Object Oriented Programming : **OOPs concept is based on the concept of objects, which contain data (fields/attributes) and behavior (methods/functions).**

Class : Defines the structure, attributes, and behaviors. It is like a blueprint.

Object : A real instance of the class, holding actual data. It is a physical entity.

Class	Object

class ClassName
{
variables
methods
}
	ClassName ObjectName=new ClassName();

// Defining the class *(Blueprint)*
public class **Car** {
 	
// Fields (attributes)
String make;
char model;
int year;
 	
// Method to display car details
void **displayDetails()**
{
System.out.println("Car Make: " + make);
System.out.println("Car Model: " + model);
System.out.println("Car Year: " + year);
OR
System.out.prntln(make+" "+model+" "+year);
}
Output : Toyota C 2020
	// Main class to create an object
	public class Main {
	public static void main(String[] args) {
	
	// Creating an object of the Car class
	Car **myCar** = new Car();
	
	// Assigning data using object reference
	myCar.make=Toyota;
	myCar.**model**=C;
	myCar.**year**=2020;
	Input
	
	// Calling the method of the object
	myCar.displayDetails();
	Output :
	Car Make: Toyota
	Car Model: C
	Car Year: 2020

// User defined method (to directly assign data in main class)
void setCarData(**String** cMake, **char** cModel, **int** cYear)
{
make=**cMake**;
model=cModel;
year=cYear;
}
	//Assigning data using user defined method
	myCar.setCarData("Toyota", 'C',2020);
	Input
	
	//Calling the method of the object
	myCar.displayDetails();

// Constructor to initialize the object **(this)**
Car(String make, char model, int year)
{
this.make = make;
this.model = model;
this.year = year;
}
	//Creating object & Assigning data using constructor
	Car myCar = new Car("Toyota", "Corolla", 2020);

How many ways we can store data into variable ?

1) By using object reference variable

2) By using method

3) By using constructor

Methods :

Block or **group of statements** which will perform certain task.

We must **call** the method **through object**.

1) No parameters ←→ No return value

2) No parameters ←→ Returns value

3) Takes parameters ←→ No return value

4) Takes parameters ←→ Returns value

Class (**without main** method)	Class (**with main** method)

public class Greetings {	public class GreetingsMain {
	public static void main(String[] args)
	{
	Greetings gr=new Greetings(); //Object

1) No parameters ←→No return value
void m1()
{
System.out.println("Hello..");
}
	gr.m1();

2) No parameters ←→ Returns value
String m2()
{
return("Hello how are you?");
}
	String s=gr.m2();
	System.out.println(s);
	OR
	System.out.println(gr.m2());

3) Takes parameters ←→ No return value
void m3(String **name**)
{
System.out.println("Hello "+ **name**);
}
	gr.m3("John");

4) Takes parameters ←→ Returns value
String m4(String **name**)
{
return("**Hello** "+name);
}
	String s=gr.m4("David");
	System.out.println(s);
	OR
	System.out.println(gr.m4("David"));

Constructor: A constructor in Java is a special **type of method** used to initialize objects. Constructors are **automatically called** when an **object is created** using the **new** keyword.

Default Constructor

```
public class ConstructorDemo {
int x,y;
ConstructorDemo()
        {
                x=10;
                y=20;
        }
void sum()
        {
                System.out.println(x+y);
        }


public static void main(String[] args) {
ConstructorDemo cd=new ConstructorDemo();
cd.sum();

//30
```

Parameterized Constructor

```
public class ConstructorDemo {
int x,y;
ConstructorDemo(int a, int b)
        {
                x=a;
                y=b;
        }
void sum()
        {
                System.out.println(x+y);
        }


public static void main(String[] args) {
ConstructorDemo cd=new ConstructorDemo(100,200);
cd.sum();

//300
```

Method	Constructor
Method **name can be anything**	Constructor **name** should be **same as class name**
Method **may or may not return a value**	Constructor will **never return a value** (not even void)
If method is **not returning any value**, then **specify void**	We don't specify the void
Method can take parameters/arguments	Constructor can take parameters/arguments
We **have to invoke/call** methods explicitly through object	Constructor **automatically invoked** at the time of **object creation**
Used for specifying logic	Used for initializing the values of the variables

What are the four pillars of OOP?

1. **Encapsulation** – Hiding implementation details and exposing only necessary features.
2. **Inheritance** – Acquiring properties of a parent class in a child class.
3. **Polymorphism** – Same method, different behavior (Overloading & Overriding).
4. **Abstraction** – Hiding implementation details using abstract classes or interfaces.

Call by Value: When you pass a primitive type (like int, float, etc.) to a method, a copy of the value is passed. Any changes made to the parameter inside the method do not affect the original variable outside the method.

```
public class Test {

        void m1(int number)
        {
            number=number+10;
            Syso("Value in the method:"+ number);
        }
```

```
//passing copy of the variable
public class CallByValue {
        public static void main(String[] args) {

        Test test=new Test();

        int number=100;
        Syso("Before method:"+number);     //100

        test.m1(number);                   //110

        Syso("After method:"+number);      //100
        //Original number doesn't impact
}
```

Call By Reference:

Instead of value, **passing the object** reference.

By taking the reference of the object(test), we call the reference.

```
public class Test {

   int number;

// Method to modify the number field of the
Test object
   void m2(Test t) {

    t.number = t.number + 10;
// Modify the number field of the passed object

Syso("Value in the method: " + t.number);
 // Print the modified value
   }
}
```

This shows that the **number field of the Test object is modified inside the m2()** method because Java **passes** the **reference** (**not the actual object**) **to** the **method**.
The **change** is reflected in the **main method as well.**

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

// Create a Test object
      Test test = new Test();
      test.number = 100; // Initializing the number
field

// Print value before method call
  Syso("Value before method: " + test.number);

// Call the m2 method and pass the test object
      test.m2(test);

// Print the value of number after the method
call
   Syso("Value after method: " + test.number);
   }
}

//100
//110
//110
```

Polymorphism: One thing can have many forms. *(One method can have many forms i.e. different parameters (int, double etc)*

Shape - rectangle, triangle, circle etc...

Water - vapor, ice Burge

In Java, **polymorphism** can be achieved in two primary ways:

1. Compile-time Polymorphism **(Method Overloading) :** Occurs when multiple **methods** in the same class have the **same name** but **differ in the number or type of parameters.**

   class X

   {

   void **add()**

   void **add(int x, int y)**

   }

2. Runtime Polymorphism **(Method Overriding) :** Occurs when a s**ubclass** provides a **specific** implementation of a **method** that is **already defined in** its **superclass**. The method that gets executed is determined at **runtime** based on the actual object type.

   e.g. A subclass provides a new implementation for an **inherited** method.

   class Animal {

   void **makeSound()** {

   System.out.println("Animal makes a sound");

   }

   }

   // Child class (Overriding the method)

   class Dog **extends** Animal {

   @Override

   void **makeSound()** {

   System.out.println("Dog barks");

   }

   }

Method Overloading: Defining multiple methods in the **same class** with the **same name** but different parameters.

Normal Class, Method creation	*Main Class, Object creation, Method Call*

class **Calculator** {
 	
// Declare the variables outside the method
int a = 10; // Instance variable a
int b = 20; // Instance variable b
 	
// Overloaded method with **void** return type
void add()
{
int sum = a + b;
// Use the instance variables a and b
Syso("Sum of " +a+ " and " +b+ ": " +sum);
// Prints the sum directly inside the method
}
 	
// Overloaded method to add three integers
int add(int a, int b, int c)
{
return a + b + c;
}
 	
// Overloaded method to add two double values
double add(double a, double b)
{
return a + b;
}
}
	public static void **main**(String[] args) {
	
	Calculator calc = new Calculator();
	
	// Calls the method that prints the result directly inside it
	calc.add();
	// add() method uses instance variables a and b
	
	//Sum of 10 and 20: 30
	
	Syso(calc.add(2, 3, 4));
	// Calls add(int, int, int) and prints the result
	
	//9
	
	Syso(calc.add(2.5, 3.5));
	// Calls add(double, double) and prints the result
	}
	}
	
	//6.0

Constructor Overloading: Defining multiple constructors in the **same class** with the **same name** but different parameters.

