History of Java:-

- James Gosling, Mike Sheridan and Patrick Naughton initiated the Java language project in June 1991.
- Java was originally developed by James Gosling at Sun Microsystems and released in 1995.

• The language was initially called Oak after an oak tree that stood outside Gosling's office.

• Later the project went by the name Green and was finally renamed Java, from Java coffee, a type of coffee from Indonesia.

• Gosling and his team did a brainstorm session and after the session, they came up with several names such as JAVA, DNA, SILK, RUBY, etc.

• Sun Microsystems released the first public implementation as Java 1.0 in 1996.

Java Platforms:-

1. Java SE ¬ Java Platform Standard Edition.
   It is also called as Core Java.
   For general purpose use on Desktop PC's, servers and similar devices.
2. Java EE ¬ Java Platform Enterprise Edition.
   It is also called as advanced Java / enterprise java / web java.
   ¬ Java SE plus various API's which are useful client-server applications.
3. Java ME ¬ Java Platform Micro Edition.
   ¬ Specifies several different sets of libraries for devices with limited storage, display, and power capacities.
   ¬ It is often used to develop applications for mobile devices, PDAs, TV set-top boxes and printers.
4. Java Card ¬ A technology that allows small Java-based applications (applets) to be run securely on smart cards and similar small-memory devices.

Components Of Java Platform:-

SDK = Development Tools + Documentation + Libraries + Runtime Environment. • JDK = Java Development Tools + Java Docs + rt.jar + JVM. ¬ JDK : Java Development Kit.

¬ It is a software, that need to be install on developers machine.

¬ We can download it from oracle official website.

• JDK = Java Development Tools + Java Docs + JRE[rt.jar + JVM].

• JRE : Java Runtime Environment.

¬ rt.jar and JVM are integrated part of JRE.

¬ JRE is a software which comes with JDK. We can also download it separately. ¬ To deploy application, we should install it on client's machine.

• rt.jar file contains core Java API in compiled form.

• JVM : An engine, which manages execution of Java application.

JDK installation Directory Structure:-

1.bin :- it contains java lang tools Ex. javac ,javah java etc

2.include :-JNI stands for java native interface.

3.lib :it contain library file which is required use java lang tool

4.src :-if we extract src.zip then src directory gets creaated

5.docs :-it contains documentation of java API

6.jre :-it contains JVM and rt.jar file

7.db :-it contains database specific files(driver ,sample,database)

Session 1 & 2: Java Basics:-

Java Language and Its Features:-

Java is a high-level, object-oriented, and platform-independent programming language. Some key features include:

- Platform Independent: Java programs are compiled into bytecode that runs on any device with a JVM.
- Object-Oriented: Follows the principles of OOP such as inheritance, encapsulation, and polymorphism.
- Secure: Includes runtime security features like bytecode verification and access control.
- Robust: Offers strong memory management and built-in exception handling.
- Multithreaded: Supports concurrent execution of two or more threads.
- Distributed: Enables building distributed applications using RMI and EJB.
-

JDK, JRE, and JVM:-

- **JDK (Java Development Kit)**: Full-featured kit including compiler, debugger, and JRE. Used for developing Java programs.
-

- **JRE (Java Runtime Environment)**: Includes JVM and class libraries. Required to run Java applications.
- **JVM (Java Virtual Machine)**: Abstract machine that loads, verifies, and executes Java bytecode.

Component of JVM :-

1.ClassLoader Subsystem:

a. Bootstrap class loader

b. Extension class loader

c. Application class loader

d. User defined class loader

2.Runtime Data Area:

a. Method Area

b. Heap

c. Java stack

d. Pc register

e. Native method stack

3.Execution Engine :

a. Interpreter

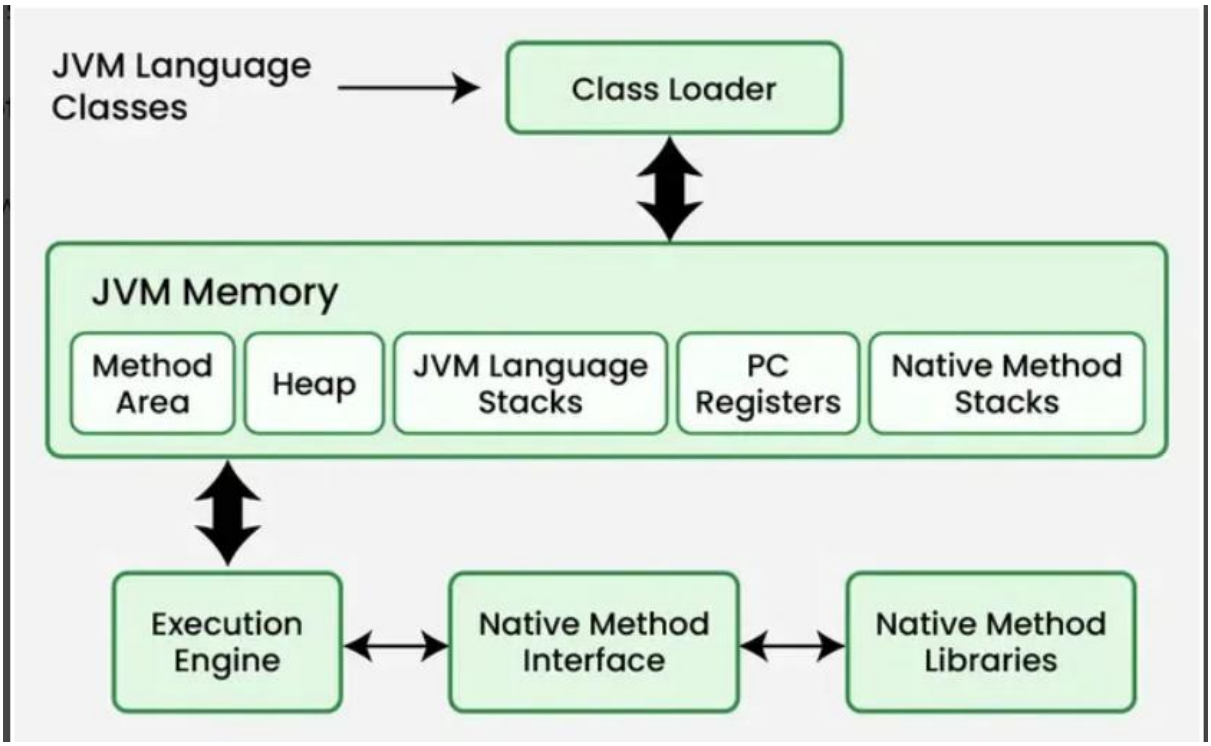
b. Just in time compiler

c. garbage collector

◊The Bytecode is object oriented assembly lang code design for JVM ◊If we create blank java file then it wont give me compile time error. But runtime error gave me Error: could not find or load main class program. calling main method is job of JVM not complier.

JVM Architecture Overview:-

-



Data types :-

Data type describes:

- Memory is required to store the data
- Kind of data memory holds Operations to perform on the data
- Java is strictly type checked language.

In java, data types are classified as:

- Primitive types or Value types
- Non-primitive types or Reference types

```
Data types
    |- Primitive types (Value types)
    |       |- Boolean: boolean
    |       |- Character: char
    |       |- Integral: byte, short, int, long
    |       |- Floating-point: float, double
    |
    |- Non-Primitive types (Reference types)
            |- class
            |- interface
            |- enum
            |- Array
```

1. boolean (size is not specified)

2. byte (size is 1 byte)

- 3. char (size is 2 bytes)

4. short (size is 2 bytes)

