argument list of overriding methods (method of child class) must match the Overridden method(the method of parent class).
    - The data types of the arguments and their sequence should exactly match.
- The Access Modifier of the overriding method (method of subclass) cannot be more restrictive than the overridden method of parent class.
- Method overriding is used for runtime polymorphism
- A static method cannot be overridden.
    - It is because the static method is bound with class whereas the instance method is bound with an object.
    - Static belongs to the class area, and an instance belongs to the heap area.
- we cannot override java main method as it is a static method

Example - method overriding

```
//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}

  public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
}
```

Output

```
Bike is running safely
```

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

method overloading Vs. method overriding

No.	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

Dynamic Method Dispatch

- Dynamic Method Dispatch is the process in which a call to an overridden method is resolved at runtime using a superclass reference pointing to a subclass object.
- Achieves runtime polymorphism.
- The method call is resolved based on actual object type, not reference type.
- Works only with overridden methods.

```
class Animal {
    void sound() {
        System.out.println("Animal sound");
    }
}

class Cat extends Animal {
    void sound() {
        System.out.println("Cat meows");
    }
}

public class Demo {
    public static void main(String[] args) {
        Animal a = new Cat(); // Superclass reference, subclass object
        a.sound();            // Output: Cat meows (decided at runtime)
    }
}
```

Comparison Table

Feature	Method Overloading	Method Overriding	Dynamic Method Dispatch
Definition	Same method name, different parameters	Same method name and parameters in subclass	Runtime decision of overridden method call
Class Type	Same class	Different classes (inheritance)	Superclass reference to subclass object
Return Type	Can be different	Must be same (or covariant)	Based on overridden method
Polymorphism Type	Compile-time polymorphism (static)	Runtime polymorphism (dynamic)	Runtime polymorphism
Binding Time	Compile-time	Runtime	Runtime
Inheritance Required	No	Yes	Yes

Short Questions

1. What is polymorphism in Java? Explain with a simple example.
2. Define method overloading. What are the key rules for achieving method overloading?
3. Can you pass objects as parameters to methods? If yes, provide a small code snippet demonstrating this.
4. Is it possible for a method to return an object? Explain briefly.
5. What is recursion? Mention one advantage and one disadvantage of using recursion.
6. What is the purpose of the static keyword in Java? Give an example of a static variable.
7. Explain the concept of a final variable in Java. Can its value be changed after initialization?
8. What is an inner class? Briefly describe one scenario where an inner class might be useful.
9. Define the term "super class" in the context of inheritance.
10. What is a "subclass"? How is it related to a superclass?
11. What is the primary use of the super keyword in Java?
12. What is the access level of a protected member? Where can it be accessed?
13. Briefly explain the concept of method overriding.
14. What is dynamic method dispatch?
15. Can a final class be extended? Can a final method be overridden?

Essay Questions

1. Differentiate between method overloading and method overriding with suitable code examples for each.
2. Write a recursive Java method to calculate the factorial of a given number. Explain how the recursion works for factorial(4).
3. Discuss the characteristics of static variables and static methods. Provide a scenario where using static members would be appropriate.
4. Design a Java class Point with x and y coordinates. Write a method createMidPoint that takes two Point objects as parameters and returns a new Point object representing their midpoint.
5. Explain any two types of inheritance in Java with a simple class hierarchy diagram for each.
6. Describe two distinct uses of the super keyword in Java, providing a code snippet for each.
7. Explain the calling order of constructors in an inheritance hierarchy. Provide an example demonstrating this order.
8. Discuss the protected access modifier in detail. Provide an example demonstrating how protected members are accessed within the same package and in subclasses in different packages.
9. Explain the concept of dynamic method dispatch with a clear code example. How does it enable runtime polymorphism?
10. Explain how the final keyword is used with classes and methods in the context of inheritance. What are the implications of declaring a class or a method as final?
11. Elaborate on polymorphism in Java. Explain how method overloading and method overriding contribute to achieving polymorphism. Provide a comprehensive example demonstrating both

12. Compare and contrast static members (variables and methods) with instance members. Discuss their memory allocation, accessibility, and appropriate use cases with examples.
13. Describe different types of inner classes in Java (e.g., non-static, static nested, anonymous). Discuss their characteristics and provide a suitable use case for at least two types.
14. Design a class hierarchy for Shape, Circle, and Rectangle. Implement methods for calculating area and perimeter. Demonstrate how method overriding and the super keyword are used effectively within this hierarchy.
15. Explain the concept of abstract classes and abstract methods. How do they relate to polymorphism and inheritance? Provide an example of an abstract class and its concrete subclass.
16. Explain the various uses of the final keyword in Java: with variables (local, instance, static), methods, and classes. Provide illustrative code examples for each usage and discuss their implications.
17. You are tasked with designing a simplified Employee Management System. Create a Person class, an Employee class (inheriting from Person), and a Manager class (inheriting from Employee). Demonstrate:
    - Constructor chaining using super().
    - Method overriding for a displayDetails() method in each class.
    - Polymorphism by creating an array of Employee objects and calling displayDetails() on them.

Module 3

Packages and Interfaces –

Packages - Defining a Package, CLASSPATH, Access Protection, Importing Packages.

Interfaces - Interfaces v/s Abstract classes, defining an interface, implementing interfaces, accessing implementations through interface references, extending interface(s).

Exception Handling - Checked Exceptions, Unchecked Exceptions, try Block and catch Clause, Multiple catch Clauses, Nested try Statements, throw, throws and finally, Java Built-in Exceptions, Custom Exceptions.

Introduction to design patterns in Java : Singleton and Adaptor.

Java Packages

A package in Java is used to group related classes. Think of it as a folder in a file directory. We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:

- Built-in Packages (packages from the Java API)
- User-defined Packages (create your own packages)

Built-in Packages

The Java API is a library of prewritten classes that are free to use, included in the Java Development Environment.

The library contains components for managing input, database programming, and much much more. The complete list can be found at Oracle's website: The library is divided into packages and classes. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contains all the classes that belong to the specified package.

To use a class or a package from the library, you need to use the import keyword:

Import a Class

If you find a class you want to use, for example, the Scanner class, which is used to get user input, write the following code:

Example

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

In the example above, java.util is a package, while Scanner is a class of the java.util package.

To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation. In our example, we will use the nextLine() method, which is used to read a complete line:

Example

Using the Scanner class to get user input:

```
import java.util.Scanner;

class MyClass {

  public static void main(String[] args) {

    Scanner myObj = new Scanner(System.in);

    System.out.println("Enter username");

    String userName = myObj.nextLine();

    System.out.println("Username is: " + userName);

  }

}
```

Syntax

import *package.name.Class*; // Import a single class

import *package.name*.*; // Import the whole package

User-defined Packages

To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders on your computer:

Example

```
└── root
    └── mypack
        └── MyPackageClass.java
```

To create a package, use the package keyword:

MyPackageClass.java

```
package mypack;
class MyPackageClass {
  public static void main(String[] args) {
    System.out.println("This is my package!");
  }
}
```

Save the file as MyPackageClass.java, and compile it:

```
C:\Users\Your Name>javac MyPackageClass.java
```

Then compile the package:

```
C:\Users\Your Name>javac -d . MyPackageClass.java
```

This forces the compiler to create the "mypack" package.

The -d keyword specifies the destination for where to save the class file. You can use any directory name, like c:/user (windows), or, if you want to keep the package within the same directory, you can use the dot sign ".", like in the example above.

Note: The package name should be written in lower case to avoid conflict with class names.

When we compiled the package in the example above, a new folder was created, called "mypack".

To run the MyPackageClass.java file, write the following:

```
C:\Users\Your Name>java mypack.MyPackageClass
```

The output will be:

```
This is my package!
```

Detailed Explanation of CLASSPATH in Java

The CLASSPATH is a parameter—either set as an environment variable or passed as a command-line argument—that tells the Java compiler (javac) and Java Virtual Machine (java) where to look for .class files or Java packages used in your program.

 Why is CLASSPATH Important?

When you compile or run Java programs that use external packages or classes, Java needs to know where to find those classes. That's where the CLASSPATH comes in.

Access Protection in Packages

Java provides four levels of access control:

Modifier	Same Class	Same Package	Subclass	Other Package
private	✓	✗	✗	✗
(default)	✓	✓	✗	✗
protected	✓	✓	✓	✗
public	✓	✓	✓	✓

1. Same Class

- All modifiers work inside the same class.
- So everything is here.
- Example: A method can always access private or public variables if it's in the same class.

2. Same Package

- public, protected, and default (no modifier) can be accessed by other classes in the same package.
- private is not accessible outside its own class.

3. Subclass (Other Package)

- This means: A class that extends your class but is in a different package.
- public: Yes — visible everywhere.
- protected: Yes — visible to subclasses, even in different packages.
- default: No — not visible in a different package.
- private: No — not visible anywhere outside the class.

---

4. Other Packages (not subclass)

- public: Yes — accessible from anywhere.
- protected, default, and private: No — not accessible from other packages unless you are a subclass (for protected).

How to Set CLASSPATH in Ubuntu (Linux)

What is CLASSPATH?

- It tells Java where to find your .class files or .jar files.
- Useful when you're working with user-defined packages or external libraries.

Example: Suppose You Have This Project

project/

```
├── mypackage/
│   └── MyClass.java
└── Test.java
```

mypackage/MyClass.java

```
package mypackage;

public class MyClass {
    public void display() {
        System.out.println("Hello from MyClass!");
    }
}
```

Test.java

```
import mypackage.MyClass;

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

Step-by-Step: Set CLASSPATH and Compile

1. Open Terminal and go to your project directory:
   cd /path/to/project

2. Compile the package class first:
   javac mypackage/MyClass.java

3. Compile the main class with the classpath:
   javac -cp . Test.java

4. Run the program with the classpath:
   java -cp . Test

   -cp . means: "Look for classes in the current directory."

Difference Between Abstract Class and Interface in Java

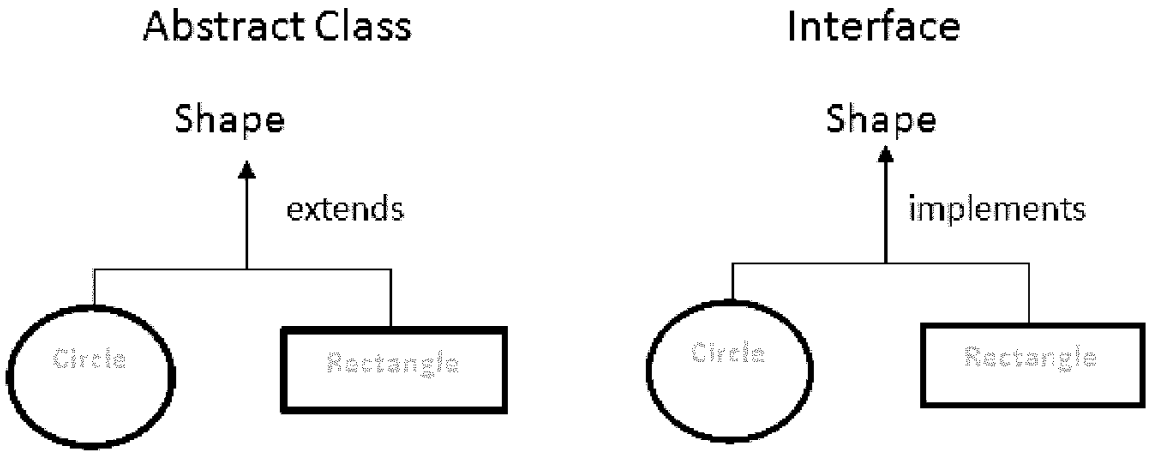
In object-oriented programming (OOP), both abstract classes and interfaces serve as fundamental constructs for defining contracts. They establish a blueprint for other classes, ensuring consistent implementation of methods and behaviors. However, they each come with distinct characteristics and use cases.

Difference Between Abstract Class and Interface

Points	Abstract Class	Interface
Definition	Cannot be instantiated; contains both abstract (without implementation) and concrete methods (with implementation)	Specifies a set of methods a class must implement; methods are abstract by default.
Implementation Method	Can have both implemented and abstract methods.	Methods are abstract by default; Java 8, can have default and static methods.
Inheritance	class can inherit from only one abstract class.	A class can implement multiple interfaces.

Access Modifiers	Methods and properties can have any access modifier (public, protected, private).	Methods and properties are implicitly public.
Variables	Can have member variables (final, non-final, static, non-static).	Variables are implicitly public, static, and final (constants).

As we know that abstraction refers to hiding the internal implementation of the feature and only showing the functionality to the users. i.e., showing only the required features, and hiding how those features are implemented behind the scene. Whereas, an Interface is another way to achieve abstraction in Java. Both abstract class and interface are used for abstraction



Interfaces

An interface is a completely "abstract class" that is used to group related methods with empty bodies:

Example

```
// interface
interface Animal {
  public void animalSound(); // interface method (does not have a body)
  public void run(); // interface method (does not have a body)
}
```

To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class with the implements keyword (instead of extends). The body of the interface method is provided by the "implement" class:

Example

```
// Interface
interface Animal {
  public void animalSound(); // interface method (does not have a body)
  public void sleep(); // interface method (does not have a body)
}

// Pig "implements" the Animal interface
class Pig implements Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
  public void sleep() {
```

```
    // The body of sleep() is provided here
    System.out.println("Zzz");
  }
}

class Main {
  public static void main(String[] args) {
    Pig myPig = new Pig();  // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Design Patterns in Java

Design patterns are proven solutions to common problems in software design. They help create more maintainable, scalable, and efficient code by offering reusable approaches to solving architectural problems.