```
public class Box {

double width, height, depth;
Box()                          //1st Constructor
      {
            width=0;
            height=0;
            depth=0;
            OR
            width=height=depth=0;
      }

Box(double w, double h, double d)    //2nd
      {
            width=w;
            height=h;
            depth=d;
      }

Box(double len)                      //3rd
      {
            width=height=depth=len;
      }

double volume()
//normal method for calculation/output
      {
            return (width*height*depth);
      }
}
//w, h, d, len are variables
```

```
public class BoxMain {

      public static void main(String[] args)
{
      Box b=new Box();                  //1

      Box b=new Box(5.0,5.5,5.7);       //2

      Box b=new Box(10.5);              //3
//Created 3 objects to call 3 constructors

      Syso(b.volume());
//Calling normal method for output
      }
}
```

Day12 1:25

Can we **pass parameters to main method?** **Yes**

Can we **overload main method?** **Yes**

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

this Keyword:

When a constructor or method parameter has the same name as an instance variable, this is used to differentiate between them.

OR

If using **same name** to **class variables** and **local variables**, then this keyword is used to **differentiate** between them. (*this keyword always **refers to the class***)

```
public class ThisKeyword {


        int x, y;
        // class variables/ instance variables


Example for method:
        void setData(int x, int y)
//a,b are the local variables(if taken instead x ,y)
        {
                this.x=x;
                this.y=y;
        }
OR
Example for constructor:
        ThisKeyword(int x, int y)
        {
                this.x=x;
                this.y=y;
        }
        void display()
        {
        System.out.println(x+"   "+y);
        }
```

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



//Object creation, methods to assign values and print
        ThisKeyword th=new ThisKeyword();
        th.setData(10,20);
        th.display();



OR
        ThisKeyword th=new ThisKeyword(10,20);
        th.display();


}
```

Types of variables:

- Class variables/Instance variables
- Local variables

Encapsulation: **Data hiding by wrapping variables & methods in a single unit (class).**

Use: If you want to provide some kind of security to the class variables

1) **All variables** should be **private**
2) **For every variable** there should be **2 methods (get & set)**
3) **Variables** can **be operated** only **through methods**

```
public class Account {

private int accno;
private String name;
private double amount;

public int getAccno() {
return accno;
        }
public void setAccno(int accno) {
this.accno = accno;
        }
public String getName() {
return name;
        }
public void setName(String name) {
this.name = name;
        }
public double getAmount() {
return amount;
        }
public void setAmount(double amount) {
this.amount = amount;
        }

}
```

NOTE: Every **getter** should return the value instead of only printing

```
public class AccountMain {

public static void main(String[] args) {

Account acc=new Account();

acc.setAccno(10101);

acc.setName("John");

acc.setAmount(12552.535);

System.out.println(acc.getAccno());
System.out.println(acc.getName());
System.out.println(acc.getAmount())
        }
}
```

Generate **Setters and Getters :**

NOTE: Instead of creating it manually -

Go to Source > Generate getters and setters > Select variable to generate getters and setters > Generate

Key Features of Encapsulation:

1. **Data Hiding:** Internal details of a class are hidden from the outside world. Access to them is controlled using **access modifiers.**
    - **Private (private)**: Accessible only within the class.
    - **Protected (protected)**: Accessible within the class and its subclasses.
    - **Public (public)**: Accessible from anywhere.
2. **Getter and Setter Methods:** Instead of directly accessing class variables, encapsulation promotes using getter and setter methods to read and modify data safely.
3. **Improves Maintainability and Flexibility:** Since data is accessed through methods, logic can be modified without affecting external code.
4. **Enhances Security:** Prevents unauthorized access and accidental modification of critical data.

System.out.println() What it is ?

System.out.println("welcome")
 	
class Test
{
static String s="welcome";
}
Test.s.lenght()
 	
class System
{
  static PrintStream out;
}
 	
System.out.print()
System.out.println()
	System : Predefined class
	
	out : PrintStream type static variable
	
	PrintSteam : Predefined Class
	
	print and **println** : Methods belongs to PrintStream class

static Keyword:

Make variable **static only if we have a common data** across multiple objects (eg. dept numbers are same). Then it will be **common across multiple object else** variables are **independent**.

Advantage: Saves memory and updating it is easy.

1) static methods can *access* **static stuff directly (without object).** *NOTE: Bcz public static void main(String[] args) {*
2) static methods can *access* **non-static stuff through object.**
3) non-static methods can *access* **everything directly.**

Class with Main method
(**call** static & non-static methods **from same class**)

```
public class StaticDemo {


static int a=10;                // static variable
int b=20;                       // non-static variable

static void m1()                // static method
{
System.out.println("this is m1 static method...");
}

void m2()                       // non-static
{
System.out.println("this is m2 non-static method...");
}

void m()                        // non-static
{
System.out.println(a);
System.out.println(b);
m1();
m2();
}

public static void main(String[] args)
{
System.out.println(a);
m1();
System.out.println(b);
//cannot access directly bcoz variable b is non-static
m2();
// we cannot access directly bcoz method m2 is non-static

// Create an object to call non static method/variable

StaticDemo sd=new StaticDemo();
System.out.println(sd.b);
sd.m2();
sd.m();
        }
}
```

Main class
(**call** static & non-static methods **from another class**)

```
public class StaticMain {
public static void main(String[] args) {
//Use reference class name – belongs to which class
System.out.println(StaticDemo.a);


StaticDemo.m1();


// Create an object to call non static method/variable

StaticDemo sd=new StaticDemo();
System.out.println(sd.b);
sd.m2();


sd.m();

        }
}
```

NOTE:
We can call static and non-static method from another class using **reference class name**.
i.e. **StaticDemo**

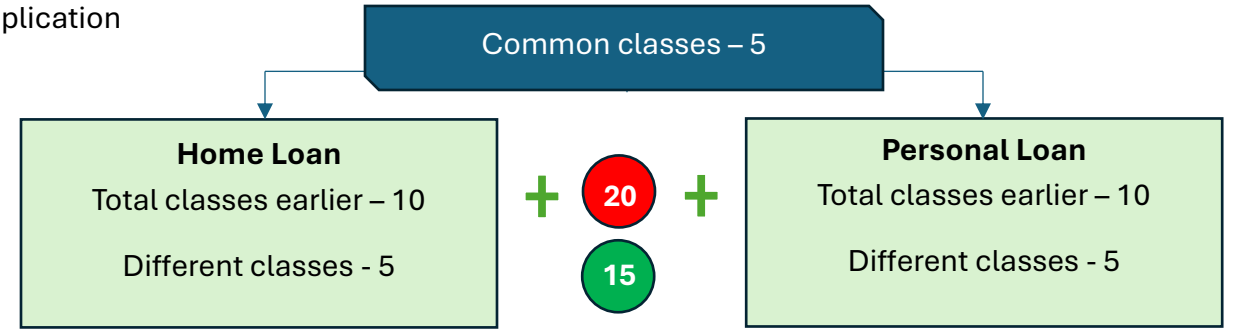
m() is a non – static method
It has static and non-static methods in it 20`, and we called m() method through the object sd.