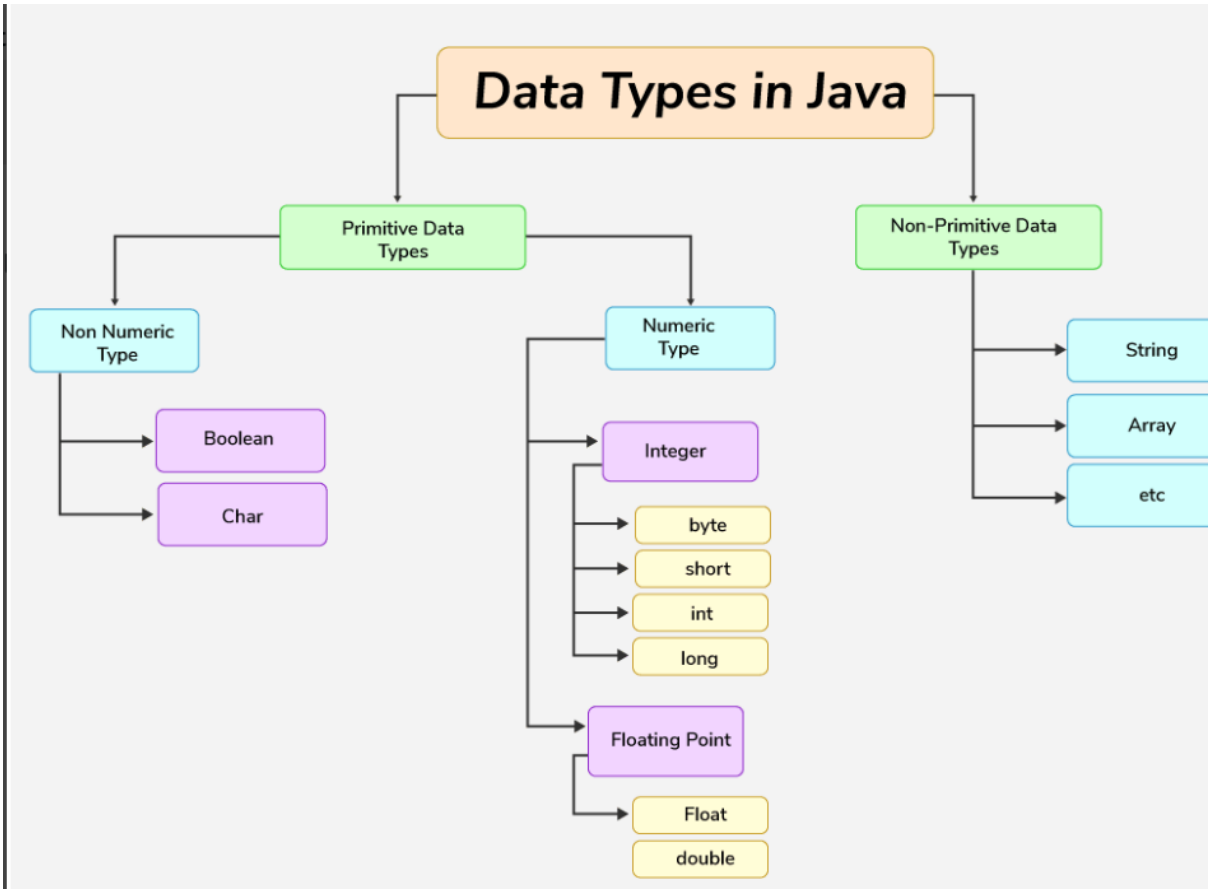
5. int (size is 4 bytes)

6. float (size is 4 bytes)

7. double (size is 8 bytes)

8. long (size is 8 bytes)

- primitive types(boolean, byte, char, short, int ,float, double, long) are not classes in Java.



Widening:-

• Process of converting value of variable of narrower type into wider type is called widening.

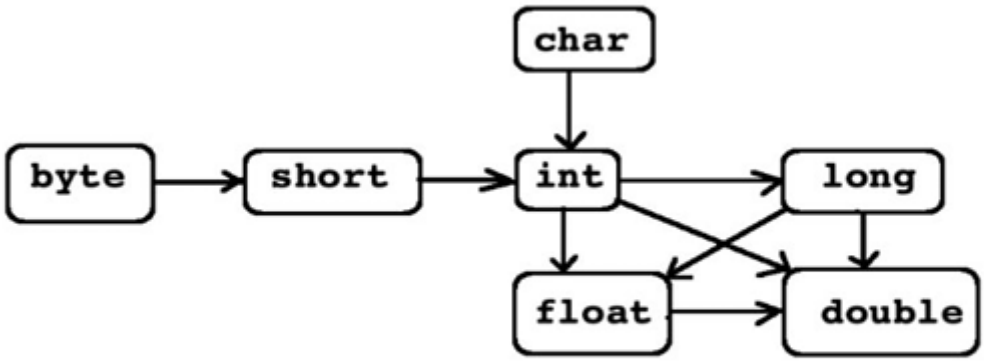
• In case of widening, explicit type casting is optional.

```
class Program
{
public static void main(String[] args) {
 int num1 = 10; //Initialization
 //double num2 = (double)num1; //Widening //OK:10.0 typecasting
```

- double num2 = num1; //Widening //OK:10.0

System.out.println("Num2 : "+num2); }

}



Widening Conversion

Narrowing:-

• Process of converting value of variable of wider type into narrower type is called narrowing.

• In case of narrowing, explicit type casting is mandatory.

Class Program{

public static void main1(String[] args) {

double num1 = 10.5; //Initialization

double num2 = num1; //Initialization

System.out.println("Num2 : "+num2);

}

public static void main(String[] args)

{ double num1 = 10.5; //Initialization

//int num2 = num1; //error: incompatible types: possible lossy conversion from double to int

int num2 = (int)num1; //Narrowing //OK : 10

System.out.println("Num2 : "+num2); }

}

-

Variables:-

- A variable is a container which holds a value. It represents a memory location.
- A variable is declared with data type and initialized with another variable or literal.

In Java, variable can be

- Local: Within a method -- Created on stack.
- Non-static/Instance field: Within a class - Accessed using object.
- Static field: Within a class - Accessed using class-name.

Operators :-

- Java divides the operators into the following catgories:

1. 1Arithmetic operators: +, -, *, /, %
2. 2Assignment operators: =, +=, -=, etc.
3. 3Comparison operators: ==, !=, , <=, >=, instanceof
4. 4.Logical operators: &&, ||, !

- Combine the conditions (boolean - true/false)

5. Bitwise operators: &, |, ^, ~, <>, >>>
6. Misc operators: ternary ?:, dot .

- Dot operator: ClassName.member, objName.member

Session 3 & 4: OOP Concepts and Java Classes:-

Wrapper Class:-

• In Java, primitive types are not classes. But for every primitive type, Java has defined a class. It is called wrapper class.

• All wrapper classes are final.

• All wrapper classes are declared in java.lang package.

• Uses of Wrapper class :-

1. To parse string(i.e. to convert state of string into numeric type).

2. To store value of primitive type into instance of generic class, type argument must be wrapper class.

- ¬ Stack stk = new Stack( ); //Not OK

¬ Stack stk = new Stack( ); //OK

◊string convert into interger then used wrapper class integer.parseInt();

int number1=Integer.parseInt("123"); float number2=Float.parseFloat("123.45f");
Double number3=Double.parseDouble("123.45f");

1. boolean => java.lang.Boolean

2. byte => java.lang.Byte

3. char => java.lang.Character

4. short => java.lang.Short

5. int => java.lang.Integer

6. float => java.lang.Float

7. double => java.lang.Double

8. long => java.lang.Long

Applications of wrapper classes:-

- Use primitive values like objects :

// int 123 converted to Integer object holding 123.

Integer i = new Integer(123);

- Convert types

Integer i = new Integer(123);

byte b = i.byteValue();

long l = i.longValue();

short s = i.shortValue();

double d = i.doubleValue();

String str = i.toString();

String val = "-12345";

int num = Integer.parseInt(val);

- Get size and range of primitive types :

System.out.printf("int size: %d bytes = %d bits\n", Integer.BYTES, Integer.SIZE);
System.out.printf("int max: %d, min: %d\n", Integer.MAX_VALUE, Integer.MIN_VALUE);

-

- Helper/utility methods:

```
System.out.println("Sum = " + Integer.sum(22, 7));
System.out.println("Max = " + Integer.max(22, 7));
System.out.println("Min = " + Integer.min(22, 7));
```

Boxing:-

- Converting from value (primitive) type to reference type.
  int x = 123;
  Integer y = new Integer(x); //
- boxing Java 5 allows auto-conversion from primitive type to corresponding wrapper type. This is called as "auto-boxing".
  int x = 123; Integer y = x; // auto-boxing

Unboxing:-

- Converting from reference type to value (primitive) type.
  Integer y = new Integer(123);
  int x = y.intValue(); // unboxing
- Java 5 allows auto-conversion from wrapper type to corresponding value type. This is called as "auto unboxing".
  Integer y = new Integer(123); int x = y; // auto-unboxing

Class and Object:-

- Class is collection of logically related data members ("fields"/attributes/properties) and the member functions ("methods"/operations/messages) to operate on that data.
- A class is user defined data type. It is used to create one or more instances called as "Objects".
- Class is blueprint/prototype/template of the object; while Object is an instance of the class.
- Class is logical entity and Object represent physical real-world entity. Class represents group of such instances which is having common structure and common behavior.
- class is an imaginary entity.
  I. Example: Car, Book, Laptop etc.
  II. Class implementation represents encapsulation.
  III. e.g. Human is a class and You are one of the object of the class

Instance:-

- Definition

1. In java, object is also called as instance

2. An entity, which is having physical existance is called as instance.

3. An entity, which is having state, behavior and identity is called as instance.

- instance is a real time entity.

- Example: "Tata Nano", "Java Complete Reference", "MacBook Air" etc.

Types of methods in a Java class.

Methods are at class-level, not at object-level. In other words, same copy of class methods is used by all objects of the class.

Parameterless Constructor

- In Java, fields have default values if unintialized. Primitive types default value is usually zero (refer data types table) and Reference type default value is null. Constructor should initialize fields to the desired values.
- Has same name as of class name and no return type. Automatically called when object is created (with no arguments).
- If no construcor defined in class, Java compiler provides a default construcor (Parameterless).
- Human obj = new Human();

Parameterized Constructor:-

- Constructor should initialize fields to the given values.
- Has same name as of class name and no return type. Automatically called when object is created (with arguments).
- Human obj = new Human(40, 76.5, 5.5);

Inspectors/Getters :-

- Used to get value of the field outside the class.
- double ht = obj.getHeight();

Mutators/Setters :-

- Used to set value of the field from outside the class. It modifies state of the object.
- obj.setHeight(5.5);

- Executing a method on object is also interpreted as

1. Calling member function/method on object.
- 2. Invoking member function/method on object.

3. Doing operation on the object.
4. Passing message to the object.

- Each object has State:

1. Represents values of fields in the object.
2. Behaviour: Represents methods in the class and also depends on Object state.
   Identity: Represents uniqueness of the object.