1. Singleton Pattern

Purpose:

Ensures a class has only one instance and provides a global point of access to it.

Use Cases:

- Configuration settings
- Logger classes

- Database connections

Implementation:

```
public class Singleton {
    // Step 1: Create a private static instance
    private static Singleton instance;

    // Step 2: Make the constructor private to prevent instantiation
    private Singleton() {
        System.out.println("Singleton instance created.");
    }

    // Step 3: Provide a public static method to get the instance
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton(); // Lazy initialization
        }
        return instance;
    }
}
```

Usage:

```
public class Main {

    public static void main(String[] args) {

        Singleton s1 = Singleton.getInstance();

        Singleton s2 = Singleton.getInstance();



        System.out.println(s1 == s2); // true - both refer to the same instance

    }

}
```

---

2. Adapter Pattern

Purpose:

Allows incompatible interfaces to work together. It converts one interface into another expected by the client.

Use Cases:

- When integrating legacy code with new systems
- When working with third-party libraries

Example Scenario:

You have a class that uses a method request(), but a new system uses specificRequest(). You use an adapter to bridge the gap.

Class Diagram:

Client --> Target (expected interface)

^

|

Adapter --> Adaptee (incompatible interface)

Implementation:

```
// Target interface
interface Target {
    void request();
}

// Adaptee class with incompatible method
class Adaptee {
    public void specificRequest() {
        System.out.println("Specific request in Adaptee");
    }
}

// Adapter class that makes Adaptee compatible with Target
class Adapter implements Target {
    private Adaptee adaptee;

    public Adapter(Adaptee adaptee) {
```

```
        this.adaptee = adaptee;
    }

    @Override
    public void request() {
        adaptee.specificRequest(); // Delegating the call
    }
}
```

Usage:

```
public class Main {
    public static void main(String[] args) {
        Adaptee adaptee = new Adaptee();
        Target adapter = new Adapter(adaptee);
        adapter.request(); // Output: Specific request in Adaptee
    }
}
```

Pattern	Purpose	Key Feature

Singleton	Ensure a single instance of a class	Controlled instantiation
Adapter	Connect incompatible interfaces	Converts interface of a class

Exception Handling in Java

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

What is Exception in Java?

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

Exception Hierarchy

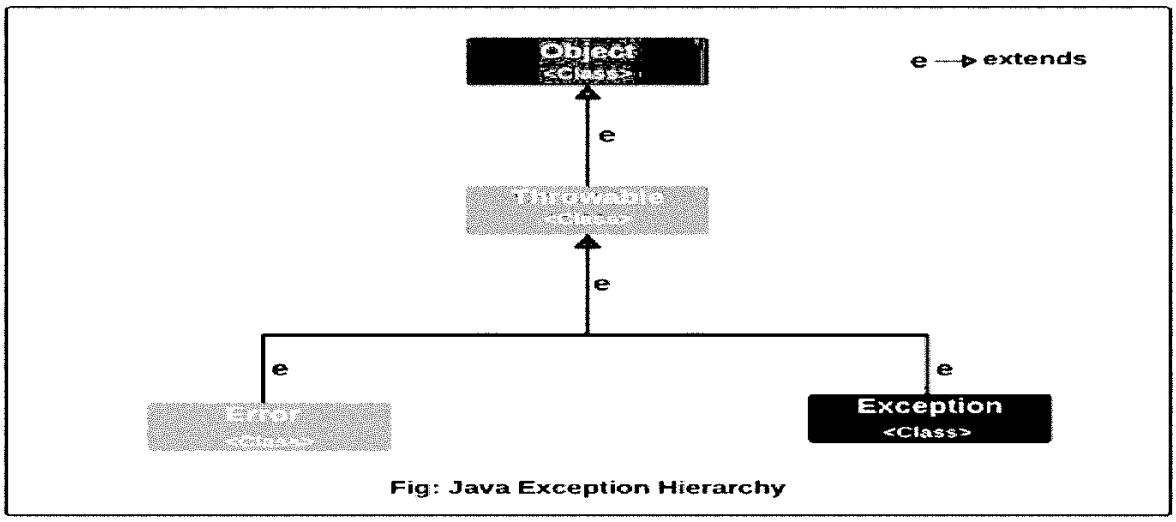


Fig: Java Exception Hierarchy

The Throwable class is the root of exception hierarchy and is an immediate subclass of Object class.Exception Hierarchy

1.Throwable class:

As shown in the above figure, Throwable class which is derived from Object class, is a top of exception hierarchy from which all exception classes are derived directly or indirectly. It is the root of all exception classes. It is present in the java.lang package.

Throwable class is the superclass of all exceptions in java. This class has two subclasses: Error and Exception. Errors or exceptions occurring in java programs are objects of these classes. Using Throwable class, you can also create your own custom exceptions

2. Error:

Error class is the subclass of Throwable class and a superclass of all the runtime error classes. It terminates the program if there is problem-related to a system or resources (JVM).

An error generally represents an unusual problem or situation from which it is difficult to recover. It does not occur by programmer mistakes. It generally occurs if the system is not working properly or resources are not allocated properly.

VirtualMachineError, StackOverFlowError, AssertionError, LinkageError, OutOfMmeoryError, etc are examples of error.

3. Exception:

It is represented by an Exception class that represents errors caused by the program and by external factors. Exception class is a subclass of Throwable class and a superclass of all the exception classes.

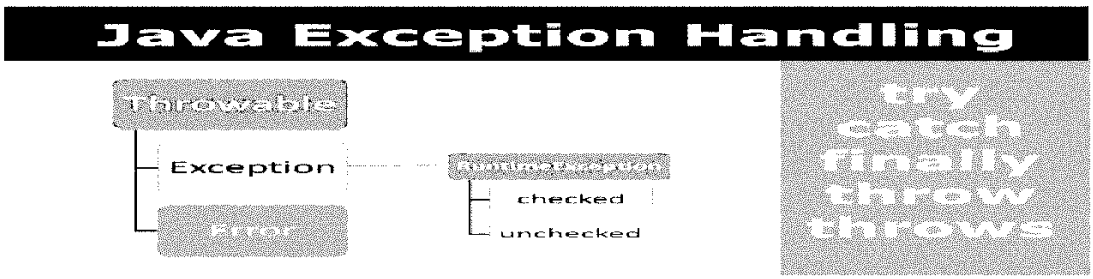
All the exception classes are derived directly or indirectly from the Exception class. They originate from within the application.

The exception class provides two constructors:

public Exception() (Default constructor)

public Exception(String message) (It takes a message string as argument)

Each of the exception classes provides two constructors: one with no argument and another with a String type argument. Exception class does not provide its own method. It inherits all methods provided by the Throwable class.



Unchecked exception

- Unchecked exception classes are defined inside

  java.lang package.

  - The unchecked exceptions are subclasses of the standard type RuntimeException.
  - In the Java language, these are called *unchecked exceptions because the compiler does not check to see whether there is a method that handles or* throws these exceptions.

  - If the program has an unchecked exception then it will *compile without error* but an exception occurs when the program runs.
  - .g Exceptions under Error , ArrayIndexOutOfBoundException

- E.g Exceptions under Error , ArrayIndexOutOfBoundException

Exception	Meaning
ArithmeticException	Arithmetic 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.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Checked exception

- There are some exceptions that are defined by java.lang that must be included in a

method's throws list, if a method generates such exceptions and that *method does not handle it itself*. These are called checked exceptions

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.

- IOException
- FileNotFoundException
- SQLException

Difference Between Checked Exception and Unchecked Exception

Checked Exception / Compile Time Exception	Unchecked Exception / Runtime Exception
1. Checked Exceptions are the exceptions that are checked and handled at compile time.	1. Unchecked Exceptions are the exceptions that are not checked at compiled time.
2. The program gives a compilation error if a method throws a checked exception.	2. The program compiles fine because the compiler is not able to check the exception.
3. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword.	3. A method is not forced by compiler to declare the unchecked exceptions thrown by its implementation. Generally, such methods almost always do not declare them, as well.
4. A checked exceptions occur when the chances of failure are too high.	4. Unchecked exception occurs mostly due to programming mistakes.
5. They are direct subclass of Exception class but do not inherit from RuntimeException.	5. They are direct subclass of RuntimeException class.

try Block and *catch Clause*

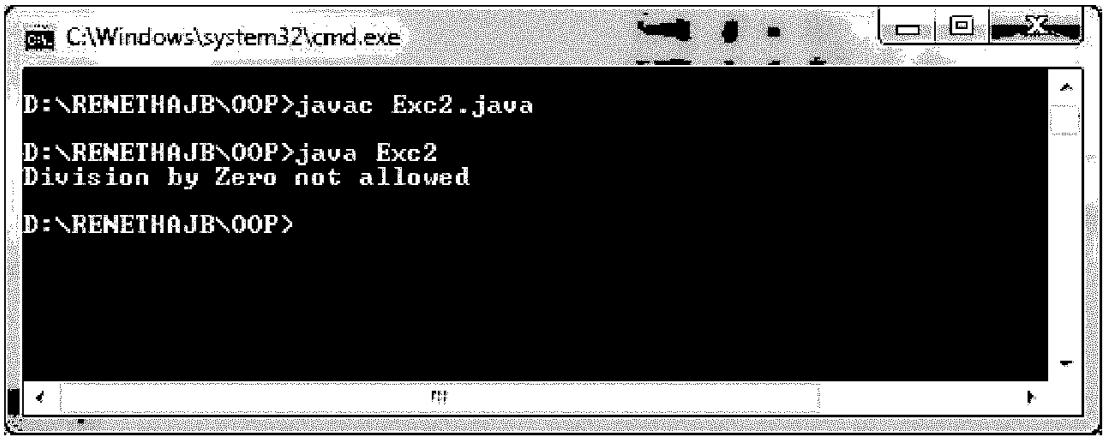
Benefits of exception handling

- First, it allows us to fix the error.
- Second, it prevents the program from terminating.
- To guard against and handle a run-time error, simply enclose the code that we want to monitor inside a try block.

- Immediately *after the try block*, there is a catch clause that specifies the exception type that we wish to catch . The catch block can process that exception..