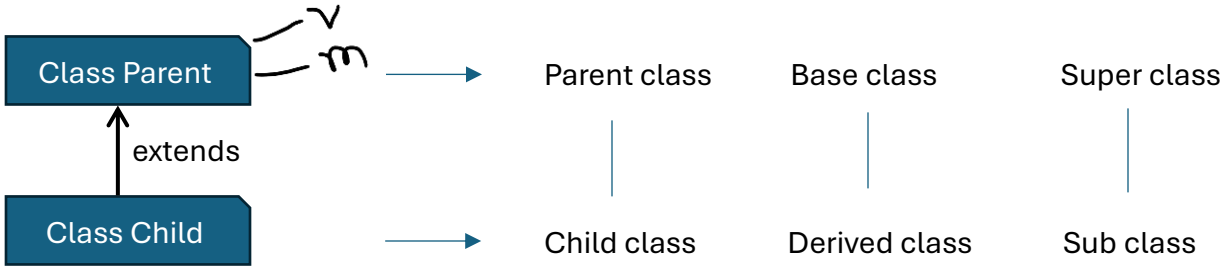
Inheritance:

Acquiring all the properties (**Variables**) & behaviors (**methods**) from **one class to another class** is called inheritance. Creating a new class based on an existing class to promote code reuse.

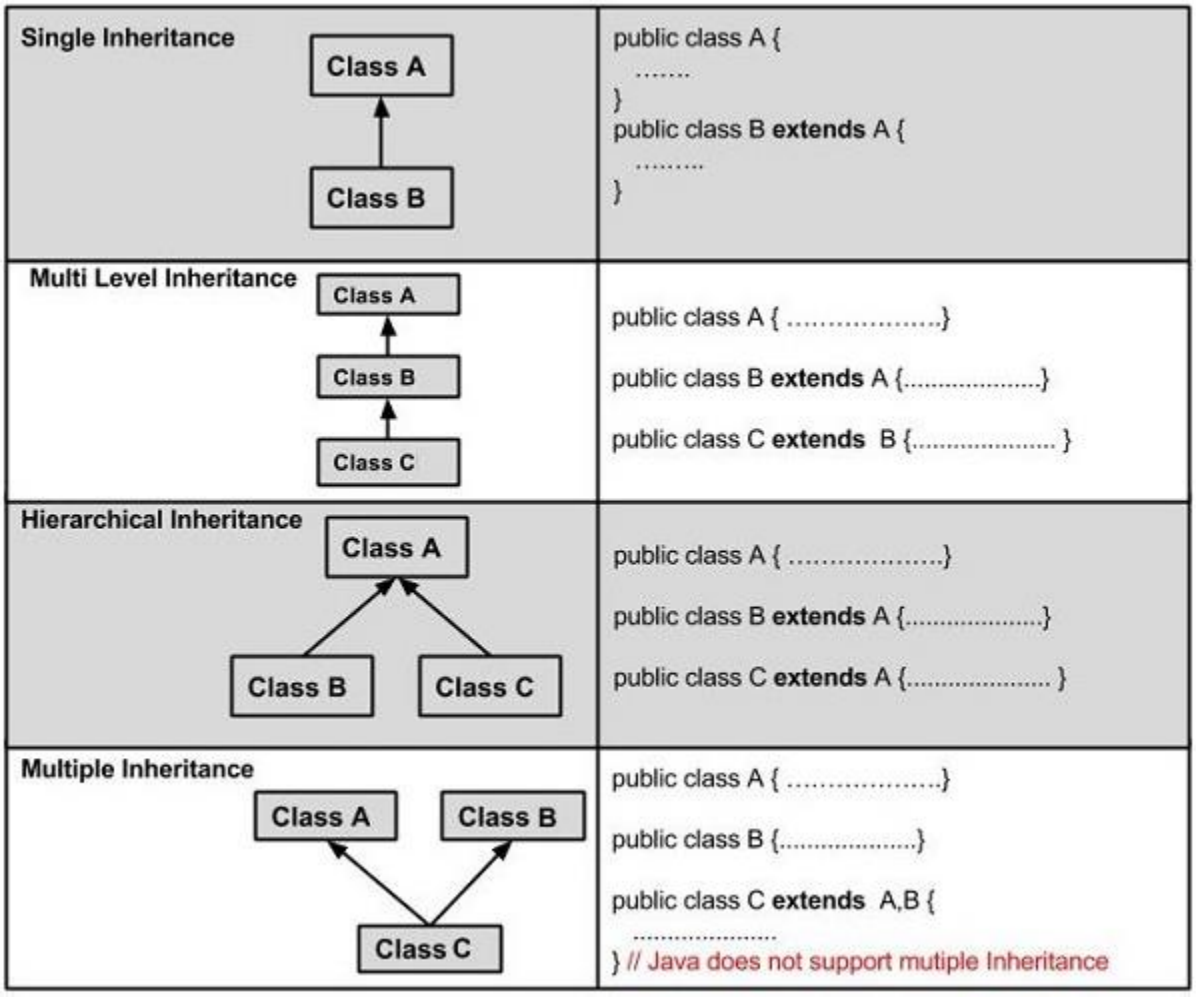
Objective:

1) Re-usability
2) Avoid duplication





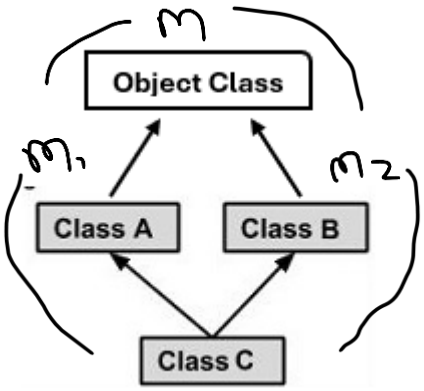
Types: class Child extends Parent



NOTE: We cannot implement **multiple inheritance** using class concept (bcz we cannot extend multiple class at a time) but with Interface concept (Interface A, B, C instead of parent class A, B, C)

Why cannot we do multiple inheritance?

Even though you have **not created** any **duplicate methods** in Parent class A and B (i.e. m1 and m2) still those classes are having duplicate methods (i.e. m) coming from **Object class** (i.e. **default parent class in java**). By default, whenever you create a class, it acquires everything from Predefined class i.e. object class in java (e.g. method m).