Constructor :-

It is a method of class which is used to initialize instance.

Constructor is a special syntax of Java because:

1. Its name and class name is always same.

2. It doesn't have any return type

3. It is designed to call implicitly.

4. In the lifetime on instance, it gets called only once.

```
public Date( ){  //Constructor of the class
System.out.println("Inside constructor");
 Calendar c = Calendar.getInstance( );
 this.day = c.get( Calendar.DATE );
this.month = c.get( Calendar.MONTH ) + 1;
this.year = c.get( Calendar.YEAR );



}
```

- We can not call constructor on instance explicitly.

```
Date date = new Date(); //OK
date.Date( ); //Not ok
```

- We can use any access modifer on constructor.
- If constructor is public then we can create instance of a class inside method of same class as well as different class.
-

- If constructor is private then we can create instance of class inside method of same class only.
- Types of constructor in Java:

1. Parameterless constructor:-
   A constructor which do not have any parameter is called parameterless constructor.
2. Parameterized constructor:-

Constructor of a class which take parameters is called parameterized constructor.

3. Default constructor:-
    - If we do not define any constructor(no parameterless, no parameterized) inside class then compiler generate one constructor for the class by default. It is called default constructor.
    - Compiler generated default constructor is zero parameter i.e parameterless constructor.
    - Compiler never generate default parameterized constructor. It means that, if we want to create instance with arguments then we must define parameterized constructor inside class.

Constructor chaining :-

- In Java, we can call constructor from another constructor. It is called constructor chaining.
- For constructor chaining, we should use this statement.
- this statement must be first statement inside constructor body.

this reference :-

- If we call non static method on instance then non static method get this reference.
- this reference contains reference of current/calling instance.
- Using this reference, non static method and non static field can communicate with each other. Hence this reference is considered as a link/connection between them.
- this reference is considered as a first parameter to the method. Hence it gets space once per method call on Java Stacks.
- We can not use this reference to access local variable. We should use this reference to access non static field/method.
- Use of this reference to access non static field/method is optional.
- If name of the local variable and name of field is same then to refer field we should use this reference.

-

```
class Complex{
    private int real;
    public void setReal( int real ){
        this.real = real;
    }
}
```

Definition:-

this reference is implicit parameter available inside every non static method of the class which is used to store reference of current/calling instance.

Object Oriented Programming Structure / System(OOPS):-

Major pillars / parts / elemets :-

1. Abstraction : To achive simplicity
2. Encapsulation : To achive data hiding and data security
3. Modularity : To minimize module dependency
4. Hierarchy : To achive reusability

Minor pillars / parts / elemets :-

1. Typing : To minimize maintenance of the system.
2. Concurrency : To utilize hardware resources efficiently.
3. Persistence : To maintain state of the instance on secondary storage

Advantages of OOPS:-

1. To achieve simplicity 2. To achieve data hiding and data security. 3. To minimize the module dependency so that failure in single part should not stop complete system.

4. To achieve reusability so that we can reduce development time/cost/efforts. 5. To reduce maintenance of the system.

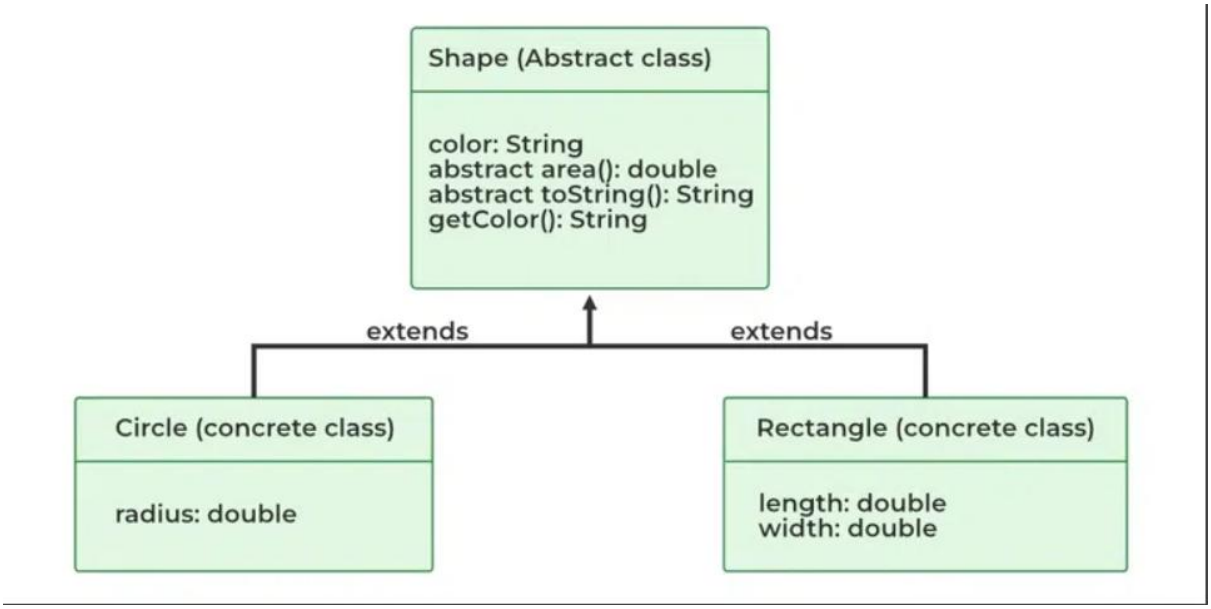
6. To fully utilize hardware resources.

7. To maintain state of object on secondary storage so that failure in system should not impact on data.

•

Abstraction:-

- It is a major pillar of oops.
- Abstraction is the process of getting essential things from system/object. Abstraction focuses on the essential characteristics of som object, relative to the perspective of viewer. In simple word, abstraction changes from user to user.
- Abstraction describes external behavior of an instance.
- Creating instance and calling methods on it represents abstraction programatically.



How to Achieve Abstraction in Java?

Java provides two ways to implement abstraction, which are listed below:

- Abstract Classes (Partial Abstraction)
- Interface (100% Abstraction)

Real-Life Example of Abstraction:-

- **The television remote control** is the best **example** of **abstraction**. It simplifies the interaction with a TV by hiding all the complex technology. We don't need to understand how the tv internally works, we just need to press the button to change the channel or adjust the volume.
- *Consider a real-life example of a man driving a car. The man only knows that pressing the accelerator will increase the speed of a car or applying brakes will stop the car, but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc. in the car. This is what abstraction is.*

Encapsulation:-

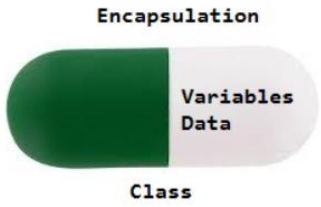
-

- It is a major pillar of oops.
- Definition:-

1. To achive abstraction, we need to provide some implementation. This implementation of abstraction is called encapsulation.
2. Binding of data and code together is called as encapsulation.
   - Hiding represents encapsulation.
   - Class implementation represents encapsulation.
   - Use of encapsulation: -

   1. To achive abstraction
   2. To achive data hiding(also called as data encapsulation).
   - Process of declaring field private is called as data hiding.
   - Data hiding help to achive data security.

This image below demonstrates the concept of encapsulation, where a class (represented by the capsule) hides its internal data (variables) and only exposes a controlled interface (encapsulation).



// Java Program which demonstrate Encapsulation:-

```
class Account {


   // Private data members (encapsulated)

   private long accNo; // Account number

   private String name;

   private String email;

   private float amount;


   // Public getter and setter methods (controlled access)

   public long getAccNo() { return accNo; }

   public void setAccNo(long accNo) { this.accNo = accNo; }
```

```
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }
    public float getAmount() { return amount; }
    public void setAmount(float amount) { this.amount = amount; }
}

// Main Class
public class Main
{
    public static void main(String[] args)
    {
        // Create an Account object
        Account acc = new Account();

        // Set values using setter methods (controlled access)
        acc.setAccNo(90482098491L);
        acc.setName("ABC");
        acc.setEmail("abc@gmail.com");
        acc.setAmount(100000f);

        // Get values using getter methods
        System.out.println("Account Number: " + acc.getAccNo());
```

- `System.out.println("Name: " + acc.getName());`

```
        System.out.println("Email: " + acc.getEmail());

        System.out.println("Amount: " + acc.getAmount());

    }

}
```

Modularity :-

- It is a major pillar of oops
- Modularity is the process of designing and developing complex system using small parts/modules.
- Main purpose of modularity is to minimize module dependancy.
- Using .jar file, we can achive modularity in Java.
- **Example to understand modularity better:**
- In the object-oriented approach, the concept of modularity revolves around the concept of well-organized interactions between different components. Modularity refers to an organizing structure in which different components of a software system are divided into separate functional units.
- For example, a house or apartment can be viewed as consisting of several interacting units; electrical, heating, cooling, plumbing, structure, etc. Rather than viewing it as one giant jumble of wires, vents, pipes, and boards, the organized architect designing a house or apartment will view them as separate modules that interact in well-defined ways. In doing so, he/she is using the concept of modularity to bring clarity of thought that provides a natural way of organizing functions into distinct manageable units. Likewise, using modularity in a software system can also provide a powerful organizing framework that brings clarity to an implementation.