Example

```
class Exc2{
                public static void main(String args[])
                    {
                        t ry
{
int d = 0;
int a = 42 / d;
}
catch(ArithmeticException ae)
{
System.out.println("Division by Zero not allowed");
}
}
}
```



try-catch block to handle division by zero exception

```
import java.io.*;
class GFG {
   public static void main(String[] args)
   {
      int a = 5;
      int b = 0;
      try {
         System.out.println(a / b); // throw Exception
```

```
        }
        catch (ArithmeticException e) {
            // Exception handler
            System.out.println(
                "Divided by zero operation cannot possible");
        }
    }
}
```

```
mayur@mayur: $ java GFG
Divided by zero operation cannot possible
mayur@mayur: $
```

- A try and its catch statement form a unit.
- The scope of the catch clause is restricted to those statements specified by the immediately preceding try statement.
  - Each catch block can catch exceptions in statements inside immediately preceding the try block.
- A catch statement cannot catch an exception thrown by another try statement (except in the case of nested try statements).
- The statements that are protected by try must be surrounded by curly braces. (That is, they must be within a block.)
- We cannot use try on a single statement

Multiple catch Clauses

- There can be more than one exception in a single piece of code.
  - To handle this type of situation, we 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 other catch statements are bypassed(ignored), and execution continues after the try/catch block.

- Why do we need multiple catch blocks in Java?

- Multiple catch blocks in Java are used to handle different types of exceptions. When statements in a single try block generate multiple exceptions, we require multiple catch blocks to handle different types of exceptions. This mechanism is called a multi-catch block in java.

- A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. You can catch different exceptions in different catch blocks if it contains a different exception handler. When an exception occurs in a try block, the corresponding catch block that handles that particular exception executes. So, if you have to perform different tasks at the occurrence of different exceptions, you can use the multi-try catch in Java.

  Syntax

```
try {
  // code
} catch (ExceptionType1 e1) {
  // Catch block
} catch (ExceptionType2 e2) {
  // Catch block
} catch (ExceptionType3 e3) {
  // Catch block
}
```

  Example

```
public class MultipleCatchBlock4 {
```

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


        try{
            String s=null;
            System.out.println(s.length());
           }
           catch(ArithmeticException e)
             {
              System.out.println("Arithmetic Exception occurs");
             }
           catch(ArrayIndexOutOfBoundsException e)
             {
              System.out.println("ArrayIndexOutOfBounds Exception occurs");
             }
           catch(Exception e)
             {
              System.out.println("Parent Exception occurs");
             }
           System.out.println("rest of the code");
    }
}
```

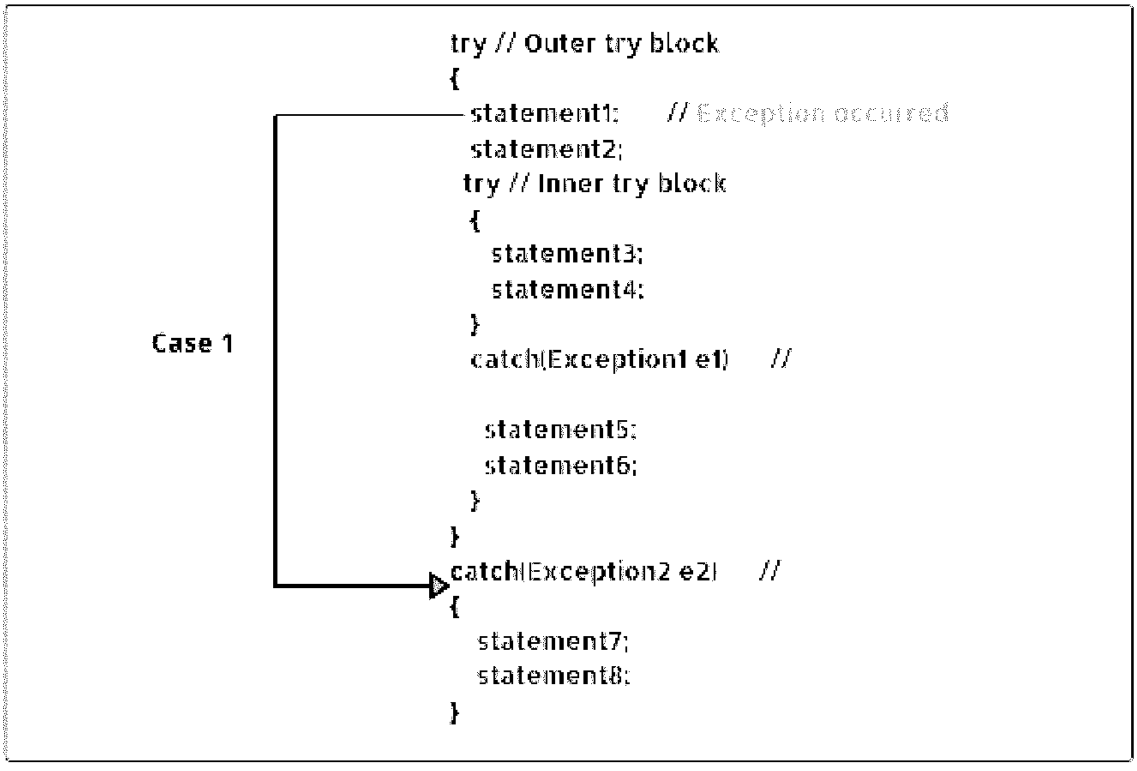
Output

Parent Exception occurs

rest of the code

Nested *try* Statements

- The try statement can be nested.
  - A try statement can be inside the block of another try.
- Each time a try statement is entered, the context of that exception is pushed on the stack.
  - 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 all of the nested try statements are exhausted.
  - If no catch statement matches, then the Java run-time system will handle the exception.



Example

class Geeks {

```
    // Main method
    public static void main(String args[]) {
        // Main try block
        try {
            // Initializing array
            int a[] = { 1, 2, 3, 4, 5 };

            // Trying to print element at index 5
            System.out.println(a[5]);

            // Inner try block (try-block2)
            try {
                // Performing division by zero
                int x = a[2] / 0; // This will throw ArithmeticException
            } catch (ArithmeticException e2) {
                System.out.println("Division by zero is not possible");
            }
        } catch (ArrayIndexOutOfBoundsException e1) {
            System.out.println("ArrayIndexOutOfBoundsException");
            System.out.println("Element at such index does not exist");
        }
    }
}
```

Output

```
ArrayIndexOutOfBoundsException
Element at such index does not exist
```

More features of Java :

Exception Handling:

- *throw*
- *throws*
- *Finally*

1.throw

We know that if an error occurs, an exception object is getting created and then Java runtime starts processing to handle them. Sometimes we might want to generate exceptions explicitly in our code. For example, in a user authentication program, we should throw exceptions to clients if the password is null. The throw keyword is used to throw exceptions to the runtime to handle it.

2.throws

When we are throwing an exception in a method and not handling it, then we have to use the throws keyword in the method signature to let the caller know the exceptions that might be thrown by the method. The caller method might handle these exceptions or propagate them to its caller method using the throws keyword. We can provide multiple exceptions in the throws clause, and it can be used with the main() method also

Throw vs Throws in java

1. Throws clause is used to declare an exception, which means it works similar to the try-catch block. On the other hand, throw keyword is used to throw an exception explicitly.
2. If we see syntax wise then throw is followed by an instance of Exception class and throws is followed by exception class names.

For example:
throw new ArithmeticException("Arithmetic Exception");
and
throws ArithmeticException;
3. Throw keyword is used in the method body to throw an exception, while throws is used in method signature to declare the exceptions that can occur in the statements present in the method.
For example:
Throw:

```
...
void myMethod() {
  try {
    //throwing arithmetic exception using throw
    throw new ArithmeticException("Something went wrong!!");
  }
  catch (Exception exp) {
    System.out.println("Error: "+exp.getMessage());
  }
}
...
```

Throws:

```
...
```

```
//Declaring arithmetic exception using throws
void sample() throws ArithmeticException{
  //Statements
}
```
…

4. You can throw one exception at a time but you can handle multiple exceptions by declaring them using throws keyword.

For example:

Throw:

```
void myMethod() {
  //Throwing single exception using throw
  throw new ArithmeticException("An integer should not be divided by zero!!");
}
```
..

Throws:

```
//Declaring multiple exceptions using throws
void myMethod() throws ArithmeticException, NullPointerException{
  //Statements where exception might occur
}
```

finally – the finally block is optional and can be used only with a try-catch block. Since exceptions halt the process of execution, we might have some resources open that will not get closed, so we can use the finally block. The finally block always gets executed, whether an exception occurred or not.

Finally Block in Java | A "finally" is a keyword used to create a block of code that follows a try or catch block.

A finally block contains all the crucial codes such as closing connections, stream, etc that is always executed whether an exception occurs within a try block or not.

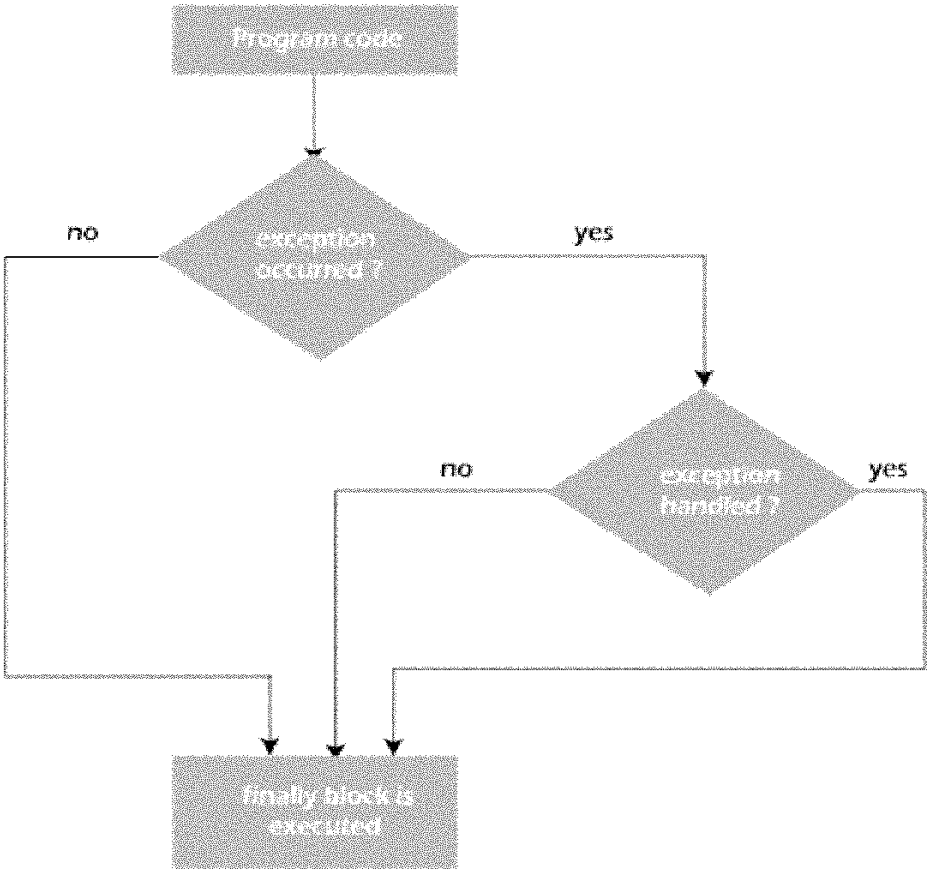
When the finally block is attached with a try-catch block, it is always executed whether the catch block has handled the exception thrown by the try block or not.