Predefined

Parent Class

Child class

```
class A
{
        int a;
        void display()
        {
                System.out.println(a);
        }
}

class B extends A
{
        int b;
        void show()
        {
                System.out.println(b);
        }
}
class C extends B
{
        int c;
        void print()
        {
                System.out.println(c);
        }
}
```

Single

multilevel

```
public class InheritanceTypes {
        public static void main(String[] args)
        {
                B bobj=new B();
                bobj.a=10;
                bobj.b=20;

                bobj.display();
                bobj.show();
```

Single

```
                C cobj=new C();
                cobj.a=100;
                cobj.b=200;
                cobj.c=300;

                cobj.display();
                cobj.show();
                cobj.print();

        }
}
```

multilevel

```
class Parent
{
void display(int a)
        {       System.out.println(a);          }
}

class Child1 extends Parent{
void show(int b)

class Child2 extends Parent
void print(int c)
```

Hierarchy

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

        Child1 c1=new Child1();
        c1.display(100);
        c1.show(200);

        Child2 c2=new Child2();
        c2.display(10);
        c2.print(20);
        }
```

What is ??

public static void main(String args[])

{

}

public - Access modifier *(can accessible everywhere in the project)*

static - Directly called by JVM *(without object) (static keyword must be the before method name)*

void - No returned value

String args[] - String type array *(It can accept any type of data using " i.e. "10.5" "A" "Arshad" , that's why it is string type array)*

public static void main(String a[])	Valid
public static void main(String []a)	Valid
void main(String args[]) public static	Invalid
public static void main(int a[])	Invalid
static public void main(String args[])	Valid
static void public main(String args[])	Invalid

Explain the difference between == and .equals() in Java?

- ✅ **== (Reference Comparison)** – Compares memory addresses.
- ✅ **.equals() (Content Comparison)** – Compares actual values of objects.

e.g.
```
String a = new String("Java");
String b = new String("Java");

System.out.println(a == b);          // false (Different memory locations)
System.out.println(a.equals(b));     // true (Same content)
```

Method Overloading:

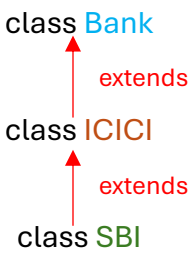
1. Possible only in **single** and **multiple classes *(inheritance)***
2. We should **change** the **signature (Parent)** of the method
3. Method names are same
4. Belongs to **polymorphism**

Method Overriding:

1. Possible only in **multiple classes *(inheritance)***
2. We should **not change** the **signature (Parent)** of the method but **body(Child)** we should **change**
3. Method names are same
4. Belongs to **inheritance**

Method Overloading vs. Method Overriding in Java:

Feature	Method Overloading	Method Overriding
Definition	Defining multiple methods in the **same class** with the **same name** but different parameters.	Defining a method in a **subclass** that has the **same signature** as a method in the **superclass**, but with a different implementation.
Where It Occurs	**Same class** (multiple methods with the same name but different parameters).	**Subclass & Superclass relationship** (subclass provides its own version of a method).
Parameters	Must be **different** (either in the number, type, or order of parameters).	Must be **exactly the same** as the superclass method.
Return Type	Can be **different**.	Must be **same** (or a covariant return type).
Access Modifiers	Can have different access levels.	Cannot have a more restrictive access level than the overridden method in the superclass.
static Methods	Can be overloaded.	Cannot be overridden (but can be **hidden** if redefined in the subclass).
final Methods	Can be overloaded.	Cannot be overridden.
Constructors	Can be overloaded (multiple constructors in the same class).	Cannot be overridden (constructors are not inherited).
Polymorphism Type	**Compile-time Polymorphism** (decision is made at compile-time).	**Runtime Polymorphism** (decision is made at runtime).
@Override Annotation	Not required.	**Required (Recommended)** to ensure proper overriding.



In above, class Bank is immediate parent class of class ICICI and class ICICI is immediate parent class of class SBI.

super Keyword:

1. super keyword is used to invoke the immediate parent class variable (else latest variable invokes)
2. super keyword is used to invoke the immediate parent class method
3. super keyword is used to invoke the immediate parent class constructor

Overriding: Defining a method in a **subclass** that has the **same signature** as a method in the **superclass**, but with a different implementation.

```
class Bank
{
        double roi()
        {
            return 0;
        }
}
class ICICI extends Bank
{
        double roi()
        {
            return 10.5;
        }
}
class SBI extends Bank
{
        double roi()
        {
            return 11.5;
        }
}

public class OverridingDemo {

public static void main(String[] args) {

        ICICI ic=new ICICI();
        System.out.println(ic.roi());       //10.5

        SBI sb=new SBI();
        System.out.println(sb.roi());       //11.5

        }
}
```

```
class ABC
{
        void m1(int a)
        {
            System.out.println(a);
        }
        void m2(int b)
        {
            System.out.println(b);
        }
}

class XYZ extends ABC
{
        void m1(int a)                  // overriding
        {
            System.out.println(a*a);
        }
        void m2(int b)                  //overriding
        {
            System.out.println(b*b);
        }
        void m2(int a, int b)           //overloading
        {
            System.out.println(a+b);
        }
}
public class OverloadingVsOverriding {

        public static void main(String[] args) {

            XYZ xyzobj=new XYZ();
            xyzobj.m1(10);
            xyzobj.m2(5);
            xyzobj.m2(10,20);
        }
}
```

Example of Method overriding, Constructor overloading:

```
public class Animal {

String color="white";

void eat()
{
        System.out.println("eating....");
}
Animal()                        //constructor
{
        System.out.println("This is Animal..");
}
Animal(String name)             //constructor
{
        System.out.println(name);
}

class Dog extends Animal

{
String color="black";

void displayColor()
{
        System.out.println(super.color);
}
void eat()
{
        //System.out.println("eating bread");
        super.eat();
}
Dog()                           //constructor
{
        super();  //Optional: invoke parent class
                    constructor
        //System.out.println("this is Dog..");
}

Dog(String name)                //constructor
{
        super(name);
}
}
```

```
public class TestSuper {

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

            d.displayColor();
            d.eat();

            or

            Dog d=new Dog("Elephant");
        }

}
```

NOTE:

- ✓ No need to use super keyword to invoke constructor from parent class.
- ✓ As the constructor invokes at the time of object creation, it will 1st invoke from parent class then child class.
- ✓ Constructor name should be same as class name that is why constructor overriding is not possible

E.g. **Why constructor overriding not possible..?**

```
public class Animal {

Animal()                        //constructor
{
        System.out.println("This is Animal..");
}

class Dog extends Animal

Animal()                        //constructor
{
        System.out.println("This is Animal..");
}

// Constructor name should be same as class name
```

final Keyword:

If applied final keyword on:

Variables - We cannot change the value of the variable (constant)
Methods - We cannot override those methods in Child classes
Class - We cannot extend the class

```
class Test
{
        final int x=100;
}

public class FinalKeyword {

        public static void main(String[] args) {

        Test t=new Test();
        t.x=200;
// we cannot change the value of x. x is final variable.
        System.out.println(t.x);
        }