Hierarchy :-

- It is a major pillar of oops.
- Level/order/ranking of abstraction is called as hierarchy.
- Using hierarchy, we can achive code reusability.
- Application of reusability:
    1. To reduce development time.
    2. To reduce development cost
    3. To reduce develpers effort.
- Types of hierarchy: -
    1. Has-a => Association
    2. Is-a => Generalization(is also called as inheritance)
    3. Use-a => Dependancy 4. Create-a => Instantiation
-

Typing:-

- It is a minor pillar of oops.
- It is also called as polymorphism(poly(many) + morphism(forms/behavior)). polymorphism is a Greek word.
- An ability of any instance to take multiple forms is called as polymorphism. Using typing/polymorphism, we can reduce maintenance of the application. In Java, we can achive polymorphism using two ways:
- Method overloading (It represents compile time polymorphism)
- Method overriding (It represents run time polymorphism)
- In Java, runtime polymorphism is also called as dynamic method dispatch

Concurrency :-

- It is a minor pillar of oops. Concurrency is the process of executing multiple task simultaneously.
- Using thread, we can achive Concurrency in Java. Using Concurrency, we can utilize H/W resources efficiently.

Persistance:-

- It is a minor pillar of oops.
- Process of maintaining state of instance inside file / database is called as Persistance.
- We can implement it using file handing(serialization / derserialization) and database programming(JDBC).

Access modifiers (for type members):-

- private:-
  When "private" specifier is mentioned before the member field/method. The members are accessible in current class (member OR this.member).

- default: -When no specifier is mentioned before the member field/method. The members are accessible in current class (member OR this.member). The members are accessible in all classes in same package (obj.member OR ClassName.member).

- protected:- When "protected" specifier is mentioned before the member field/method. The members are accessible in current class (member OR this.member). The members are accessible in all classes in same package including its sub-classes (super.member, obj.member OR ClassName.member). The members are accessible in sub classes
- outside the package including its sub-classes (super.member).

- public: -When "public" specifier is mentioned before the member field/method. The members are accessible in current class (member OR this.member). The members are accessible in all other classes (super.member, obj.member OR ClassName.member).

- Scopes:-
  private (lowest)
  default
  protected
  public (highest)

	default	private	protected	public
same class	yes	yes	yes	yes
same package subclass	yes	no	yes	yes
same package non-subclass	yes	no	yes	yes
different package subclass	no	no	yes	yes
different package non-subclass	no	no	no	yes

Session 5:

Static variables and methods:-

static keyword:-

- In OOP, static means "shared" i.e. static members belong to the class (not object) and shared by all objects of the class.
- Static members are called as "class members"; whereas non-static members are called as "instance memebers".
- In Java, static keyword is used for :=

1. static fields
2. static methods
3. static block
4. static import

- Note that, static local variables cannot be created in Java.

Static fields:-

- Copies of non-static/instance fields are created one for each object.
- Single copy of the static/class field is created (in method area) and is shared by all objects of the class.
- Can be initialized by static field initializer or static block.
- Accessible in static as well as non-static methods of the class.
- Can be accessed by class name or object name outside the class (if not private). However, accessing via object name is misleading (avoid it).

Advantages of Static Keyword:-

- Static members use the memory only once and this helps save memory when we have to deal with big programs.
- Static members provide fast access because static members belong to the class not to an object and that's why they can be access faster than regular member.
- We can access static members from anywhere, whether an object of the class has been created or not.
- We can use static final variables to create constant that stays the same throughout the program.

Disadvantages of Static Keyword

- Static members can't be overridden or dynamically bound like instance members.
- Static methods and variables make unit testing difficult due to tight coupling.
- Static variables create a global state, which can lead to unwanted side effects across different parts of the program.
- Static variables stay in memory as long as the program runs, which might cause memory to be used longer than needed.
- Using too many static members can reduce the benefits of object-oriented programming, like hiding data and using inheritance.

Static methods:-

- Methods can be called from outside the class (if not private) using class name or object name. However, accessing via object name is misleading (avoid it).
- When needs to call a method without object, then make it static.
- Since static methods are designed to be called on class name, they do not have "this" reference. Hence, cannot access non-static members in the static method (directly). However, we can access them on an object reference.
- Applications :

1. To initialize/access static fields.
2. Helper/utility methods

```
import java.util.Arrays;

 // in main()
```

```
int[] arr = { 33, 88, 44, 22, 66 };

Arrays.sort(arr);

 System.out.println(Arrays.toString(arr));
```

Static field initializer:-

- Similar to field initializer, static fields can be initialized at declaration.

```
// static field

static double price = 5000.0;

// static final field – constant

 static final double PI = 3.142;
```

Static Method :-

- If we want to access non static members of the class then we should define non static method inside class

```
class Test{

 private int num1;

 public int getNum1( ){

 return this.num1;

 }

public void setNum1( int num1 ){

this.num1 = num1;

}

 }
```

- If we want to access static members of the class then we should define static method inside class.

```
class Test{

private static int num2;

public static int getNum2( ){
```

```
 return Test.num2;

 }

public void setNum2( int num2 ){

 Test.num2 = num2;

} }
```

Why static method do not get this reference?

- If we call non static method on instance then method gets this reference.

- Static method is designed to call on class name.

- Since static method is not designed to call on instance, it doesn't get this reference.

- Since static method do not get this reference, we can not access non static members inside static method.
- In other words, static method can access only static members of the class directly.
- If we want to access non static members inside static method then we need to use instance of the class.
- Inside non static method, we can access static as well as non static members directly.

```
class Test{

 private int num1 = 10;

private static int num2 = 20;

public void printRecord( ){

 System.out.println("Num1 : "+this.num1);

System.out.println("Num2 : "+Test.num2); } }
```

- If static members belongs to same class then we can access static members inside another static method directly(W/O class name).
- If static members belongs to the different class then we can not access static members inside another static method directly(W/O class name). In this case we must use classname and dot operator.
- Inside method, if there is a need to use this reference then we should declare method non static otherwise we should declare method static.
-

Static	Non-Static
Static members have one copy shared across the class.	Non-static members have a separate copy for each instance of the class.
Static members are accessed via the class name.	Non-static members are accessed via an object reference.
Static members cannot be overridden.	Non-static members can be overridden in subclasses.
Static members cannot use this or super keyword.	Non-static members can use this and super keyword.
Static members exist for the duration of the class's lifecycle.	Non-static members exist as long as the object they belong to is alive.

- Method local variable get space once per method call.
- We can declare, method local variable final but we can not declare it static.

- ➢ static variable is also called as class level variable.
- ➢ class level variables should exist at class scope.
- ➢ Hence we can not declare local variable static. But we can declare field static.

❖ **Static block :-**

- Like Object/Instance initializer block, a class can have any number of static initialization blocks, and they can appear anywhere in the class body.
  - Static initialization blocks are executed in the order their declaration in the class.

- A static block is executed only once when a class is loaded in JVM.

Static import :-

- To access static members of a class in the same class, the "ClassName." is optional.
- To access static members of another class, the "ClassName." is mandetory.
- If need to access static members of other class frequently, use "import static" so that we can access static members of other class direcly (without ClassName.).

Singleton class:-

- Design patterns are standard solutions to the well-known problems.
- Singleton is a design pattern.
-

- It enables access to an object throughout the application source code. Singleton class is a class whose single object is created throughout the application.
- To make a singleton class in Java Code:

  step 1: Write a class with desired fields and methods.

  step 2: Make constructor(s) private.

  step 3: Add a private static field to hold instance of the class.

  step 4: Initialize the field to single object using static field initializer or static block.

  step 5: Add a public static method to return the object.

```
public class Singleton {
    // fields and methods
    // since ctor is declared private, object of the class cannot be created
outside the class.
    private Singleton() {
        // initialization code
    }
    // holds reference of "the" created object.
    private static Singleton obj;
    static {
        // as static block is executed once, only one object is created
        obj = new Singleton();
    }
    // static getter method so that users can access the object
    public static Singleton getInstance() {
        return obj;
    }
}
```

  Association:-

  - If "has-a" relationship exist between the types, then use association.
  - To implement association, we should declare instance/collection of inner class as a field inside another class.
  - There are two types of associations :-

    1. Composition
    2. Aggregation

Composition :

-

Represents part-of relation i.e. tight coupling between the objects.

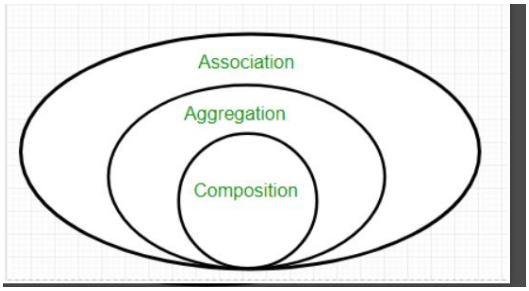
- The inner object is essential part of outer object.
- Engine is part of Car.
- Wall is part of Car.