```
try
{
 statement1;
 statement2;
}
catch(Exceptiontype e1)
{
  statement3;
 }
 statement4;
finally
{
 statement5;
```

}

Why finally is needed?

- When exceptions are thrown, execution in a method takes a nonlinear path and changes the normal flow through the method.
  - Sometimes exceptions cause the method to return prematurely.
  - This may cause problems in some cases.
  - E.g a method opens a file upon entry and closes it upon exit, then we will not want the code that closes the file to be bypassed by the exception-handling mechanism.
    - In such situations the code for closing that file and other codes that should not be bypassed should be written inside finally block
    - This will ensure that necessary codes are not skipped because of exception handling.



Points To Remember While Using Finally Block in Java

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses whenever a try/catch block is present.

- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.
- finally block is not executed in case exit() method is called before finally block or a fatal error occurs in program execution.
- finally block is executed even method returns a value before finally block.

Why Java Finally Block Used?

- Java finally block can be used for clean-up (closing) the connections, files opened, streams, etc. those must be closed before exiting the program.
- It can also be used to print some final information.

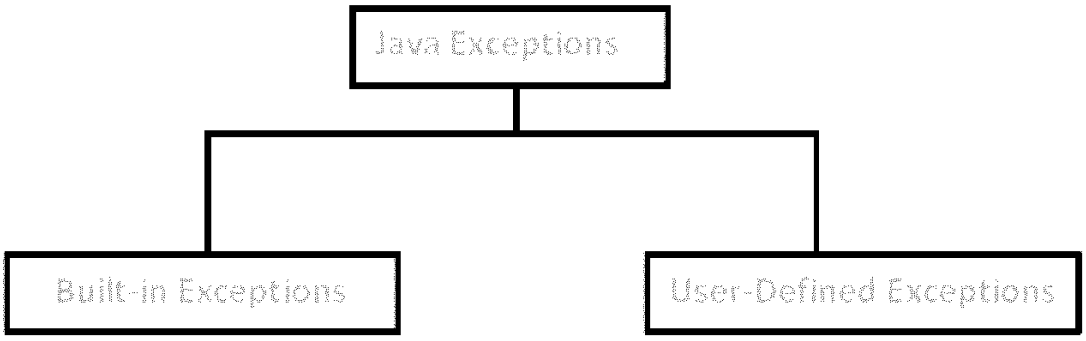
Example 1

```
public class FinallyExample {
   public static void main(String[] args) {
      try {
         int result = 10 / 0; // This will cause ArithmeticException
         System.out.println("Result: " + result);
      } catch (ArithmeticException e) {
         System.out.println("Error: Division by zero");
      } finally {
         System.out.println("Finally block executed");
      }
   }
}
```

OUTPUT

```
Error: Division by zero
Finally block executed
```

Types of Exception in Java with Examples



Built-in Exceptions in Java

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java.

1. **ArithmeticException:** It is thrown when an exceptional condition has occurred in an arithmetic operation.
2. **ArrayIndexOutOfBoundsException:** It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
3. **ClassNotFoundException:** This Exception is raised when we try to access a class whose definition is not found
4. **FileNotFoundException:** This Exception is raised when a file is not accessible or does not open.
5. **IOException:** It is thrown when an input-output operation failed or interrupted
6. InterruptedException: It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.
7. **NoSuchFieldException:** It is thrown when a class does not contain the field (or variable) specified
8. **NoSuchMethodException:** It is thrown when accessing a method that is not found.
9. **NullPointerException:** This exception is raised when referring to the members of a null object. Null represents nothing

10. **NumberFormatException:** This exception is raised when a method could not convert a string into a numeric format.
11. **RuntimeException:** This represents an exception that occurs during runtime.
12. **StringIndexOutOfBoundsException:** It is thrown by String class methods to indicate that an index is either negative or greater than the size of the string
13. **IllegalArgumentException :** This exception will throw the error or error statement when the method receives an argument which is not accurately fit to the given relation or condition. It comes under the unchecked exception.
14. **IllegalStateException :** This exception will throw an error or error message when the method is not accessed for the particular operation in the application. It comes under the unchecked exception.

Examples of Built-in Exception:

1. **Arithmetic exception :** It is thrown when an exceptional condition has occurred in an arithmetic operation.

```
// Java program to demonstrate
// ArithmeticException
class ArithmeticException_Demo {
  public static void main(String[] args) {
    try {
      int a = 30, b = 0;
      int c = a / b; // cannot divide by zero
      System.out.println("Result = " + c);
    } catch (ArithmeticException e) {
      System.out.println("Can't divide a number by 0");
    }
  }
}
```

Output
Can't divide a number by 0

2. ArrayIndexOutOfBounds Exception: It is thrown to indicate that an array has been

accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.

```
// Java program to demonstrate
// ArrayIndexOutOfBoundException
class ArrayIndexOutOfBound_Demo {
  public static void main(String[] args) {
    try {
      int[] a = new int[5];
      a[5] = 9; // accessing 6th element in an array of
            // size 5
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Array Index is Out Of Bounds");
    }
  }
}
```

Output
Array Index is Out Of Bounds

3. ClassNotFoundException : This Exception is raised when we try to access a class whose definition is not found.

```
// Java program to illustrate the
// concept of ClassNotFoundException
class Bishal {

}

class Geeks {

}

class MyClass {
  public static void main(String[] args) {
    try {
      Object o = Class.forName(args[0]).newInstance();
      System.out.println("Class created for " + o.getClass().getName());
    } catch (ClassNotFoundException | InstantiationException | IllegalAccessException e) {
      System.out.println("Exception occurred: " + e.getMessage());
    }
```

```
  }
}
```

Output:

ClassNotFoundException

4. FileNotFoundException : This Exception is raised when a file is not accessible or does not open.

```
// Java program to demonstrate
// FileNotFoundException
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;

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

Output
File does not exist

5. IOException : It is thrown when an input-output operation failed or interrupted

```
// Java program to illustrate IOException
import java.io.*;

class Geeks {
  public static void main(String[] args) {
    try {
      FileInputStream f = new FileInputStream("abc.txt");
      int i;
      while ((i = f.read()) != -1) {
        System.out.print((char)i);
```

```
        }
        f.close();
      } catch (IOException e) {
        System.out.println("IOException occurred: " + e.getMessage());
      }
   }
}
```

Output:

error: unreported exception IOException; must be caught or declared to be thrown

How to Create User Defined Exceptions in Java?

To create a custom exception in java, we have to create a class by extending it with the Exception class from the java.lang package.

```
//user-defined exception in java
public class SimpleCustomException extends Exception{

}
```

Example

```
// class representing custom exception
class InvalidAgeException  extends Exception
{
   public InvalidAgeException (String str)
   {
      // calling the constructor of parent Exception
      super(str);
   }
}

// class that uses custom exception InvalidAgeException
public class TestCustomException1
{

   // method to check the age
   static void validate (int age) throws InvalidAgeException{
     if(age < 18){

      // throw an object of user defined exception
      throw new InvalidAgeException("age is not valid to vote");
   }
     else {
      System.out.println("welcome to vote");
```

```
    }
        } // main method
    public static void main(String args[])
    {
        try
        {
            // calling the method
            validate(13);
        }
        catch (InvalidAgeException ex)
        {
            System.out.println("Caught the exception");

            // printing the message from InvalidAgeException object
            System.out.println("Exception occured: " + ex);
        }

        System.out.println("rest of the code...");
    }
}
```

```
C:\Users\Anurati\Desktop\abcDemo>javac TestCustomException1.java

C:\Users\Anurati\Desktop\abcDemo>java TestCustomException1
Caught the exception
Exception occured: InvalidAgeException: age is not valid to vote
rest of the code...
```

Short answer Questions

1. Define a Java package. How do you import a user-defined package?
2. What is CLASSPATH? How is it related to packages?
3. List any three access modifiers and explain their use in packages.
4. Write any three differences between interfaces and abstract classes.
5. Write the syntax for defining and implementing an interface.
6. What is a custom exception? When and why is it used?
7. Differentiate between throw and throws in Java with examples.
8. Explain the use of finally block in exception handling.
9. What is the Singleton design pattern? Mention its advantages.
10. When is the Adapter pattern used? Explain with a real-world analogy.

11. Give an example of nested try-catch in Java.
12. What are built-in exceptions in Java? Mention any three.
13. How can you access interface methods using interface references?
14. Write a short note on extending interfaces in Java.
15. Give an example to demonstrate multiple catch blocks.

Essay questions

1. Explain the steps to create and use a **user-defined package** in Java. How is CLASSPATH used in this context?
2. Define and compare **interfaces** and **abstract classes**. Write a Java program to implement and access interface methods.
3. Explain **exception handling** in Java. Describe try, catch, throw, throws, and finally with examples.
4. Write a Java program that demonstrates **custom exception handling**. Explain how custom exceptions are declared and used.
5. Discuss the **Singleton Design Pattern** in detail. Write a Java program to implement it and explain how it restricts object creation.
6. What is the **Adapter Design Pattern**? How is it implemented in Java? Write code to demonstrate class-based or object-based adapter.
7. Describe how **multiple catch blocks** and **nested try statements** work in Java. Give a complete example.
8. Explain the difference between **checked** and **unchecked exceptions** with appropriate code snippets.
9. Write a Java program to create a package named mathops containing arithmetic operations. Import and use it in another class.
10. Write a program that handles **file reading** using throws clause. Explain how exception propagation works.

MODULE 4

SOLID Principles in Java (https://www.javatpoint.com/solid-principles-java)
Swings fundamentals – Overview of AWT, Swing v/s AWT, Swing Key Features, Model View Controller (MVC), Swing Controls, Components andContainers, Swing Packages, Event Handling in Swings, Swing Layout Managers, Exploring Swings– JFrame, JLabel, The Swing Buttons,JTextField.
Event handling – Event Handling Mechanisms, Delegation Event Model,Event Classes, Sources of Events, Event Listener Interfaces, Using the Delegation Event Model.
Developing Database Applications using JDBC – JDBC overview, Types,Steps, Common JDBC Components, Connection Establishment, SQL Fundamentals [For projects only] - Creating and Executing basic SQL Queries, Working with Result Set, Performing CRUD Operations with JDBC.

What is AWT?

AWT also known as Abstract window toolkit. It is a platform dependent API used for developing GUI (graphical user interface) or applications that are window based. It was developed by Sun Microsystems In 1995 and is heavy weighted. It is generated by the system's host operating system and contains a large number of methods and classes which are used for creating and managing the UI of an application. The main difference between AWT and Swing is that AWT is purely used for GUI whereas Swing is used for both GUI as well as for making web applications.

What is Swing?

In Java, a swing is a light-weighted, GUI (graphical user interface) that is used for creating different applications. It has platform-independent components and enables users to create buttons as well as scroll bars. It also includes a package for creating applications for desktops. The components of swing are written in Java and are a part of the Java foundation class.

Key Differences Between AWT and Swing

Package:

- AWT: Part of java.awt package.
- Swing: Part of javax.swing package.

Component Design:

- AWT: Uses native system components, hence platform-dependent look.
- Swing: Uses lightweight components, offering a consistent look across platforms.

Performance:

- AWT: Generally slower since it's tied to native GUI components.
- Swing: Faster, more responsive due to lightweight design.

Customizability:

- AWT: Limited customization for components' appearance.
- Swing: Highly customizable with features like 'Look and Feel'.

Difference Between AWT and Swing

AWT	Swing
In Java, awt is an API that is used for developing GUI (Graphical user interface) applications.	Swing on another hand, in Java, is used for creating various applications including the web.
Java AWT components are heavily weighted.	Java swing components are light-weighted.
Java AWT has fewer functionalities as compared to that of swing.	Swing has greater functionalities as compared to awt in Java.
The code execution time in Java AWT is more.	The execution time of code in Java swing is less compared to that of awt.
In Java AWT, the components are platform-dependent.	The swing components are platform-independent.
MVC (Model-View-Controller) is not supported by Java awt.	MVC (Model-View-Controller) is supported by Java swing.
In Java, awt provides less powerful components.	Swing provides more powerful components.
The awt components are provided by the package java.awt	The swing components are provided by javax.swing package
There are abundant features in awt that are developed by developers and act as a thin layer between development and operating system.	Swing has a higher level of in-built components for developers which facilitates them to write less code.

The awt in Java has slower performance as compared to swing.	Swing is faster than that of awt.
AWT Components are dependent on the operating system.	Swing components are completely scripted in Java. Its components are not dependent on the operating system.