}
```

```
final class Arshad
{
final void m1()
        {
        System.out.println("m1 from Test1");
        }
}

class Mujawar extends Arshad
// we cannot extend the class (Arshad is final class)
{
        void m1()
// we cannot override final methods (m1 is final method)
        {
        System.out.println("m1 from Test2");
        }
}
public class FinalKeyword2 {
        public static void main(String[] args) {
        }
}
```

Difference between final, finally, and finalize in Java ?

Keyword	Description	Usage
final	Used for constants, prevents modification.	final variable: Cannot be reassigned. final method: Cannot be overridden. final class: Cannot be inherited.
finally	Used in exception handling, always executes.	Always executes after try-catch block, even if an exception occurs.
finalize	A method used for garbage collection.	Called by the Garbage Collector before an object is destroyed.

Data abstraction:

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

Interface

1) An **interface is a blueprint of class**.
2) Interface **contains final & Static variables**.
3) Interface **contains abstract methods**. (also allowed **default methods & Static methods** from **java8** onwards)
4) An **abstract method** is a method **contains signature but not body** (Un-implemented method).
5) **Methods in interface** are **public**.
6) Interface supports the functionality of **multiple inheritance**.
7) We can **define interface** with **interface keyword**.
8) A **class extends another class**; an **interface extends another interface**, but a **class implements an interface.**
9) We **can create Object reference for Interface**, but we **cannot instantiate interface**.

Access modifiers:

public - **directly access** all variables & methods everywhere

protected - accessible **outside of package (sub classes)** through **inheritance**

default – accessible **only within the same package**

private - access **only within the same class**

continue

```
interface Shape
{
int length=10;              // final and static
int width=20;               // final and static

void circle();              // abstract method

default void square()
        {
        System.out.println("this is square -
                        default method....");
        }
static void rectangle()
        {
        System.out.println("this is
                rectangle- static method...");
        }
}
public class InterfaceDemo implements
Shape
{
public void circle()
        {
        System.out.println(" this is circle –
                        abstract method...");
        }
//Whenever you are implementing any method from the
interface into the class need to specify public access
modifier – implementation of abstract method
void triangle()
        {
        System.out.println("this is
                                triangle..");
        }
```

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

//Scenario 1

InterfaceDemo  idobj=new InterfaceDemo();
idobj.circle();             // abstract
idobj.square();             // default
Shape.rectangle();          // static
( static method directly accessed through interface name )

System.out.println(Shape.length+Shape.width);//30
//System.out.println(idobj.length+idobj.width);

idobj.triangle();           // access


//Scenario 2

Shape sh=new InterfaceDemo();
//use implemented class name at the time of obj creation

sh.circle();                // abstract method
sh.square();                // default method
//sh.rectangle();           // cannot access
Shape.rectangle();          // static method
//sh.triangle() ;           // cannot access


        }
}
```

Why interface is needed, where we are going to use.? (Development)

Initially developers aware of requirements but they don't to know how to implement them,
they will start creating requirement in the form of interfaces they keep all abstract method, once they understand how to implement then they can start creating classes.

We are going to use existing interface (Selenium WebDriver)(Testing)

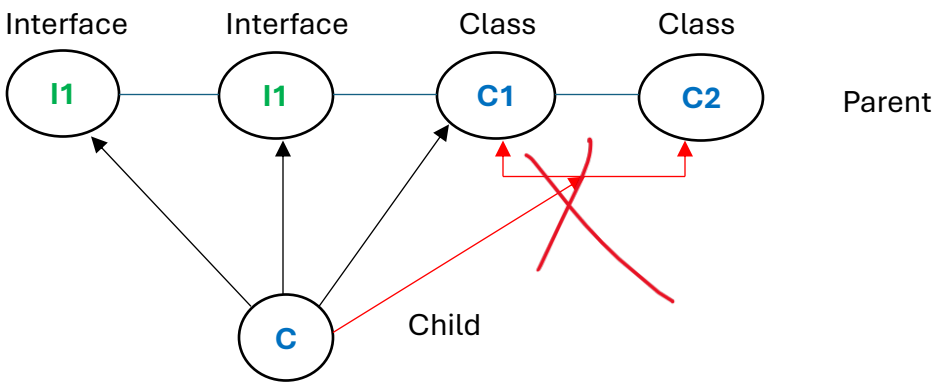
Initially they have created WebDriver which contains so many types of methods later on they have created multiple classes to implement this webdriver.
e.g. ChromBrower class, EdgeBroweser class

Multiple Inheritance:

```
public interface I1 {

        int x=100;
        void m1();

}
```

```
public interface I2 {

        int y=200;
        void m2();
}
```

```
public class MultipleInheritance implements I1,I2
{

 public void m1()
        {
                System.out.println(" this is m1...");
        }

 public void m2()
        {
                System.out.println("this is m2...");
        }

 public static void main(String[] args) {

        MultipleInheritance mi=new MultipleInheritance();
        mi.m1();
        mi.m2();

        System.out.println(mi.x);
        System.out.println(mi.y);
 }
}
```



C extends C1 implements I1, I2 //Possible
C extends C1,C2 implements I1, I2 //Not Possible (only one class is allowed as parent)

Multiple Inheritance using Interface concept

Wrapper Classes – Data Conversion

In Java, a **wrapper** refers to a **class that encapsulates a primitive data type**, allowing it to be **treated as an object**. Java provides **wrapper classes for all primitive data types** in the **java.lang** package.

- For every primitive data type there is corresponding wrapper class is available.
- **Wrapper classes convert primitive to object type and vice versa.**
- **Collection** in java allows **only object type** of **data**.

List of Wrapper Classes:

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

Why Use Wrapper Classes?

1. **Collection Framework Compatibility** – Collections (e.g., ArrayList, HashMap) only work with objects, not primitives.
2. **Utility Methods** – Wrapper classes provide useful methods for conversions, parsing, etc.
3. **Autoboxing & Unboxing** – Automatic conversion between primitive types and their wrapper objects.
   Auto boxing (Primitive ⟶ Object)
   Un-boxing (Object ⟶ Primitive)

Key Features:

- **Autoboxing**: Automatically converts primitives to wrapper objects.
- **Unboxing**: Automatically converts wrapper objects to primitives.
- **Immutable**: Wrapper objects are immutable (cannot be changed after creation).
- **Parsing & Conversion**: Methods like parseInt(), toString(), and valueOf() help in conversions.

Example:

```
int x=100;
double d=10.5;
Integer x=100;
Double d=10.5
String s="welcome";
~~String s1="welcome";~~          // cannot convert to number
String s1="150";                  // can convert to number
String s2="160";                  // can convert to number
```

Scenario 1: int, double, bool, char → String (Possible)
Scenario 2: String → int, double, bool, ~~char (Not possible)~~

public class **WrapperExample** {

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

int no = 10;

// Autoboxing: Converting primitive to Wrapper Object
Integer num = no;              // Object
Or
Integer num = 10;         // Equivalent to Integer.valueOf(10)
Double price = 99.99;
Character letter = 'A';
Boolean bool =  true;

// Unboxing: Converting Wrapper Object to primitive
int n = num;                   // Equivalent to num.intValue()
double p = price;
char l = letter;
boolean b =  bool;

// Wrapper class methods
String str = Integer.toString(100);
                              // Convert int to String
int parsedValue = Integer.parseInt("50");
                              // Convert String to int

    System.out.println("Autoboxed Integer: " + num);
    System.out.println("Unboxed int: " + n);
    System.out.println("Converted String: " + str);
    System.out.println("Parsed int: " + parsedValue);
  }
}
```

Conversion Type	Method
Widening (auto)	int → long → float → double
Narrowing (manual)	(type) value
Primitive → Object	Integer.valueOf(int)
Object → Primitive	obj.intValue()
Primitive → String	String.valueOf(int)
String → Primitive	Integer.parseInt(str)

public class **DataConvertions** {

1. Implicit (Widening) Conversion

int → double