Association:

Represents has-a relation i.e. loose coupling between the objects.

- The inner object can be added, removed, or replaced easily in outer object.
- Car has a Driver.
- Company has Employees.

Aggregation :
is a specific form of association in which one class, the whole, contains a collection of other classes, the parts; here, however, the lifecycle of the parts does not depend upon the whole. For example, a library and books show aggregation, since books may exist somewhere apart from the library.



Inheritance :-

- If "is-a"/"kind-of" relationship exist between the types, then use inheritance. Inheritance is process -- generalization to specialization.
- All members of parent class are inherited to the child class.

Example:

- Manager is a Employee
- Mango is a Fruit
- Triangle is a Shape

- In Java, inheritance is done using extends keyword

- Java doesn't support multiple implementation inheritance i.e. a class cannot be inherited from multiple super-classes.
- However Java does support multiple interface inheritance i.e. a class can be inherited from multiple super interfaces.

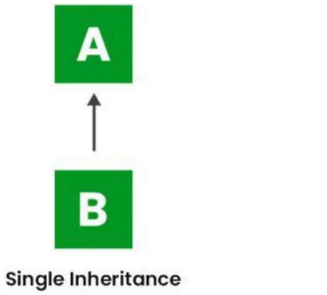
Why Use Inheritance in Java?

- **Code Reusability:** The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.
- **Method Overriding:** Method Overriding is achievable only through Inheritance. It is one of the ways by which Java achieves Run Time Polymorphism.
- **Abstraction:** The concept of abstraction where we do not have to provide all details, is achieved through inheritance. Abstraction only shows the functionality to the user.
- **that** Organizes classes in a structured manner, improving readability and maintainability.

Types of Inheritance in Java

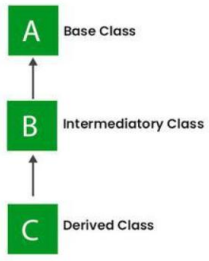
Below are the different types of inheritance which are supported by Java.

- Single Inheritance



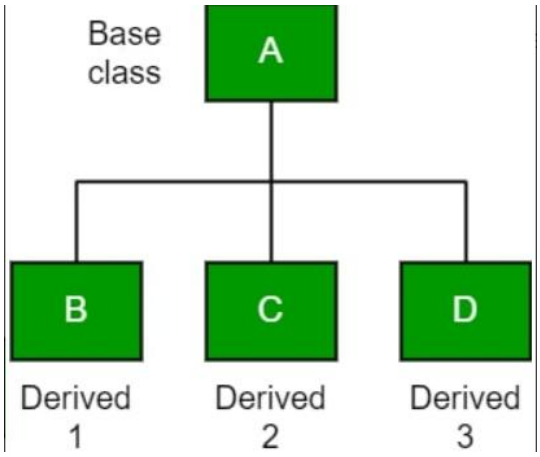
Single Inheritance

- Multilevel Inheritance

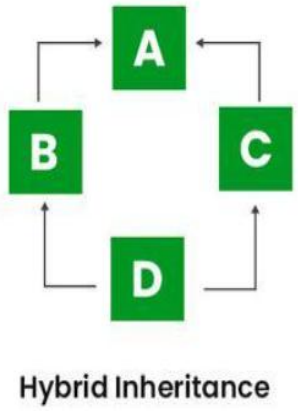


Multilevel Inheritance

- Hierarchical Inheritance
-



- Hybrid Inheritance



Polymorphism:-

- Poly=Many, Morphism=Forms
- Polymorphism is taking many forms.
- OOP has two types of Polymorphism:

    - Compile-time Polymorphism / Static Polymorphism :
- Implemented by "Method overloading". Compiler can identify which method to be called at compile time depending on types of arguments. This is also referred as "Early binding".

- Run-time Polymorphism / Dynamic Polymorphism :
- Implemented by "Method overriding".
- The method to be called is decided at runtime depending on type of object. This is also referred as "Late binding" or "Dyanmic method dispatch".
- Process of calling method of sub class using reference of super class is called as dynamic method dispatch. ###Access Modifier
- •private (lowest)
- default

- protected
- public (highest)

Method overriding :-

Redefining a super-class method in sub-class with exactly same signature is called as "Method overriding".

Programmer should override a method in sub-class in one of the following scenarios :

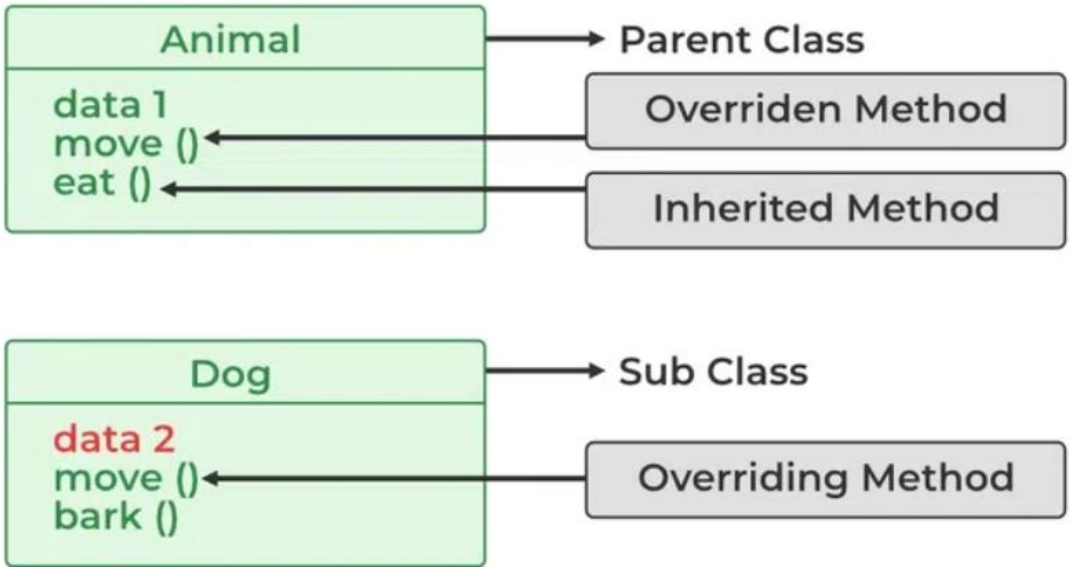
- Super-class has not provided method implementation at all (abstract method).
- Super-class has provided partial method implementation and sub-class needs additional code. Here sub-class implementation may call super-class method (using super keyword).
- Sub-class needs different implementation than that of super-class method implementation.

Rules of method overriding in Java:-

1. Each method in Java can be overridden unless it is private, static or final.
2. Sub-class method must have same or wider access modifier than super-class method.

```
class SuperClass {

 protected Number calculate(Integer i, Float f) {

 // ...

} }

class SubClass extends SuperClass {

 /*protected or*/ public Number calculate(Integer i, Float f) {

 // ... } }
```

-



3. Arguments of sub-class method must be same as of super-class method. The return-type of sub class method can be same or sub-class of the super-class's method's return-type. This is called as "covarient" return-type.
4. Checked exception list in sub-class method should be same or subset of exception list in super class method.
5. If these rules are not followed, compiler raises error or compiler treats sub-class method as a new method.

- Java 5.0 added @Override annotation (on sub-class method) informs compiler that programmer is intending to override the method from the super-class.
- @Override checks if sub-class method is compatible with corresponding super-class method or not (as per rules). If not compatible, it raise compile time error.
- Note that, @Override is not compulsory to override the method. But it is good practice as it improves readability and reduces human errors.

Method Overloading:-

Method overloading in Java is also known as **Compile-time Polymorphism, Static Polymorphism, or Early binding**, because the decision about which method to call is made at compile time. When there are overloaded methods that accept both a parent type and a child type, and the provided argument could match either one, Java prefers the method that takes the more specific (child) type. This is because it offers a better match.

Different Ways of Method Overloading in Java:-

The different ways of method overloading in Java are mentioned below:

- Changing the Number of Parameters
- Changing Data Types of the Arguments

- Changing the Order of the Parameters of Methods

Up-casting & Down-casting :-

- Up-casting: Assigning sub-class reference to a super-class reference. Sub-class "is a" Super-class, so no explicit casting is required. Using such super-class reference, super-class methods overridden into sub-class can also be called.

- Down-casting: Assigning super-class reference to sub-class reference. Every super-class is not necessarily a sub-class, so explicit casting is required.

Garbage collection:-

Garbage collection is automatic memory management by JVM.

If a Java object is unreachable (i.e. not accessible through any reference), then it is automatically released by the garbage collector.