Features of Java Swing

1. **Platform Independent**
2. **Customizable**
3. **Plugging**
4. **MVC**
5. **Manageable Look and Feel**
6. **Lightweight**

1. Platform Independent

It is platform-independent, as the swing components used to construct the program are not platform-specific. It works on any platform and in any location.

2. Customizable

Swing controls are simple to customize. It can be changed, and the swing component application's visual appearance is independent of its internal representation.

3. Plugging

Java Swing has a pluggable look and feel. This feature allows users to change the appearance and feel of Swing components without having to restart the application. The Swing library will enable components to have the same look and feel across all platforms, regardless of where the program is running. The Swing library provides an API that allows complete control over the appearance and feel of an application's graphical user interface.

4. MVC

The MVC Relationships:
A visual component is made up of three distinct aspects in general:

- The appearance of the component when it is rendered on the screen.
- The component responds to the user.

- The state information connection with the component.

One component architecture has proven to be exceptionally effective over time.
MVC stands for Model-View-Controller.

- The model agrees with the state information associated with the Component in MVC terminology.
- The view determines how the component appears on screen and any aspects of the view that are influenced by the model's current state.
- The controller is in charge of determining how the component responds to the user.

5. Manageable Look and Feel

It's simple to manage and configure. Its mechanism and composition pattern allows changes to be made to the settings while the program runs. Constant changes to the application code can be made without making any changes to the user interface.

6. Lightweight

Lightweight Components: The JDK's AWT has supported lightweight component development since version 1.1. A component must not rely on non-Java [O/s based] system classes to be considered light. The look and feel classes in Java help Swing components have their view.
You can also read about Why is Java Platform Independent here.

7. Rich Control

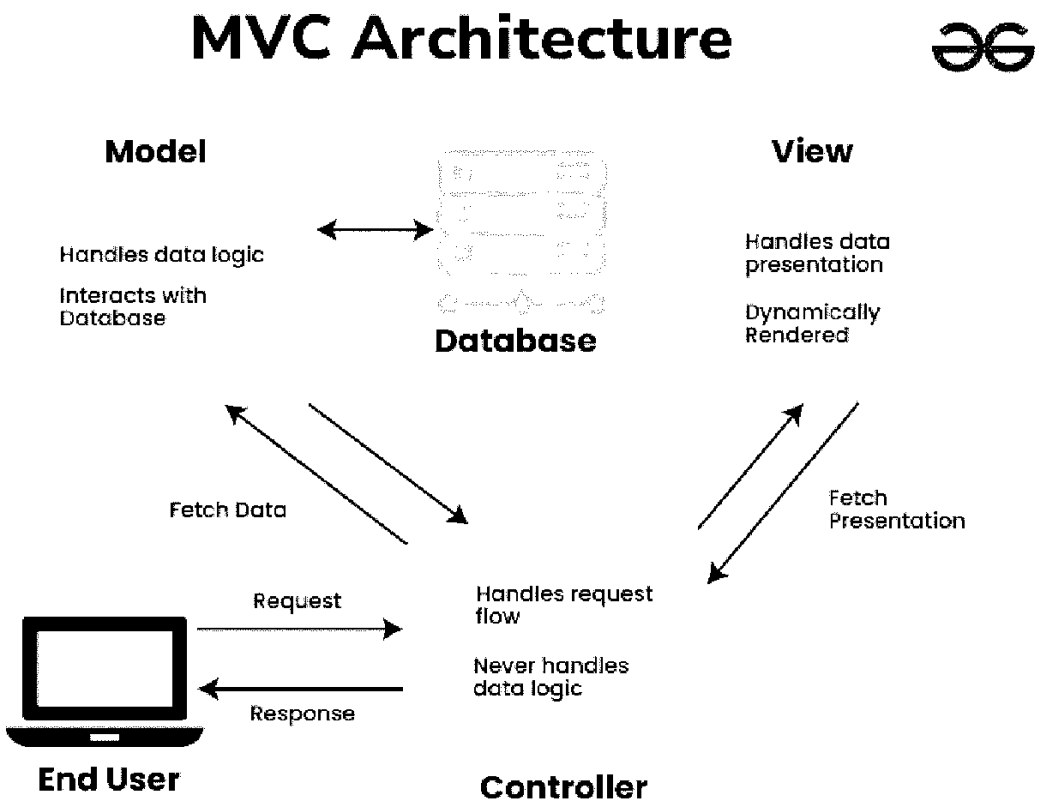
Swing offers a wide variety of rich, interactive controls such as buttons, text fields, tables, sliders, and trees. These components enable the creation of sophisticated user interfaces with advanced features like drag-and-drop support, tooltips, and event handling, enhancing user interaction.

8. Configurable

Swing allows extensive configuration of its components. Developers can set properties such as size, color, font, and event handling, and even define custom behaviours for components. This configurability ensures that the GUI can adapt to various application requirements and user preferences.

What is MVC Architecture?

MVC (Model-View-Controller) architecture is a universal pattern of a structure in which an application is divided into three parts which are all dedicated to certain parts of the whole application. This pattern is normally used in software development to create organized and easy-to-maintain code. Here's a deeper look at each component:



- **Model:** It is worth stating that the Model stands as the data layer for the application. It is directly involved in managing the data as well as the control of the application's logic and rules.
- **View:** The View is in the presentation tier. It plays a role of presenting the information given by the Model to the user and transferring the user commands to the Controller. The View is used to display the data to the user in a readable and manageable way using the interface created by the Controller.
- **Controller:** The Controller CE works in the middle between the Model and the View. It takes the input from the View, sometimes modifies it with the help of the Model, and sends it back to the View. the results back to the View.

The MVC architecture is significant in system design for the following reasons:

- **Separation of Concerns:** MVC structures an application into three integrated elements and that really separates concern. Due to the clear division of

responsibilities among each of the components, the functioning of the application becomes more logical and comprehensible.

- **Reusability:** Due to the fact that Model, View, and Controller are all distinct entities, components can be utilized in the various sections of the application or different projects. For example, a Model class that contains user data can be used many times in the views and the controllers.
- **Scalability:** MVC amplifies the creation of applications that can be developed further. When the application advances, new functionalities of the application can be added without significant alterations to some parts of the application because they are elusive.
- **Testability:** This separation of concerns makes it easier for the testing of each part from the other as we do from other problems. One of the testing strategies is to have separate tests for Model, View, and Controller parts, in this way, one is sure that each part is functioning properly before combining them.

Key Components of MVC Architecture

Here are detailed explanations and examples for each of the key components of the MVC architecture:Here are detailed explanations and examples for each of the key components of the MVC architecture

:**Model:** The Model component brings together all the core details and business knowledge. Therefore, it is concerned with the management of data, states and rules of the application. For example, in a booking system the Model might deal with user data, booking details and with calculations of prices.

Control	Class	Description
Label	JLabel	Displays a short string or an image icon.
Button	JButton	A push button that can trigger an action.
Text Field	JTextField	Allows the user to enter a single line of text.

Password Field	JPasswordField	A text field that hides input (for passwords).
Text Area	JTextArea	Multi-line text input area.
Check Box	JCheckBox	Represents an on/off toggle. Multiple boxes can be selected.
Radio Button	JRadioButton	Allows single selection among grouped options.
Combo Box	JComboBox	A drop-down list of items.
List	JList	Displays a list of items.
Panel	JPanel	A generic container for grouping components.
Frame	JFrame	A top-level window with a title and border.
Dialog	JDialog	A pop-up window for messages or input.
Scroll Pane	JScrollPane	Provides scrollable view for components.
Tree	JTree	Displlays Hierarchial data
Table	JTable	Displays data in a tabular format.

View: The View component is responsible for the presentation layer. It takes the data provided by the Model and presents it to the user in a readable format.

Controller: The Controller component handles user input and interactions. It processes the input, interacts with the Model, and updates the View accordingly.

Common Swing Controls (Components)

In Java, Swing is a part of Java Foundation Classes (JFC) used to create Graphical User Interfaces (GUIs). It provides a rich set of lightweight components that work the same across all platforms.

Basic Example Using Swing Controls

```
import javax.swing.*;

public class SwingExample {

    public static void main(String[] args) {

        // Create frame

        JFrame frame = new JFrame("Swing Example");

        frame.setSize(300, 200);

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Create components

        JLabel label = new JLabel("Enter Name:");

        JTextField textField = new JTextField(15);

        JButton button = new JButton("Submit");

        // Add components to panel
```

```
        JPanel panel = new JPanel();

        panel.add(label);

        panel.add(textField);

        panel.add(button);

        // Add panel to frame

        frame.add(panel);

        frame.setVisible(true);

    }

}
```

Components and Containers

In Java, Swing is a part of Java Foundation Classes (JFC) used to create Graphical User Interfaces (GUIs). It provides a rich set of lightweight components that work the same across all platforms.

Common Swing Controls (Components)

Control	Class	Description
Label	JLabel	Displays a short string or an image icon.

Button	JButton	A push button that can trigger an action.
Text Field	JTextField	Allows the user to enter a single line of text.
Password Field	JPasswordField	A text field that hides input (for passwords).
Text Area	JTextArea	Multi-line text input area.
Check Box	JCheckBox	Represents an on/off toggle. Multiple boxes can be selected.
Radio Button	JRadioButton	Allows single selection among grouped options.
Combo Box	JComboBox	A drop-down list of items.
List	JList	Displays a list of items.
Panel	JPanel	A generic container for grouping components.
Frame	JFrame	A top-level window with a title and border.
Dialog	JDialog	A pop-up window for messages or input.
Scroll Pane	JScrollPane	Provides scrollable view for components.
Table	JTable	Displays data in a tabular format.

Tree	JTree	Displays hierarchical data.

Basic Example Using Swing Controls

```
import javax.swing.*;

public class SwingExample {

    public static void main(String[] args) {

        // Create frame

        JFrame frame = new JFrame("Swing Example");

        frame.setSize(300, 200);

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);



        // Create components

        JLabel label = new JLabel("Enter Name:");

        JTextField textField = new JTextField(15);

        JButton button = new JButton("Submit");



        // Add components to panel

        JPanel panel = new JPanel();

        panel.add(label);

        panel.add(textField);

        panel.add(button);
```

```
        // Add panel to frame
        frame.add(panel);
        frame.setVisible(true);
    }
}
```

Notes

- All Swing classes are in the javax.swing package.
- Swing components are lightweight, unlike AWT which uses native code.
- Event handling in Swing is done via Event Listeners (e.g., ActionListener, ItemListener).
- Layout managers (FlowLayout, BorderLayout, GridLayout, etc.) help organize components in a GUI.
- In Java GUI programming (especially using AWT and Swing), Components and Containers are two fundamental building blocks:

---

1. Component

A Component is an object that represents a GUI element. It can be anything that can be seen and interacted with in a window — like buttons, labels, text fields, etc.

- All components inherit from java.awt.Component.
- Examples of components:

- Button
- Label
- TextField
- Checkbox
- List
- Canvas
- In Swing: JButton, JLabel, JTextField, etc.

Examples:

Button b = new Button("Click Me");

Label l = new Label("Hello, World!");

---

2. Container

A Container is a special type of component that can hold other components (including other containers).

- Containers help in grouping and organizing components.
- Every container is a subclass of java.awt.Container.
- Containers often use layout managers to control how components are arranged.

Types of Containers:

Container	Description
Panel	A simple container for holding components.
Frame	A top-level window with a title bar and border.
Dialog	A pop-up window for messages or input.
Applet	A container used in web-based Java programs.

In Swing: JPanel, JFrame, JDialog, etc.