```
int num = 100;
double d = num;              // int to double (automatic conversion)

System.out.println("Integer value: " + num);
System.out.println("Converted to double: " + d);
```

2. Explicit (Narrowing) Conversion

double → int

```
double d = 99.99;
int num = (int) d;                  // Explicit conversion (double to int)
                                    // type casting
System.out.println("Double value: " + d);
System.out.println("Converted to int: " + num);
                                    // 99 (decimal part lost)
```

3. Type Conversion using Wrapper Classes

```
int num = 50;

Integer obj = Integer.valueOf(num);  // Boxing (primitive to object)
int value = obj.intValue();          // Unboxing (object to primitive)

System.out.println("Boxed Integer: " + obj);
System.out.println("Unboxed int: " + value);
```

4. String Conversion

Primitive to String: int, double, bool, char → String

Use String.valueOf() or toString()

```
int num = 100;
String str = String.valueOf(num);
or
Integer.toString(num)
System.out.println("Converted String: " + str);

boolean bool=true;
String str=String.valueOf(bool);
System.out.println("Converted String: " + str);
```

String to Primitive: String → int, double, bool, char (not possible)

Use wrapper class methods like parseInt(), parseDouble()

```
String str = "123";
int num = Integer.parseInt(str);
System.out.println("Converted int: " + num);

String str ="10.5";
double dou = Double.parseDouble(str);
System.out.println("Converted double: " + dou);

String str = "true";
boolean bool = Boolean.parseBoolean(str);
System.out.println("Converted boolean: " + bool);
```

NOTE:

```
String s="welcome";       // cannot convert to number
String  → char            // cannot covert - not possible
```

Packages:

built-in packages - java.util, java.io, etc.
user-defined packages - Custom packages created using package keyword.
sub packages - A package inside another package.

Access modifiers:
public - **directly access** all variables & methods everywhere
protected - accessible **outside of package (sub classes)** through **inheritance**
default – accessible **only within the same package**
private - access **only within the same class**

```
package mainPack.subPack2;
import mainPack.subPack1.ClassTest1;                    // if accessing outside the package
public class ClassTest2 {

public class ClassTest2 extends ClassTest1{             // Protected example
```

Type Casting in Java - Type casting refers to converting one data type into another.

1. **Implicit (Widening) Casting** – byte → short → int → long → float → double
   Performed automatically when converting a smaller type to a larger type.
2. **Explicit (Narrowing) Casting** – double → float → long → int → short → byte
   Requires manual (type) conversion when converting a larger type to a smaller type.

int i=100;
double d=i; // **up casting**
System.out.println(d); //100.0
	double d=10.5;
	int i=(int)d; // **down casting**
	System.out.println(i); //10

Ex1:
```
Object o=new String("welcome");
StringBuffer sb=(StringBuffer) o;
```
Rule1 ✅ Rule2 ✅ Rule3 ❌

Ex2:
```
String s=new String("welcome");
StringBuffer sb=(StringBuffer) s;
```
Rule1 ❌

Ex3:
```
Object o=new String("welcome");
StringBuffer sb=(String) o;
```
Rule1 ✅ Rule2 ❌

Ex4:
```
String s=new String("welcome");
StringBuffer sb=(String) s;
```
Rule1 ✅ Rule2 ❌

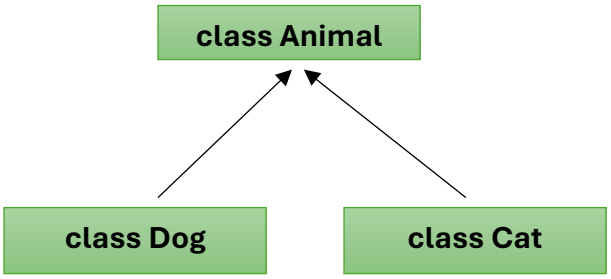
Ex5:
```
Object o=new String("welcome");
String s=(String) o;

System.out.println(s);
```
Rule1 ✅ Rules2 ✅ Rule3 ✅

A B C D

Cat ct = (Cat) an;

reference variable for cat obj — Converting **an** to Cat — **animal** type of **object/variable**

class Animal{}
class Dog extends Animal{}
class Cat extends Animal{}



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

Rule 1: Conversion is valid or not: The type of 'D' and 'C' must have some relationship (either parent to child or child to parent or same type).

Animal an=new Dog(); //Animal reference (an) is being converted into a Dog reference. A Dog object is created, but it is stored in Animal ref.
Cat ct=(Cat) an; // Rule 1 ✅

Dog dg=new Dog();
Cat ct=(Cat) dg; // Rule1 ❌

Rule2: Assignment is valid or not : 'C' must be either same or child of 'A'.

Animal an=new Dog();
Cat ct=(Cat) an; // Rule2 ✅

Animal an=new Dog();
Cat ct=(Dog) an; // Rule2 ❌

Rule3: The underlying object type of 'D' must be either same or child of 'C'.

Animal an=new Dog();
Cat ct=(Cat) an; // Rule 3 ❌

Animal an=new Dog(); // Upcasting (Dog → Animal)
Dog dg=(Dog) an; // Down casting (Animal → Dog) // Rule1 – Rule2 – Rule3 ✅

Step-by-Step Breakdown:

`Animal an = new Dog();` → Upcasting

- A Dog object is created, but it is stored in an Animal reference.
- ✅ This is safe and happens implicitly because Dog is-a Animal (inheritance).

`Dog dg = (Dog) an;` → Downcasting

- an actually holds a Dog object, so downcasting is valid.
- ✅ The explicit cast (Dog) an tells Java to treat an as a Dog object.
- Now, dg can access both Animal and Dog methods.

Exception handling:

Exception is an event which will cause program termination.

Types of Errors:

1. **Syntax Errors** – Issues in code structure, caught during compilation.
2. **Logical Errors** – Code runs but produces incorrect results.

Types of Exceptions:

1. Checked Exceptions (Compile-time Exceptions)

- Exceptions identified by the Java compiler.
- Must be handled using try-catch or declared with throws.
- **Examples:**
  - InterruptedException
  - FileNotFoundException
  - IOException

2. Unchecked Exceptions (Runtime Exceptions)

- Exceptions not checked at compile time, occurring during execution.
- Usually caused by programming mistakes.
- **Examples:**
  - ArithmeticException (e.g., division by zero)
  - NullPointerException (accessing an object reference that is null)
  - ArrayIndexOutOfBoundsException (accessing an invalid array index)

```
import java.util.Scanner;
System.out.println("program is started.....");
Scanner sc=new Scanner(System.in);
```

Example1

```
System.out.println("Enter a number:");

int num=sc.nextInt();

System.out.println(100/num);
                        // ArithmeticException
```

Example2

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

System.out.println("Enter the position(0-4):");
        int pos=sc.nextInt();

System.out.println("Enter the value:");
        int value=sc.nextInt();
        a[pos]=value;
                //ArrayIndexOutOfBoundsException

System.out.println(a[pos]);
```

Example3

```
String s="welcome";
int num=Integer.parseInt(s);
                //NumberFormatException
System.out.println(num);
```

Example4

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

```
System.out.println("program is completed.....");
```

Exception Handling using **try-catch-finally**

```
try
{
}
catch(“Exception name here and reference variable”)
{
}
finally
{
}
```

- ✓ **try** Block: The try block contains the code that might throw an exception. If an exception occurs, execution jumps to the catch block.
- ✓ **catch** Block: The catch block handles the exception. It catches specific exceptions and prevents program termination. You can also use multiple catch blocks to handle different exceptions
- ✓ **finally** Block: The finally block executes **always**, whether an exception occurs or not. It is typically used for resource cleanup (e.g., closing files or database connections).