An object become eligible for GC in one of the following cases:

1. Nullify the reference :

```
MyClass obj = new MyClass();

obj = null;
```

2. Reassign the reference:

```
MyClass obj = new MyClass();

obj = new MyClass();
```

3. Object created locally in method.

```
void method()

{ MyClass obj = new MyClass(); // ... }
```

- GC is a background thread in JVM that runs periodically and reclaim memory of unreferenced objects.
- Before object is destroyed, its finalize() method is invoked (if present).
- One should override this method if object holds any resource to be released explicitly e.g. file close, database connection, etc.
-

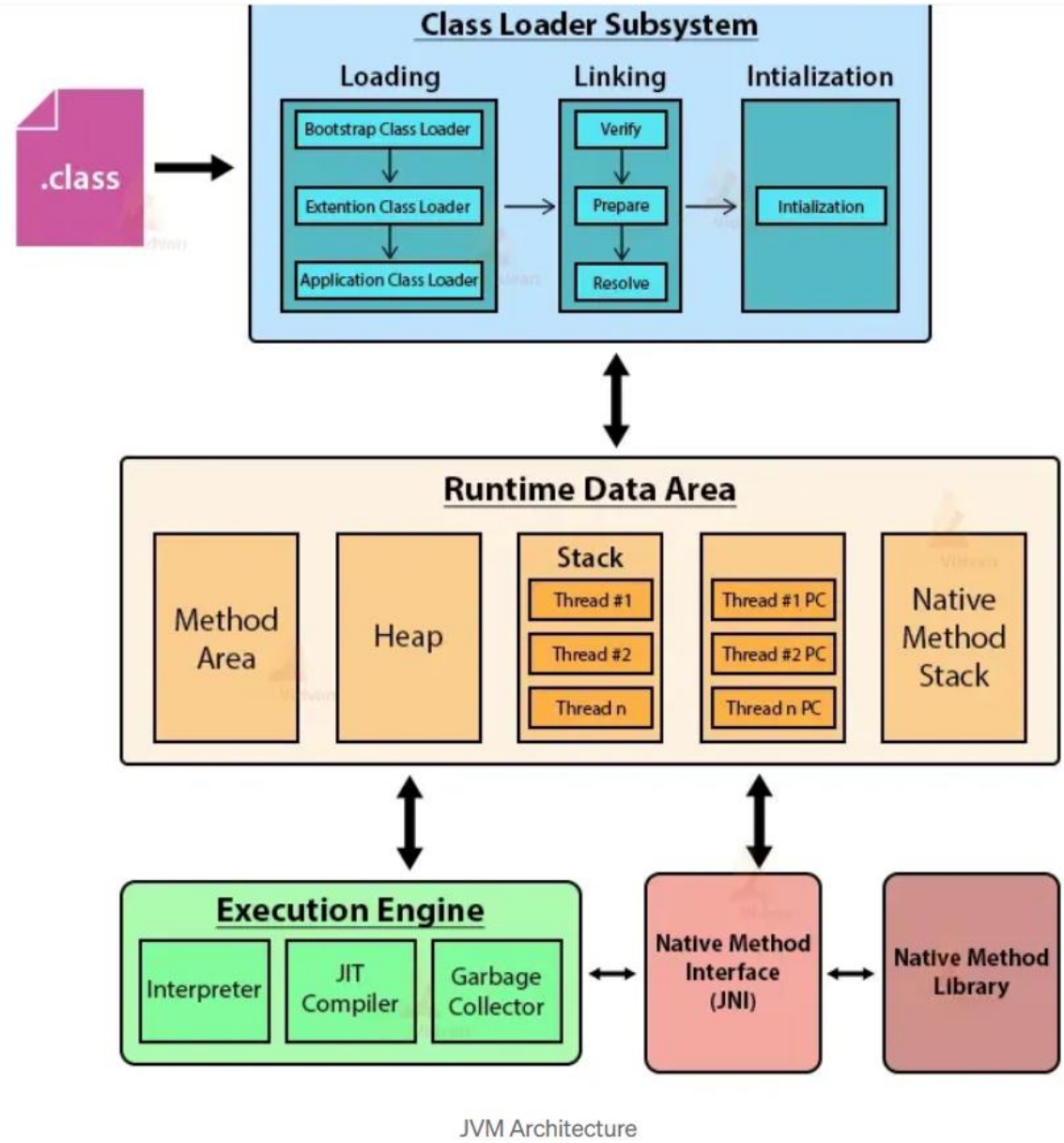
```
class MyClass {

private Connection con;

public MyClass() throws Exception {

con = DriverManager.getConnection("url", "username", "password");

}

 // ... @Override public void finalize() {

 try

if(con != null) con.close(); } catch(Exception e) { } } }

class Main { public static void method() throws Exception

 { MyClass my = new MyClass();

 my = null; System.gc();

// request GC } // ... }
```

- GC can be requested (not forced) by one of the following:-

1. .System.gc();
2. Runtime.getRuntime().gc();

JVM Architecture:-

-



JVM Architecture

Java compilation process

- Hello.java --> Java Compiler -->
  - Hello.class javac Hello.java
- Java compiler converts Java code into the Byte code.

Byte code

- Byte code is machine level instructions that can be executed by Java Virtual Machine (JVM).
  - Instruction = Op code + Operands
    - e.g. iadd op1, op2

  Each Instruction in byte-code is of 1 byte.

  .class --> JVM --> Windows (x86)
- .class --> JVM --> Linux (ARM)

- JVM converts byte-code into target machine/native code (as per architecture). .class format

- .class file contains header, byte-code, meta-data, constant-pool, etc.
- .class header contains magic number --
  - 0xCAFEBABE (first 4 bytes of .class file)
  - information of other sections

  .class file can inspected using "javap" tool.

  - terminal> javap java.lang.Object
    - Shows public and protected members
  - terminal> javap -p java.lang.Object
    - Shows private members as well
  - terminal> javap -c java.lang.Object
    - Shows byte-code of all methods
  - terminal> javap -v java.lang.Object
    - Detailed (verbose) information in .class
      - Constant pool
      - Methods & their byte-code
      - ...
- "javap" tool is part of JDK.

Executing Java program (.class)

- terminal> java Hello
- "java" is a Java Application Launcher.
- java.exe (disk) --> Loader --> (Windows
- OS) Process

  When "java" process executes, JVM (jvm.dll) gets loaded in the process.

  JVM will now find (in CLASSPATH) and execute the .class.

JVM Architecture (Overview)

- JVM = Classloader + Memory Areas + Execution Engine

Classloader sub-system

- - Load and initialize the class

Loading

- Three types of classloaders
  - Bootstrap classloader: Load Java builtin classes from jre/lib jars (e.g. rt.jar).
  - Extended classloader: Load extended classes from jre/lib/ext directory.
- Application classloader: Load classes from the application classpath.

  Reads the class from the disk and loads into JVM method (memory) area.

Linking

- Three steps: Verification, Preparation, Resolution
- Verification: Byte code verifier does verification process. Ensure that class is
- compiled by a valid compiler and not tampered.
- Preparation: Memory is allocated for static members and initialized with their default values.

  Resolution: Symbolic references in constant pool are replaced by the direct references.

Initialization

Static variables of the class are assigned with given values (field initializers).

- Execute static blocks if present.

JVM memory areas

- During execution, memory is required for byte code, objects, variables, etc.
- There are five areas: Method area, Heap area, Stack area, PC Registers, Native Method Stack area.

Method area

- Created during JVM startup.
- Shared by all threads (global).
- Class contents (for all classes) are loaded into Method area.

  Method area also holds constant pool for all loaded classes.

Heap area

- Created during JVM startup.
- Shared by all threads (global).
- All allocated objects (with new keyword) are stored in heap.

The class Metadata is stored in a java.lang.Class object (in heap) once class is loaded.

The string pool is part of Heap area.

Stack area

- Separate stack is created for each thread in JVM (when thread is created).
-
- When a method is called from the stack, a new FAR (stack frame) is created on its
- stack.

Each stack frame conatins local variable array, operand stack, and other frame data.

When method returns, the stack frame is destroyed.

PC Registers

- Separate PC register is created for each thread. It maintains address of the next instruction executed by the thread.
-

After an instruction is completed, the address in PC is auto-incremented.

Native method stack area

- Separate native method stack is created for each thread in JVM (when thread is
- created).

When a native method is called from the stack, a stack frame is created on its stack.

Execution engine

- The main comonent of JVM.
-

Execution engine executes for executing Java classes.

Interpreter

Convert byte code into machine code and execute it (instruction by instruction).

Convert byte code into machine code and execute it (instruction by instruction).

-

- Each method is interpreted by the interpreter at least once.
- If method is called frequently, interpreting it each time slow down the execution of the program.

  This limitation is overcomed by JIT (added in Java 1.1).

JIT compiler

- JIT stands for Just In Time compiler.
- Primary purpose of the JIT compiler to improve the performance.
- If a method is getting invoked multile times, the JIT compiler convert it into native code and cache it.

  If the method is called next time, its cached native code is used to speedup execution process.

Profiler

- Tracks resource (memory, threads, ...) utilization for execution.
- Part of JIT that identifies hotspots. It counts number of times any method is executing. If the number is more than a threshold value, it is considered as hotspot.

Garbage collector

- When any object is unreferenced, the GC release its memory.

JNI

- JNI acts as a bridge between Java method calls and native method implementations.