Example:

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

Panel p = new Panel();

Button b = new Button("Click Me");

p.add(b);   // Adding component to container

f.add(p);   // Adding panel to frame

f.setSize(300, 200);

f.setVisible(true);
```

Component vs Container

Feature	Component	Container
Inherits from	java.awt.Component	java.awt.Container
Can hold other components	No	Yes
Examples	Button, Label, TextField	Panel, Frame, Applet

Swing Packages

In Java, Swing is a part of the Java Foundation Classes (JFC) used for building Graphical User Interfaces (GUIs). The Swing packages contain classes for components, containers, event handling, borders, pluggable look-and-feel, and more.**Main Swing Packages in Java**

Package	Description
javax.swing	Core package for all Swing components like JButton, JLabel, JFrame, etc.
javax.swing.event	Contains classes and interfaces for event handling specific to Swing (e.g., ListSelectionEvent, ChangeListener).
javax.swing.border	Provides classes for drawing borders around components (BevelBorder, LineBorder, etc.).

javax.swing.table	Contains classes for handling table components (JTable, TableModel, etc.).
javax.swing.text	Supports customizable text components (JTextComponent, StyledDocument, etc.).
javax.swing.tree	Used for tree components (JTree, TreeModel, etc.).
javax.swing.plaf	Supports the Pluggable Look-and-Feel architecture of Swing.
javax.swing.filechooser	Contains classes related to file choosing dialogs (JFileChooser).
javax.swing.colorchooser	Contains classes for choosing colors (JColorChooser).
javax.swing.text.html	Support for HTML content in text components.
javax.swing.text.html.parser	Parser classes for HTML document structures.

Example Using javax.swing

```
import javax.swing.*;

public class SwingDemo {

    public static void main(String[] args) {

        JFrame frame = new JFrame("Swing Package Example");
```

```
        JButton button = new JButton("Click Me");

        JLabel label = new JLabel("Hello, Swing!");

        JPanel panel = new JPanel();

        panel.add(label);

        panel.add(button);

        frame.add(panel);

        frame.setSize(300, 200);

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.setVisible(true);

    }

}
```

Notes

- Most common Swing components are in javax.swing.
- Swing is platform-independent and lightweight.
- Event handling is usually done using listeners from java.awt.event and javax.swing.event

Event Handling in Swings

In Java Swing, events are handled using an event-driven programming model. Here's how it works:

1. **Event Source:**
    - The component that generates an event, such as a button (JButton), text field (JTextField), or menu (JMenuItem).
2. **Event Listener:**
    - An object that listens for and responds to events. Event listeners are implemented using specific interfaces such as:
        - ActionListener: For handling button clicks and other actions.
        - KeyListener: For handling keyboard events.
        - MouseListener: For handling mouse events.
        - Others like FocusListener, WindowListener, etc.
3. **Event Registration:**
    - The listener is registered with the event source using methods like addActionListener(), addKeyListener(), etc.
4. **Event Object:**
    - When an event occurs, an event object (e.g., ActionEvent, MouseEvent) is created and passed to the listener's method.
5. **Handling Events:**
    - The listener's method executes the required action when the event is triggered.

Example

```
JButton button = new JButton("Click Me");

button.addActionListener(e -> System.out.println("Button clicked!"));
```

Layout Manager-Java Swing

The LayoutManagers are used to arrange components in a particular manner. The Java LayoutManagers facilitate 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 following classes that represent the layout managers:

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

Some of the mainly used layouts are as follows:

BorderLayout:

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

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

Constructors of BorderLayout class:

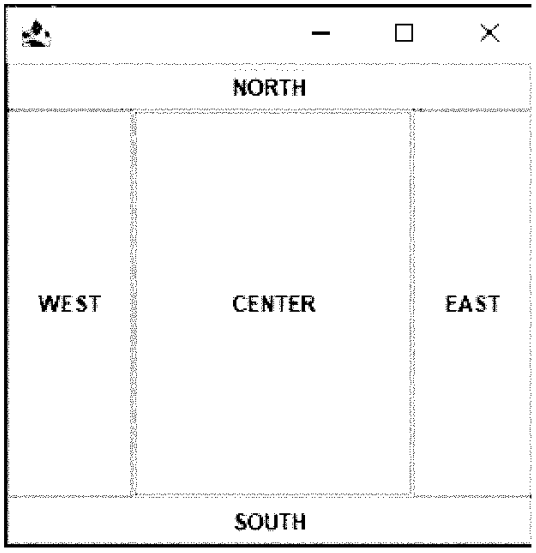
- BorderLayout(): creates a border layout but with no gaps between the components.
- BorderLayout(int hgap, int vgap): creates a border layout with the given horizontal and vertical gaps between the components.

Example:

```
import java.awt.*;

import javax.swing.*;

public class Border

{

   JFrame f;

   Border()

   {

      f = new JFrame();


      // creating buttons

      JButton b1 = new JButton("NORTH");;

      JButton b2 = new JButton("SOUTH");;

      JButton b3 = new JButton("EAST");;

      JButton b4 = new JButton("WEST");;

      JButton b5 = new JButton("CENTER");;
```

```
    f.add(b1, BorderLayout.NORTH);
    f.add(b2, BorderLayout.SOUTH);
    f.add(b3, BorderLayout.EAST);
    f.add(b4, BorderLayout.WEST);
    f.add(b5, BorderLayout.CENTER);

    f.setSize(300, 300);
    f.setVisible(true);
  }
  public static void main(String[] args) {
    new Border();
  }
  }
```

output



GridLayout:

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

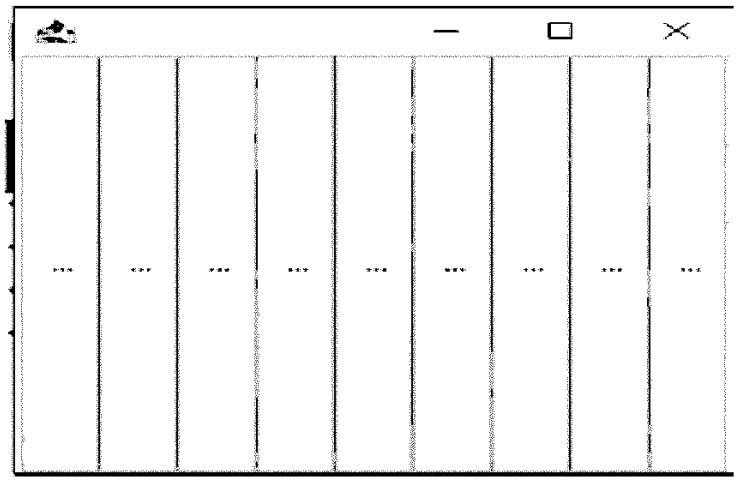
Constructors of GridLayout class

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.

Example:

```
import java.awt.*;

import javax.swing.*;

public class GridLayoutExample

{

   JFrame frameObj;

   GridLayoutExample()

   {

      frameObj = new JFrame();

      JButton btn1 = new JButton("1");

      JButton btn2 = new JButton("2");

      JButton btn3 = new JButton("3");

      JButton btn4 = new JButton("4");

      JButton btn5 = new JButton("5");

      JButton btn6 = new JButton("6");

      JButton btn7 = new JButton("7");
```

```
    JButton btn8 = new JButton("8");
    JButton btn9 = new JButton("9");
    frameObj.add(btn1); frameObj.add(btn2); frameObj.add(btn3);
    frameObj.add(btn4); frameObj.add(btn5); frameObj.add(btn6);
    frameObj.add(btn7); frameObj.add(btn8); frameObj.add(btn9);
    frameObj.setLayout(new GridLayout());
    frameObj.setSize(300, 300);
    frameObj.setVisible(true);
}
public static void main(String argvs[])
{
    new GridLayoutExample();
}}
```



FlowLayout:

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

Fields of FlowLayout class

1. public static final int LEFT
2. public static final int RIGHT
3. public static final int CENTER
4. public static final int LEADING
5. public static final int TRAILING

Constructors of FlowLayout class

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.

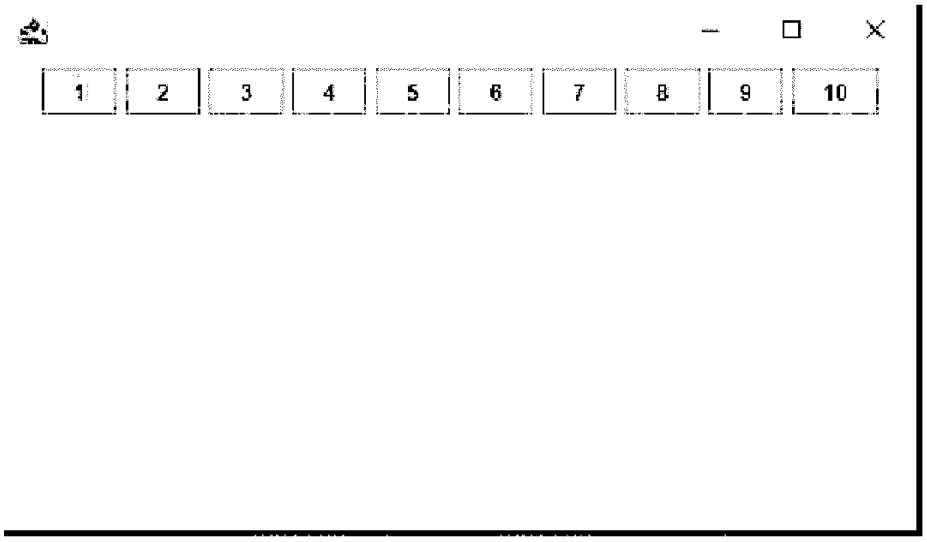
Example:

```
import java.awt.*;

import javax.swing.*;

public class FlowLayoutExample

{

   JFrame frameObj;

   FlowLayoutExample()

   {

      frameObj = new JFrame();
```

```
    JButton b1 = new JButton("1");
    JButton b2 = new JButton("2");
    JButton b3 = new JButton("3");
    JButton b4 = new JButton("4");
    JButton b5 = new JButton("5");
    JButton b6 = new JButton("6");
    JButton b7 = new JButton("7");
    JButton b8 = new JButton("8");
    JButton b9 = new JButton("9");
    JButton b10 = new JButton("10");
    frameObj.add(b1); frameObj.add(b2); frameObj.add(b3); frameObj.add(b4);
    frameObj.add(b5); frameObj.add(b6);  frameObj.add(b7);  frameObj.add(b8);
    frameObj.add(b9);  frameObj.add(b10);
    frameObj.setLayout(new FlowLayout());
    frameObj.setSize(300, 300);
    frameObj.setVisible(true);
}

public static void main(String argvs[])
{
    new FlowLayoutExample();
```

```
}

}
```

output



BoxLayout:

The Java BoxLayout class is used to arrange the components either vertically or horizontally. For this purpose, the BoxLayout class provides four constants. They are as follows:

Fields of BoxLayout Class

1. public static final int X_AXIS: Alignment of the components are horizontal from left to right.
2. public static final int Y_AXIS: Alignment of the components are vertical from top to bottom.
3. public static final int LINE_AXIS: Alignment of the components is similar to the way words are aligned in a line,
4. public static final int PAGE_AXIS: Alignment of the components is similar to the way text lines are put on a page,

Constructor of BoxLayout class

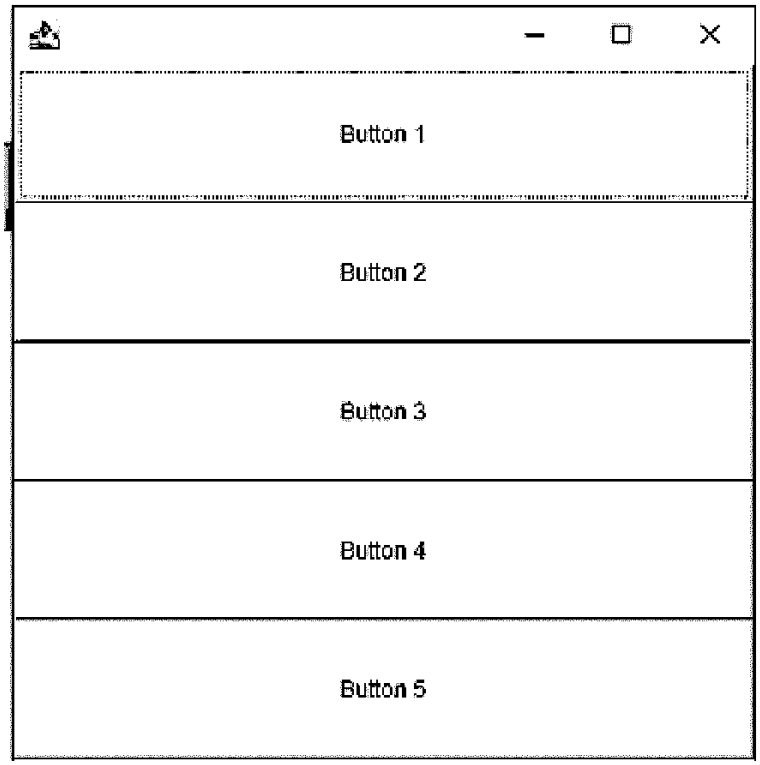
1. BoxLayout(Container c, int axis): creates a box layout that arranges the components with the given axis.

Example:

```
import java.awt.*;
import javax.swing.*;
class BoxLayoutExample1 extends Frame {
   Button buttons[];
   public BoxLayoutExample1 () {
      buttons = new Button [5];
      for (int i = 0;i<5;i++) {
         buttons[i] = new Button ("Button " + (i + 1));
         add (buttons[i]);
      }
      setLayout (new BoxLayout (this, BoxLayout.Y_AXIS));
      setSize(400,400);
      setVisible(true);
   }
   public static void main(String args[]){
      BoxLayoutExample1 b=new BoxLayoutExample1();
```

```
    }

}
```

Output:



CardLayout:

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

Constructors of CardLayout Class

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.

Commonly Used Methods of CardLayout Class

- public void next(Container parent): is used to flip to the next card of the given container.
- public void previous(Container parent): is used to flip to the previous card of the given container.

- public void first(Container parent): is used to flip to the first card of the given container.
- public void last(Container parent): is used to flip to the last card of the given container.
- public void show(Container parent, String name): is used to flip to the specified card with the given name.

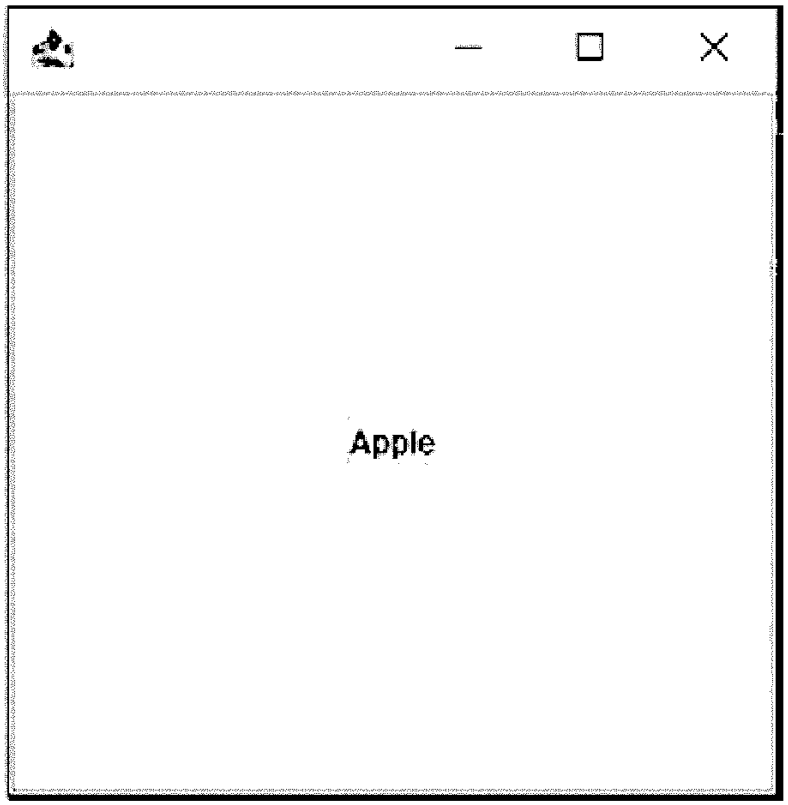
Example:

```
import java.awt.*;

import javax.swing.*;

import java.awt.event.*;

class CardLayoutExample1 extends JFrame implements ActionListener

{

   CardLayout crd;

   JButton btn1, btn2, btn3;

   Container cPane;

   CardLayoutExample1()

   {

      cPane = getContentPane();

      crd = new CardLayout();



      cPane.setLayout(crd);

      btn1 = new JButton("Apple");
```

```
        btn2 = new JButton("Boy");
        btn3 = new JButton("Cat");
        btn1.addActionListener(this);
        btn2.addActionListener(this);
        btn3.addActionListener(this);
        cPane.add("a", btn1);
        cPane.add("b", btn2);
        cPane.add("c", btn3);
    }
    public void actionPerformed(ActionEvent e)
    {
        crd.next(cPane);
    }
    public static void main(String argvs[])
    {
CardLayoutExample1 crdl = new CardLayoutExample1();
        crdl.setSize(300, 300);
        crdl.setVisible(true);
```

```
        crdl.setDefaultCloseOperation(EXIT_ON_CLOSE);

    }

    }
```

Output:



3.Exploring Swings –JFrame, JLabel, The Swing Buttons, JTextField.

Swing Overview

Swing is a part of Java's Java Foundation Classes (JFC) for building graphical user interfaces (GUIs).

Provides lightweight, platform-independent components.

Found in the javax.swing package.

1.JFrame

A top-level container used to create a window.

Represents the main application window.

Methods:

setTitle(String title) - Sets the window title.

setSize(int width, int height) - Sets the size of the window.

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE) - Specifies the action on closing the window.

setVisible(boolean) - Makes the window visible.

Example

```
JFrame frame = new JFrame("My Window");

frame.setSize(400, 300);

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

frame.setVisible(true);
```

2.JLabel

- Used to display a short string or an image icon.
- Cannot be edited by the user.

  Methods:

  - setText(String text) - Sets the text on the label.
  - setIcon(Icon icon) - Sets an image icon.

Example

```
JLabel label = new JLabel("Hello, Swing!");

frame.add(label); // Add label to JFrame
```

Swing Buttons

Buttons allow user interaction via clicks.

Common Swing buttons:

1.JButton: A standard push button.

Example

```
JButton button = new JButton("Click Me");
button.addActionListener(e -> System.out.println("Button Clicked!"));
frame.add(button);
```

2.JCheckBox: A box that can be checked or unchecked.

Example

```
JCheckBox checkBox = new JCheckBox("Check Me");
frame.add(checkBox);
```

3.JRadioButton: A button that is part of a group, allowing single selection.

Example

```
JRadioButton radioButton = new JRadioButton("Option 1");
frame.add(radioButton);
```

JTextField

- A text input field for single-line text.

  Methods:

  - getText() - Retrieves the text entered by the user.
  - setText(String text) - Sets the initial text.

Example:

```
JTextField textField = new JTextField(20); // 20 columns
frame.add(textField);
String input = textField.getText(); // Retrieve input
```

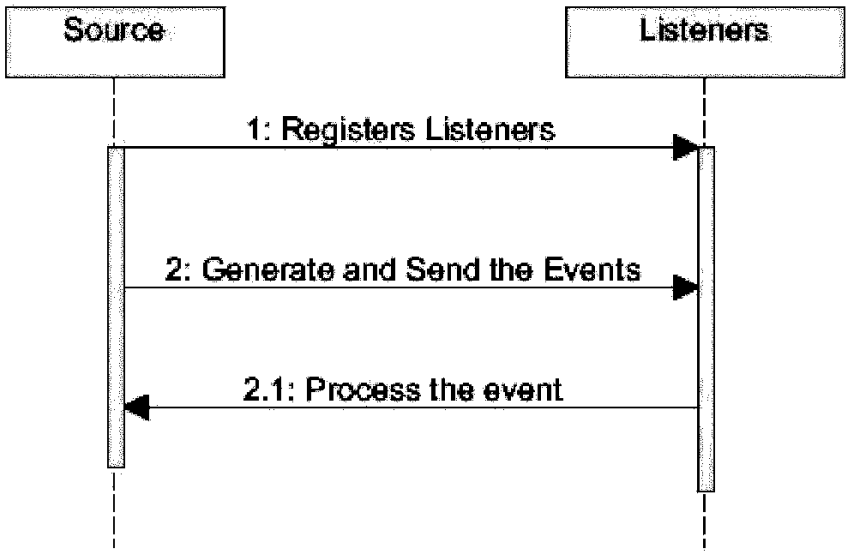
Here's an example combining the components:

```
import javax.swing.*;
```

```
public class SwingExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Swing Example");
        frame.setSize(400, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         JLabel label = new JLabel("Enter your name:");
        JTextField textField = new JTextField(15);
       JButton button = new JButton("Submit");
 button.addActionListener(e -> System.out.println("Hello, " + textField.getText() + "!"));
 JPanel panel = new JPanel(); // A container for layout
        panel.add(label);
        panel.add(textField);
        panel.add(button);
       frame.add(panel);
        frame.setVisible(true);
    }
}
```

Event Handling in Java Using the Delegation Event Model

The Delegation Event Model is the standard mechanism in Java for handling events. It is used extensively in Java Swing and AWT for implementing interactive applications. Below is an explanation of how it works:



1. Core Concepts of Delegation Event Model

1. Event Source:
    - The component that generates an event (e.g., a button, text field, or checkbox).
    - When an event occurs (e.g., a button click), the event source notifies the registered listeners.
2. Event Listener:
    - An object that listens for events and handles them.
    - It must implement a specific interface, such as ActionListener, KeyListener, or MouseListener.
    - The event listener must be registered with the event source to receive notifications.
3. Event Object:
    - An instance of a class derived from the java.util.EventObject class.
    - It encapsulates information about the event (e.g., the source of the event and details of the action performed).
    - Examples include:
        - ActionEvent (e.g., button clicks)
        - MouseEvent (e.g., mouse clicks)
        - KeyEvent (e.g., key presses)

2. How the Delegation Event Model Works

1. Event Generation:
    - When the user interacts with a GUI component, an event is generated by the Event Source.
2. Event Propagation:
    - The event object is created and passed to the registered Event Listener(s).
3. Event Handling:
    - The listener responds to the event by executing its defined logic.
    - This separation of event generation and event handling is key to the Delegation Event Model.

3. Key Steps in Handling Events

1. Implement the Listener Interface:
    - Write a class that implements the required event listener interface (e.g., ActionListener).
2. Register the Listener:
    - Attach the listener to the source component using methods like addActionListener() or addMouseListener().
3. Handle the Event:
    - Implement the listener method (e.g., actionPerformed()) to define what happens when the event occurs.

. Common Listener Interfaces

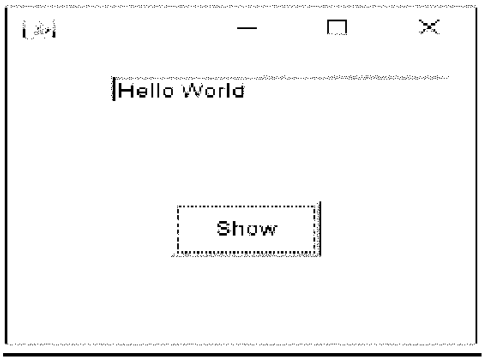
Listener Interface	Description
ActionListener	Handles action events (e.g., button clicks).
MouseListener	Handles mouse events (e.g., clicks, enters, exits).
KeyListener	Handles keyboard events (e.g., key presses).
FocusListener	Handles focus events (e.g., gaining or losing focus).
WindowListener	Handles window events (e.g., opening, closing, minimizing).

EXAMPLE