Example Demonstrating finally

```
public class ExceptionHandlingExample {
  public static void main(String[] args) {
    try
    {
      int[] arr = {1, 2, 3};
      System.out.println(arr[5]);                    // This will throw ArrayIndexOutOfBoundsException
    }
    catch (ArrayIndexOutOfBoundsException e)
    {
      System.out.println("Array index is out of bounds: " + e.getMessage());
    }
    finally
    {
      System.out.println("This will always execute.");
    }
  }
```

Output:

```
Array index is out of bounds: Index 5 out of bounds for length 3
This will always execute.
```

Understanding the finally Block

The finally block always executes, regardless of whether an exception occurs or not.

Case	Exception Occurred?	Catch Block Executed?	Finally Block Executed?
Case 1	✅ Yes	✅ Handled	✅ Yes
Case 2	✅ Yes	❌ Not Handled	✅ Yes
Case 3	❌ No	❌ Ignored	✅ Yes

Handling Unknown Exceptions (2. Unchecked - Runtime)

If you're unsure what type of exception might occur, you have **two solutions**:

1. Multiple catch Blocks

You can **use multiple catch blocks** to handle different types of exceptions separately.

```
try
{
        int num = Integer.parseInt("ABC");                    // This will throw NumberFormatException
}
catch (ArithmeticException e)
{
        System.out.println("Arithmetic Exception: " + e.getMessage());
}
catch (NumberFormatException e)
{
        System.out.println("Number Format Exception: " + e.getMessage());
}
catch (Exception e)                                           // Catches any other exception
{
        System.out.println("Some other exception occurred: " + e.getMessage());
}
```

2. Using the Exception Class

If you don't know **what exception might occur**, you can **catch all exceptions** using the generic Exception class.

```
try
{
        int x = 10 / 0;                                       // This will throw ArithmeticException
}
catch (Exception e)                                           // Catches all exception
{
        System.out.println("Exception occurred: " + e.getMessage());
}
```

Note: Catching Exception is useful but should be used cautiously, as it hides specific exceptions.

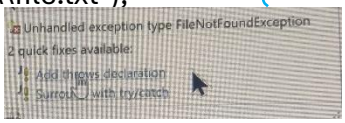
Handling Unknown Exceptions (1. Checked – Compile time)

Checked exception can be handled using **throws** and **try-catch**

```
public class CheckedExceptions {
        public static void main(String[] args) throws IOException {

                System.out.println("Program is started..");
                System.out.println("Program is progress..");
                //
                try
                {
                        FileInputStream file=new FileInputStream("C:\\file.txt");
                }
                catch (FileNotFoundException e)
                {
                        e.printStackTrace();
                }
                //
                FileInputStream file=new FileInputStream("C:\\file.txt");

                System.out.println(file.read());
                System.out.println("Program is completed..");
```

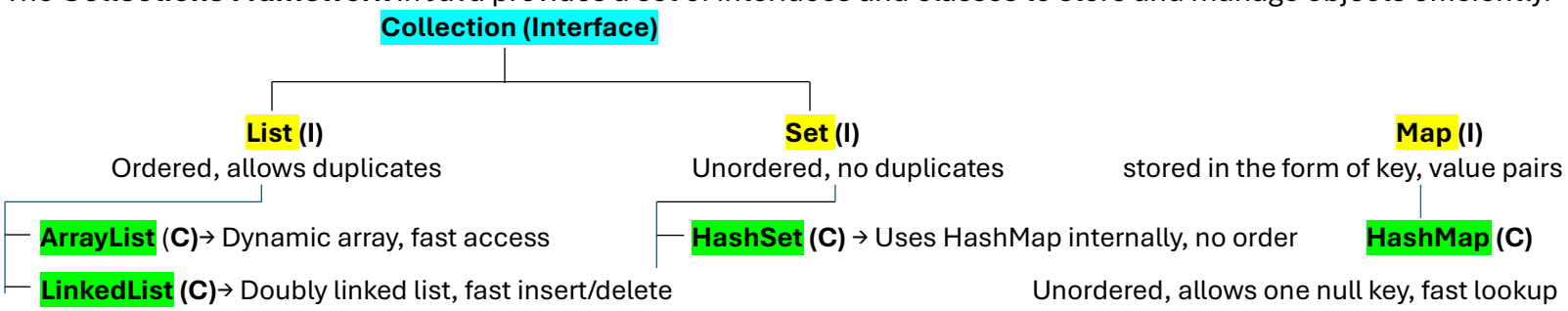
try-catch

throws



Collections:

The **Collections Framework** in Java provides a set of interfaces and classes to store and manage objects efficiently.



ArrayList:

ArrayList is a **class** in Java that **implements the List interface**, which is part of the java.util package.

An **ArrayList** in Java is a **resizable array** that is part of the **java.util** package. Unlike a normal array, which has a **fixed size**, an ArrayList **can grow and shrink dynamically**.

✅ **Key Features**:

- Heterogeneous data - allowed
- Insertion order- preserved (Index)
- Duplicate elements - allowed
- Multiple nulls - allowed

✅ **Important Methods**:

- add(), add(index, element), get(), set(), remove(), contains(), size(), isEmpty(), clear()
- Iterating using **for-loop, foreach-loop, Iterator**

HashSet:

HashSet is a **class** in Java that **implements the Set interface**, which is part of the java.util package.

✅ **Key Features**:

- Heterogeneous data - allowed
- Insertion order - Not preserved (Index not supported)
- Duplicate elements - Not Allowed
- Multiple nulls - Not allowed / only single null is allowed

HashMap:

HashMap is a **class** in Java that **implements the Map interface** and is used to store **key-value pairs**.

✅ **Key Features**:

- Heterogeneous data - allowed
- Data can be stored in the form of key, value pairs.
- Key is unique. But we can have duplicate values.
- Insertion order not preserved (Index not followed)
- Allows one null key but multiple null values

ArrayList Example :

```
import java.util.ArrayList;
import java.util.Iterator;
public class ArrayListExample {
  public static void main(String[] args) {

✅ 1. Creating an ArrayList of Strings
    ArrayList<String> myList = new ArrayList<String>();

✅ 2. Adding elements (directly as Strings)
    myList.add("Alice");                  // String
    myList.add("25");                     // Integer as String
    myList.add("3.14");                   // Double as String
    myList.add("true");                   // Boolean as String
    myList.add("A");                      // Character as String
    myList.add(null);                     // Null value
    myList.add("25");                     // Duplicate value
    myList.add("Alice");                  // Duplicate String

    System.out.println("ArrayList after adding elements: " +
myList);

✅ 3. Inserting element at a specific index
    myList.add(2, "Inserted Element");
    System.out.println("\nAfter inserting at index 2: " +
myList);

✅ 4. Accessing elements using get(index)
    System.out.println("Element at index 3: " + myList.get(3));

✅ 5. Updating an element using set(index, value)
                                  //(modify/replace/change)
    myList.set(1, "99");                  // Changing "25" to "99"
    System.out.println("After updating index 1: " + myList);

✅ 6. Removing an element by index
    myList.remove(4);
    System.out.println("After removing element at index 4: " +
myList);

✅ 7. Removing an element by value
    myList.remove("Alice");    // Removes the first occurrence
of "Alice"
    System.out.println("After removing 'Alice': " + myList);

✅ 8. Checking if an element exists
    System.out.println("Contains '3.14'? " +
myList.contains("3.14"));

✅ 9. Getting the size of the ArrayList
    System.out.println("Size of ArrayList: " + myList.size());

✅ 10. Checking if the ArrayList is empty
    System.out.println("Is the list empty? " +
myList.isEmpty());
```