Session 11:

Packages in Java:-

- A package is a namespace that organises a set of related classes and interfaces.
- Real-Life Analogy: Think of packages like folders on your computer or departments in a company.
- Example:-

  com.sunbeam → Company name (like sunbeam)

  com.sunbeam.accounts → Department for accounts

- Java provides many prebuilt packages:

  java.lang → Automatically imported (e.g., String, Math)

  java.util → Data structures (ArrayList, Scanner, etc.)

  java.io → Input/output (File, BufferedReader)

  java.sql → Database stuff

- Build your Package:
  STEP 1: Create the package (like making a folder)
  File: Calculator.java

```
package mymath;
public class Calculator {
public int add(int a, int b) {
return a + b;
}
}
```

  STEP 2: Use the package in another file
  File: TestCalc.java

```
import mymath.Calculator;
public class TestCalc {
public static void main(String[] args) {
Calculator c = new Calculator();
System.out.println("Sum = " + c.add(5, 3));
}
}
```

-

Access Control Rules:-

- ACCESS MODIFIERS in Java:
  1. public
  2. protected
  3. default (no modifier)
  4. private

Modifier	Same Class	Same Package	Subclass (Other Pkg)	Other Classes
public	✅	✅	✅	✅
protected	✅	✅	✅	❌
default	✅	✅	❌	❌
private	✅	❌	❌	❌

Type of Inheritance:-

- Inheritance = When one class acquires (inherits) the properties and behaviours (fields & methods) of another class.
  Real-Life Analogy: Think of a child inheriting traits from their parents.

- class Parent {
  // properties & methods
  }
- class Child extends Parent {
  // inherits from Parent
  }
- Types of Inheritance in Java:
  - Single Inheritance
  - Multilevel Inheritance
  - Hierarchical Inheritance
  - Hybrid (via interfaces)
    (Note: Java does not support multiple inheritance with classes directly to avoid ambiguity.)

Inheritance Type	Structure	Real-life Analogy	Memory Trick
Single	A → B	Son inherits from Father	One parent, one child
Multilevel	A → B → C	Grandfather → Father → Son	Family Tree
Hierarchical	A → B, A → C	Father has Son & Daughter	One root, many branches

-

Inheritance Type	Structure	Real-life Analogy	Memory Trick
Hybrid (via Interface)	A, B → C	Traits from Mom + Dad	Community + Family

IS A Relationship:-

- In Java, an IS-A relationship means: "Class B IS A Class A" → i.e., Class B inherits from Class A.
- Using inheritance:
  class B extends A
- Real-Life Analogy:
  Think of a Dog and an Animal.
  A Dog IS AN Animal → Animal is the parent class (superclass), Dog is the child (subclass)
- If a subclass can say 'I am a superclass', then it's an IS-A relationship.

Inheritance & Polymorphism:-

- Definition: Inheritance is when one class (child) gets the properties and behaviours (fields and methods) of another class (parent).
- Think of a Father and Son:

  Father has: lastName, driveCar()

  Son inherits: lastName and driveCar() from Father

- When to Use: When you have an "IS-A" relationship
- Java doesn't support multiple implementation inheritance, i.e. a class cannot be inherited from multiple super-classes.
- However, Java does support multiple interface inheritance, i.e. a class can be inherited from multiple
- Definition: Polymorphism means "many forms" — one interface, many behaviours.
- Two Types:
  Compile-time Polymorphism (Method Overloading)
  Runtime Polymorphism (Method Overriding)

Session 12:

-

Compile Time Polymorphism:-

- Polymorphism means “many forms.” Compile-time polymorphism means the method that gets called is decided by the compiler, before the code runs.
- In Java: This is done through Method Overloading
- Method Overloading = multiple methods with the same name but different:
  1. Number of parameters
  2. Type of parameters
  3. Order of parameters

Runtime Polymorphism:-

- Runtime Polymorphism in Java is the ability of an object to take many forms. It occurs when a parent class reference is used to call an overridden method of a child class, and the decision about which method to execute is made at runtime.
- Syntax:

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

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

public class Test {
public static void main(String[] args) {
Parent ref = new Child(); // Upcasting
ref.show(); // Output: Child show (Resolved at runtime)
}
}
```

- Runtime Polymorphism = Method Overriding + Inheritance + Parent Reference + Child Object + Runtime Decision

•

Abstract Classes:-

- An abstract class in Java is a class that cannot be instantiated on its own and may contain abstract (incomplete) methods that must be implemented by its subclasses.
- Abstract class → used to define an abstract class
- Abstract method → a method without body (no implementation)
- An abstract class can have:

  Abstract methods (without body)

  Concrete methods (with body)

  Constructors

  Static methods

  Final methods

  Member variables

- Real-Life Analogy:

  ```
  abstract class Vehicle {
  abstract void start(); // every vehicle starts differently
  void fuel() {
  System.out.println("Filling fuel...");
  }
  }

  class Car extends Vehicle {
  void start() {
  System.out.println("Starting car with key");
  }
  }

  Vehicle v = new Car();
  ```

Final:-

- The final keyword in Java is a non-access modifier used for:

-

final variables (constant values)

final methods (cannot be overridden)

final classes (cannot be inherited)

Use	Keyword	Prevents
Constant value	final	Reassignment of value
Method lock	final	Overriding
Class lock	final	Inheritance

- Final Variables (Constants)

  Definition: A final variable can be assigned only once.

- Final Methods

  Definition: A final method cannot be overridden in subclasses.

- Final Classes

  Definition: A final class cannot be extended (inherited).

- Final with Reference Variables

  final StringBuilder sb = new StringBuilder("Hello");

Interface:-

- An interface in Java is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. The methods in interfaces are abstract by default (i.e., without body).
- SYNTAX

  interface InterfaceName {
  // constant fields
  // abstract methods
  // default / static methods (Java 8+)
  }

Example:

-

```
interface Drawable {
void draw(); // abstract method
}

class Circle implements Drawable {
public void draw() {
System.out.println("Drawing Circle");
}
}
```

- INTERFACE VS ABSTRACT CLASS

Feature	Interface	Abstract Class
Inheritance	Multiple allowed	Single only
Constructors	Not allowed	Allowed
Access Modifiers	public only	Any (private, protected…)
Fields	public static final only	Can be any type
Method Type	Only abstract/default	Abstract and concrete

Session 13 & 14:

Inner Class:-

- A **class defined inside another class**.
- Helps logically group classes and increases encapsulation.
- Can access the members (even private) of the outer class.

Types of Inner Classes

1. **Non-Static Inner Class (Member Inner Class)**
   Defined as a member of the outer class (without static).
   Can access all outer class members directly.

2. **Static Nested Class**
   Declared with `static` keyword inside outer class.
   Can access only static members of outer class.
3. **Local Inner Class**

-

Defined inside a method or block.
Scope limited to that method.

4. **Anonymous Inner Class**
   No class name, defined and instantiated at the same time.
   Usually used to override methods of a class or implement interface.

Inner Class Type	Can Access Outer Class	Requires Outer Class Instance	Static Allowed	Scope
Non-Static Inner	Yes (all members)	Yes	No	Member level
Static Nested	Only static members	No	Yes	Member level
Local Inner	Yes (final locals too)	N/A (method scope)	No	Inside method/block
Anonymous Inner	Yes	N/A	No	Instantiated inline

Exception Handling:-

- Exception Handling is a mechanism to **handle runtime errors** so that the normal flow of the application can be maintained.
- Type of Error

Type	Description	Example
Compile-time	Detected during compilation	Syntax errors
Runtime	Occur while the program runs	Divide by 0, Null access

- Keywords in Java Exception Handling

Keyword	Purpose
try	Wrap code that might throw an exception
catch	Handles the exception
finally	Executes code **regardless** of exception
throw	Used to **explicitly throw** an exception
throws	Declares exception(s) a method may throw

- Exception Handling flow:

```
try {
  // Code that may cause exception
} catch (ExceptionType e) {
```

-

```
  // Code to handle exception
} finally {
  // Optional: Cleanup code (always runs)
}
```

- Common Exception in Java:

Exception Type	Description
ArithmeticException	Division by zero
NullPointerException	Accessing null object
ArrayIndexOutOfBoundsException	Invalid array index
NumberFormatException	Invalid string to number conversion
IOException	Input/Output related errors
FileNotFoundException	File not found during file access

- Checked vs Unchecked Exceptions:

Checked Exceptions	Unchecked Exceptions
Checked at compile time	Checked at runtime
Must be handled or declared	Optional handling
Example: IOException	Example: ArithmeticException

- Memory Trick:

  try-catch = “Try risky code, Catch it if it fails.”

  finally = “No matter what, I’ll run.”

  throw = “I throw a tantrum (error) myself.”

  throws = “I might throw tantrums, be ready.”

Custome Exception Classes:-

-

- A **Custom Exception** is a user-defined class that extends Exception or RuntimeException.

Use when:

- You need meaningful error messages.
- You want to handle specific application errors.

Syntax:

```
public class MyCustomException extends Exception {

   public MyCustomException(String message) {

      super(message);