```
import java.awt.*;

import java.awt.event.*;

class EventHandling extends Frame implements ActionListener
```

```
{
   TextField textField;
   EventHandling ()
   {
      textField = new TextField ();
      textField.setBounds (60, 50, 170, 20);
      Button button = new Button ("Show");
      button.setBounds (90, 140, 75, 40);
      button.addActionListener (this);
      add (button);
      add (textField);
      setSize (250, 250);
      setLayout (null);
      setVisible (true);
   }
   public void actionPerformed (ActionEvent e)
   {
      textField.setText ("Hello World");
   }
   public static void main (String args[])
   {
      new EventHandling ();
   }
}
```

OUTPUT



JDBC(Java Database Connectivity)

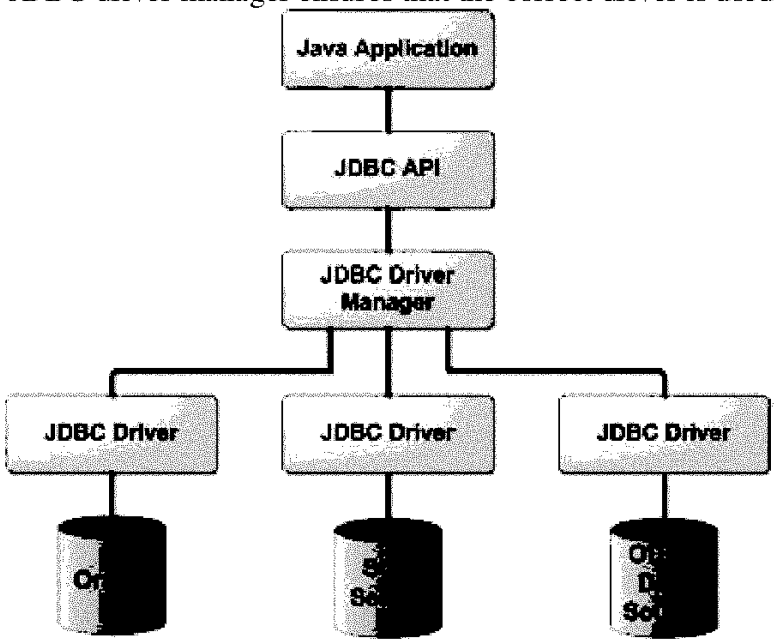
- JDBC stands for Java Database Connectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.
- The JDBC library includes APIs for each of the tasks mentioned below
  - Making a connection to a database.
  - Creating SQL or MySQL statements.
  - Executing SQL or MySQL queries in the database.
  - Viewing & Modifying the resulting records.

JDBC Architecture

- DBC Architecture consists of two layers –
  - **JDBC API:** This provides the application-to-JDBC Manager connection.
  - **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.
- The JDBC API uses a driver manager and database-specific drivers to provide

transparent connectivity to heterogeneous databases.

- The JDBC driver manager ensures that the correct driver is used to access each data source.



- The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases

- **JDBC API :** The JDBC API provides programmatic access to relational data from the Java programming language.
    - Using the JDBC API, applications can execute SQL statements, retrieve results, and propagate changes back to an underlying data source.
    - The JDBC API can also interact with multiple data sources in a distributed, heterogeneous environment.
- **DriverManager:** This class manages a list of database drivers.
- **Driver:** This interface handles the communications with the database server.

Common JDBC Components

The JDBC API provides the following interfaces and classes –

- **DriverManager:** This class manages a list of database drivers.
    - Matches connection requests from the java application with the proper database driver using communication sub protocol.
    - The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.

- **Driver:** This interface handles the communications with the database server.
    - Use DriverManager objects, which manages objects of this type.
    - It also abstracts the details associated with working with Driver objects.
- **Connection:** This interface with all methods for contacting a database.
    - The connection object represents communication context, i.e., all communication with database is through connection object only.
- **Statement:** You use objects created from this interface to submit the SQL statements to the database.

Commonly used methods of Statement interface:

The important methods of Statement interface are as follows:

1) public ResultSet executeQuery(String sql): is used to execute SELECT query. It returns the object of ResultSet.

2) public int executeUpdate(String sql): is used to execute specified query, it may be create, drop, insert, update, delete etc.

3) public boolean execute(String sql): is used to execute queries that may return multiple results.

4) public int[] executeBatch(): is used to execute a batch of commands.

- ResultSet: These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
- SQLException: This class handles any errors that occur in a database application.

Steps to connect to the database in java

They are as follows:

- Import JDBC Packages
- Register the driver class
- Creating connection
- Creating statement

- Executing queries
- Closing connection

Import JDBC Packages:

- Add import statements to your Java program to import required classes in your Java code.
- Syntax : import java.sql.* ; // for standard JDBC programs

Register the driver class :

Approach 1 :

- The forName() method of "Class" class is used to register the driver class.
- This method is used to dynamically load the driver class.
- Syntax of forName() method :
    - public static void forName(String className)throws ClassNotFoundException
- Eg :- Class.forName("com.mysql.jdbc.Driver");

Approach 2:

- The second approach you can use to register a driver, is to use the static DriverManager.registerDriver() method.

- Syntax :- DriverManager.registerDriver(Driver_object);
- Eg:-

```
Driver myDriver = new com.mysql.jdbc.Driver (); //creating object of mysql
driver DriverManager.registerDriver( myDriver ); //register driver
```

Create the connection object :

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

- public static Connection getConnection(String url)throws SQLException
- public static Connection getConnection(String url,String name,String password)throws SQLException

- Eg:-

```
Connection
con=DriverManager.getConnection("jdbc:mysql://localhost:3306/sonoo","root","mysql");
```

Create the Statement object :

- Syntax :- public Statement createStatement()throws SQLException
- Eg:- Statement stmt=con.createStatement();

Execute the query :

- The executeQuery() method of Statement interface is used to execute queries to the database.
- This method returns the object of ResultSet that can be used to get all the records of a table.

- Syntax of executeQuery() method
  - public ResultSet executeQuery(String sql)throws SQLException
- Example to execute query

```
ResultSet
rs=stmt.executeQuery("select *
from emp"); while(rs.next())
{
System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```

Close the connection object :

- By closing connection object statements and ResultSet will be closed automatically.
- The close() method of Connection interface is used to close the connection.

- Syntax of close() method
  - public void close()throws SQLException
- Example to close connection
  - con.close();

Programs

1.

```
//Java program to display the contents of table emp
import java.sql.*;
class MysqlCon

{
public static void main(String args[])
{
try
{
Class.forName("com.mysql.jdbc.Driver");
Connection
con=DriverManager.getConnection("jdbc:mysql://localhost:3306/sonoo","root","mysql");
//here sonoo is database name, root is
username and password Statement
stmt=con.createStatement();
ResultSet rs=stmt.executeQuery("select * from
emp"); while(rs.next())
System.out.println(rs.getInt(1)+" "+rs.getString(2)+"
"+rs.getString(3)); con.close();
}
catch(Exception e)
{
 System.out.println(e);
}
}
}
```

2.

```
//Java program to insert the contents into table emp
import java.sql.*;
class MysqlCon
{
public static void main(String args[])
{
try
{
Class.forName("com.mysql.jdbc.Driver");
```

```
Connection
con=DriverManager.getConnection("jdbc:mysql://localhost:3306/sonoo","root","mysql");
//here sonoo is database name, root is username and password

Statement stmt=con.createStatement();
stmt.executeUpdate("inser into emp
values(3,'test','test')"); con.close();
}
catch(Exception e)
{
System.out.println(e);
} } }
```

Different Database Queries :-

- To create a database:

  ```
  stmt.executeUpdate("create database sonoo ");
  ```

- To drop Database:

  ```
  stmt.executeUpdate("drop database sonoo ");
  ```

- To create table:

  ```
  stmt.executeUpdate("create table emp(id integer,name
  varchar(20),designation varchar(20)) ");
  ```

- To drop table :

  ```
  stmt.executeUpdate("drop table emp ");
  ```

- Select specified number of fields:-

  ```
  ResultSet rs=stmt.executeQuery("select name from
  emp"); while(rs.next())
      System.out.println(rs.getString("name")); or
      System.out.println(rs.getString(1));
  ```

- Select according to condition:-

  ```
  ResultSet rs=stmt.executeQuery("select name
  from emp where id=1"); while(rs.next())
      System.out.println(rs.getString("name")); or
      System.out.println(rs.getString(1));
  ```

- Update query:

  ```
  stmt.executeUpdate("update emp set designation=AP where id=3 ");
  ```

- Delete Query:

```
stmt.executeUpdate("delete from emp where id=1 ");
```

- Select datas by sorting them in descending according to the total mark

```
rs=stmt.executeQuery("select roll, name from student order by mark1+mark2 desc;");
while(rs.next())
{
    i++;
    System.out.println(i+"\t"+rs.getInt(1)+"\t"+rs.getString(2));
}
```

PreparedStatement interface

- The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized queries.

Method	Description
public void setInt(int paramIndex, int value)	sets the integer value to the given parameter index.
public void setString(int paramIndex, String value)	sets the String value to the given parameter index.
public void setFloat(int paramIndex, float value)	sets the float value to the given parameter index.
public void setDouble(int paramIndex, double value)	sets the double value to the given parameter index.
public int executeUpdate()	executes the query. It is used for create, drop, insert, update, delete etc.
public ResultSet executeQuery()	executes the select query. It returns an instance of ResultSet.

- Syntax:-public PreparedStatement prepareStatement(String

query)throws SQLException

Methods of PreparedStatement interface

Eg:-

```
//Java program to insert the contents into table emp using prepared statement
import java.sql.*;
class InsertPrepared
{
public static void main(String args[])
{
try{
Class.forName("co
m.mysql.jdbc.Driv
er ");
 Connection
  con=DriverManager.getConnection("jdbc:mysql://localhost:3306/sonoo","root","mysql");
  PreparedStatement stmt=con.prepareStatement("insert into Emp values(?,?)");
stmt.setInt(1,101);//1 specifies the first
parameter in the query
stmt.setString(2,"Ratan");
 int
i=stmt.executeUpd
ate();
System.out.println(
i+" records
inserted");
con.close();

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

To display contents of a table using prepared statement :-

PreparedStatement

```
stmt=con.prepareStatement("select * from emp");
ResultSet rs=stmt.executeQuery();
while(rs.next())
{
        System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```

To delete records of a table using prepared statement :-

```
PreparedStatement stmt=con.prepareStatement("delete from emp where
id=?"); stmt.setInt(1,101);
 int i=stmt.executeUpdate();
System.out.println(i+"
records deleted");
```

Short Answer Questions

1. What does the Liskov Substitution Principle state?
2. How does Swing differ from AWT in terms of component rendering?
3. What is the purpose of JFrame in a Swing application?
4. Mention three commonly used Swing controls and their purposes.
5. Define event source and give two examples from Java Swing.
6. What is DriverManager in JDBC?
7. How does executeQuery() differ from executeUpdate() in JDBC?
8. What is MVC architecture and how is it used in Swing applications?
9. List any three components from the javax.swing package.
10. Write the syntax to register a MouseListener to a component.
11. Mention the steps involved in handling events using the Delegation Event Model.
12. What is the difference between Statement and PreparedStatement in JDBC?

Essay Questions

1. Describe each of the SOLID principles with appropriate Java examples and explain how they improve software quality.
2. Create a Java Swing program for a feedback form using JFrame, JTextArea, JButton, and JLabel. Include event handling to display submitted feedback.

3. Write a Java program to connect to a MySQL database using JDBC. Insert, display, update, and delete student records using PreparedStatement.
4. Explain in detail the Delegation Event Model. Illustrate how event handling works using listeners with a Java example.
5. Design a Java Swing GUI that accepts two numbers and performs basic arithmetic operations (Add, Subtract, Multiply, Divide) using buttons. Implement action listeners for each.
6. Explain the different types of JDBC drivers. Compare their advantages and use cases.
7. Discuss the structure and use of layout managers in Swing. Write a Java program using GridLayout or BorderLayout to arrange components.
8. Write and explain a program that demonstrates the use of MouseListener and KeyListener interfaces in Java Swing.