   }

}
```

- Steps:

1. **Extend** Exception (checked) or RuntimeException (unchecked).
2. **Create a constructor** that calls super(message).

- **Best Practice:**

  Use **checked exceptions** for recoverable conditions (e.g., file not found).

  Use **unchecked exceptions** for programming bugs (e.g., invalid method arguments).

  Always provide a **meaningful message** in the constructor.

Session 15 & 16:

-

The java. Lang package, Object, Number, Math, System:-

- Object Class:

Method	Description
equals(Object obj)	Checks logical equality (can override for custom equality).
hashCode()	Returns hash code (used in collections like HashMap).
toString()	Returns string representation of an object.
clone()	Creates and returns a copy of the object.
getClass()	Returns runtime class info.
finalize()	Called by GC before object is removed. Deprecated in Java 9+.

- Number Class:

  java.lang.Number is the **abstract superclass** for classes like:

  - Integer
  - Double
  - Float
  - Long
  - Short
  - Byte
  - BigInteger, BigDecimal (from java.math)
  -

- Math Class:

Method	Description
abs(x)	Absolute value
max(a, b) / min(a, b)	Find max/min
pow(a, b)	a^b
sqrt(x)	Square root
round(x) / ceil(x) / floor(x)	Rounding operations
random()	Returns double in [0.0, 1.0)
log(x), sin(x), cos(x), tan(x)	Trigonometric/log functions

-

ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, HashMap, TreeMap:-

ArrayList:-

- ArrayList is a **resizable array** in Java.
- Part of **java.util** package.
- Allows **dynamic growth** and **shrinkage** of array size.
- Implements the **List interface**.
- Syntax:
  import java.util.ArrayList;

  ArrayList<String> list = new ArrayList<>();

- Common Methods:-

Method	Description
add(E e)	Adds element to the end
add(int index, E e)	Inserts at specific index
get(int index)	Gets element at index
set(int index, E e)	Updates value at index
remove(int index)	Removes element at index
remove(Object o)	Removes first occurrence of object
size()	Returns number of elements
clear()	Removes all elements
contains(Object o)	Checks if element exists
isEmpty()	Checks if list is empty

LinkedList:-

- A **Linked List** is a linear data structure where elements (called **nodes**) are linked using pointers.

- Each node contains:
  Data (the actual value)
  Reference (or pointer) to the next node in the sequence.

-

Unlike arrays, linked lists **do not store elements in contiguous memory locations**.

Use of Linked List:

- When you need **frequent insertions/deletions** at the start or middle.
- When size is dynamic and unknown.
- When you don't need random access by index (since get(index) is slow).

Vector:-

 Type: List (implements List interface)

 Features:

- Resizable array, similar to ArrayList.
- **Synchronized** (thread-safe) — slower than ArrayList in single-threaded.
- Maintains insertion order.
- Allows duplicates and null elements.

 Use case: When you need thread-safe dynamic arrays.

 Key methods: add(), remove(), get(), size()

HashSet:-

- **Type:** Set (implements Set interface)
- **Features:**
  - Backed by a HashMap.
  - Stores **unique elements only** — no duplicates allowed.
  - Does **not maintain order** (insertion order not guaranteed).
  - Allows **one null** element.
  - Fast operations — O(1) average for add, remove, contains.
- **Use case:** When you want a collection of unique elements without caring about order.
- **Key methods:** add(), remove(), contains()

LinkedHash Set:-

- **Type:** Set (implements Set interface)
- **Features:**
  - Like HashSet, but **maintains insertion order**.

-

    - Backed by a hash table + linked list.
    - Unique elements only, allows one null.
    - Slightly slower than HashSet due to order maintenance.
- **Use case:** When you want unique elements and order preservation.
- **Key methods:** Same as HashSet.

Tree Set:-

- **Type:** Set (implements SortedSet interface)
- **Features:**
    - Stores **unique elements only**.
    - Maintains **sorted order** (natural ordering or via Comparator).
    - Backed by a **Red-Black tree** (self-balancing binary search tree).
    - Does **not allow null** elements (throws NullPointerException).
    - Operations like add, remove, contains are O(log n).
- **Use case:** When you want unique, sorted elements.
- **Key methods:** first(), last(), ceiling(), floor()

Hash Map:-

- **Type:** Map (implements Map interface)
- **Features:**
    - Stores key-value pairs.
    - Keys are unique; values can be duplicate.
    - Uses hashing for fast lookup (O(1) average).
    - Does **not maintain order**.
    - Allows **one null key** and multiple null values.
    - Not synchronized (not thread-safe).
- **Use case:** When you want fast key-value mapping without order.
- **Key methods:** put(), get(), remove(), containsKey()

Tree Map:-

- **Type:** Map (implements SortedMap interface)
- **Features:**
    - Stores key-value pairs.
    - Keys are unique and **sorted** (natural ordering or by Comparator).
    - Backed by a Red-Black tree.
    - Operations like get, put, remove are O(log n).
    - Does **not allow null keys** (throws NullPointerException), but allows null values.

-

- **Use case:** When you want sorted key-value mappings.
- **Key methods:** firstKey(), lastKey(), ceilingKey(), floorKey()

Comparison Table:-

Collection	Ordered?	Sorted?	Allows Duplicates?	Allows Null?	Thread-safe?	Main Use Case
Vector	Yes (insertion)	No	Yes	Yes	Yes	Thread-safe list
HashSet	No	No	No	One null	No	Unique elements, fast ops
LinkedHashSet	Yes (insertion)	No	No	One null	No	Unique elements, ordered
TreeSet	Yes (sorted)	Yes	No	No	No	Unique elements, sorted
HashMap	No	No	Keys no, values yes	One null key, values null	No	Fast key-value pairs
TreeMap	Yes (sorted)	Yes	Keys no, values yes	No null keys, values null	No	Sorted key-value pairs

-

Session 17, 18, & 19:-

Generic Classes 7 Interfaces:-

Generics allow **classes, interfaces, and methods** to operate on **objects of various types** while providing **compile-time type safety**.

Why use Generic

- **Type Safety:** Catch errors at compile time.
- **Code Reusability:** Same class or interface works with different data types.
- **Eliminates Type Casting:** No need for explicit casting.
- **Cleaner Code:** More readable and maintainable.

Syntax:

```
class ClassName<T> {
    T obj;
    ClassName(T obj) {
        this.obj = obj;
    }
    T getObj() {
        return obj;
    }
}
```

Generic Interface:-

```
interface InterfaceName<T> {
    void method(T t);
}
```

•

WildCard Syntax:-

Wildcards are used in **Generics** to represent **unknown types**. They make code more **flexible and reusable** when dealing with generic classes or methods.

Syntax:

<?>: Unbounded

<? extends Type>: Upper Bounded

<? super Type>: Lower Bounded

Memory Trick:

Wildcard	Think of it as...	Use for...
<?>	“I don’t care which type”	Read-only access
<? extends T>	“T or children of T”	Safe reading
<? super T>	“T or parents of T”	Safe writing

Inheritance in Generics:-

Generics and inheritance in Java don't work the same way as class inheritance.

If class Dog extends Animal, then List<Dog> **does NOT** extend List<Animal>!

Ex:

class Animal {}

class Dog extends Animal {}

=>Animal a = new Dog();//works

=>List<Animal> animals = new ArrayList<Dog>(); // Compile-time error

•

Scenario	Code	Allowed?	Why?
List<Animal> = new ArrayList<Dog>()	❌	Not allowed	Generic types are invariant
List<? extends Animal>	✅ Read	Can read safely	Unknown subtype
List<? super Dog>	✅ Write	Can add Dog	Accepts Dog or superclass
<T extends Animal>	✅	Type bound	T must be subclass of Animal

List, Set & Map Functionality:-

List:-

List is an **ordered collection** (also called a **sequence**) in Java that allows **duplicate elements**.

Defined in java.util package.

It is an **interface** that extends the Collection interface.

Syntax: public interface List<E> extends Collection<E>

Common Implementation:

Class	Type	Thread-Safe	Key Features
ArrayList	Resizable array	❌	Fast random access, slow insertion/deletion
LinkedList	Doubly Linked List	❌	Fast insertion/deletion, slow access
Vector	Resizable array	✅	Legacy, synchronized version of ArrayList

•

Set:-

- A **Set** is a **collection** that **cannot contain duplicate elements**.
- It models the **mathematical set abstraction**.

Common Method to use set:

Method	Description
add(E e)	Adds element if not present
addAll(Collection c)	Adds all elements from another collection
remove(Object o)	Removes the specified element
clear()	Removes all elements
contains(Object o)	Checks if element exists
isEmpty()	Returns true if set is empty
size()	Number of elements
iterator()	Returns an iterator

Difference Between List and Set:

Feature	List	Set
Duplicates	Allowed	Not Allowed
Index Access	Yes (get(i))	No
Implementations	ArrayList, LinkedList	HashSet, TreeSet, LinkedHashSet

Map:-

- A **Map** is an object that maps **keys to values**.
- Each **key is unique**, but values can be **duplicate**.
- **Not a part of the Collection interface**, but is present in **java.util**.

Characteristics:

Feature	Description
Stores	Key-Value pairs
Key Uniqueness	No duplicate keys allowed
Value Duplication	Allowed
Null Keys/Values	Depends on the implementation

-

Common Map Implementation:

Map Type	Ordered?	Sorted?	Nulls Allowed	Use Case
HashMap	No order	No	1 null key, many null values	Fast lookups
LinkedHashMap	Insertion order	No	Same as HashMap	Maintain order
TreeMap	Sorted by key	Yes (natural order or custom)	No null keys	Sorted map

Tips:

- Prefer ArrayList for fast access by index.
- Use LinkedList when frequent insertions/deletions.
- HashSet is fastest for basic set operations.
- TreeSet/TreeMap keeps sorted order, but slower.
- HashMap is best for fast key-based retrieval.

•