✅ 11. **Iterating** through the ArrayList (3 methods)

```
    (i) Using for-loop
    System.out.println("\nIterating using for-loop:");
    for (int i = 0; i < myList.size(); i++)
    {
      System.out.println(myList.get(i));
    }

    (ii) Using enhanced for-each loop
    System.out.println("\nIterating using for-each loop:");
    for (Object x : myList)
    {
      System.out.println(x);
    }

    (iii) Using Iterator
    System.out.println("\nIterating using Iterator:");
    Iterator<String> it = myList.iterator();
    while (it.hasNext())
    {
      System.out.println(it.next());
    }

✅ 12. Clearing the ArrayList
    myList.clear();
    System.out.println("\nAfter clearing, is the list empty? " +
myList.isEmpty());
  }
}
```

Output:

2 → ArrayList after adding elements: [Alice, 25, 3.14, true, A, null, 25, Alice]
3 → After inserting at index 2: [Alice, 25, Inserted Element, 3.14, true, A, null, 25, Alice]
4 → Element at index 3: 3.14
5 → After updating index 1: [Alice, 99, Inserted Element, 3.14, true, A, null, 25, Alice]
6 → After removing element at index 4: [Alice, 99, Inserted Element, 3.14, A, null, 25, Alice]
7 → After removing 'Alice': [99, Inserted Element, 3.14, A, null, 25, Alice]
8 → Contains '3.14'? true
9 → Size of ArrayList: 7
10 → Is the list empty? false
11 → Iterating using (i) for-loop, (ii) for-each loop, (iii) iterator
99
Inserted Element
3.14
A
null
25
Alice
12 → After clearing, is the list empty? true

HashSet Example :

```
import java.util.ArrayList; import java.util.HashSet;
import java.util.Iterator; import java.util.Set;
public class HashSetDemo {
        public static void main(String[] args) {
```

✅ Declaration

```
HashSet myset=new HashSet();
//Set myset=new HashSet();
//HashSet <String>myset=new HashSet<String>();
Use above for homogeneous data
```

✅ adding elements into HashSet

```
myset.add(100);
myset.add(10.5);
myset.add("welcome");
myset.add(true);
myset.add('A');
myset.add(100);
myset.add(null);
myset.add(null);
```

✅ Printing HashSet

```
System.out.println(myset);
                        //[null, A, 100, 10.5, welcome, true]
```

✅ Size of HashSet

```
System.out.println("Size of hashset:"+ myset.size());       //6
```

✅ Removing element

```
myset.remove(10.5);              // 10.5 is value (not an index)
System.out.println("After removing:"+myset);
                                //[null, A, 100, welcome, true]
```

❌ **inserting elements at a specific position**
❌ **Direct access via index is NOT possible in HashSet**

✅ Convert `**HashSet**` **to** `**ArrayList**` for **indexed access**

```
ArrayList al=new ArrayList(myset);
System.out.println(al);          //[null, A, 100, welcome, true]
System.out.println(al.get(2));   //100
```

✅ **Read all the elements → using for..each**

```
for(Object x:myset)
{
        System.out.println(x);
}
```

→ **Using iterator**

```
Iterator <Object> it=myset.iterator();
while(it.hasNext())
{
        System.out.println(it.next());
}
```

✅ clearing all the elements in HashSet

```
myset.clear();
System.out.println(myset.isEmpty());                    //true
}
```

✅ **No Duplicates Allowed** → If you add 100 twice, only one instance remains.
✅ **Unordered Collection** → Elements are stored in random order.
✅ **Fast Operations** → add(), remove(), contains() are very fast due to hashing.
✅ **Allows null Value** → Only **one null** is allowed.
✅ **No Indexing** → You **cannot** retrieve elements using an index directly.

🔹 **Basic Operations:**
• **add(element)** → Adds an element to the HashSet (duplicates are not allowed).
• **remove(element)** → Removes the specified element from the HashSet.
• **contains(element)** → Returns true if the HashSet contains the specified element.

🔹 **Size and Checking:**
• **size()** → Returns the number of elements in the HashSet.
• **isEmpty()** → Returns true if the HashSet is empty.
• **clear()** → Removes all elements from the HashSet.

🔹 **Iterating Over HashSet:**
• Using **for-each loop** → Iterates through all elements.
• Using **Iterator** → Iterates using an Iterator.

HashMap Example:

```
import java.util.Map;
import java.util.Map.Entry;
import java.util.HashMap;
import java.util.Iterator;

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

✅ Declaration of HashMap (Key = Integer, Value = String)

```
    HashMap hm=new HashMap();
```

or

```
    Map hm=new HashMap();
```

or

```
    HashMap<Integer, String> hm = new HashMap<>();
```

✅ Adding key-value pairs

```
    hm.put(101, "John");
    hm.put(102, "Scott");
    hm.put(103, "Mary");
    hm.put(104, "Scott");
    hm.put(102, "David");  // Overwrites "Scott" with "David"
```

✅ Printing HashMap (Unordered, No duplicate keys)

```
    System.out.println(hm);
```

✅ Size of HashMap

```
    System.out.println("Size of HashMap: " + hm.size());
```

Output: 4

✅ Removing a key-value pair

```
    hm.remove(103);  // Removes key 103 and its associated
                          value
    System.out.println("After removing key 103: " + hm);
```

Output: {101=John, 102=David, 104=Scott}

✅ Accessing a value using its key

```
    System.out.println(hm.get(102));
```

Output: David

✅ Getting all keys, values, and key-value pairs

```
    System.out.println("Keys: " + hm.keySet());
```

Output: [101, 102, 104]

```
    System.out.println("Values: " + hm.values());
```

Output: [John, David, Scott]

```
    System.out.println("Entries: " + hm.entrySet());
```

Output: [101=John, 102=David, 104=Scott]

✅ **Read all the elements → using for-each loop**

```
System.out.println("Using for-each loop:");

 for (int k : hm.keySet())
    {
     System.out.println(k + "    " + hm.get(k));
    }
```

→ **using Iterator**

```
    System.out.println("\nUsing Iterator:");
Iterator<Entry<Integer, String>> it = hm.entrySet().iterator();

while (it.hasNext())
 {
  Entry<Integer, String> entry = it.next();
  System.out.println(entry.getKey() + "    " + entry.getValue());
 }
```

✅ Clearing all elements from HashMap

```
    hm.clear();
   System.out.println("Is HashMap empty? " + hm.isEmpty());
```

Output: true

```
   }
}
```

HashMap with Integer keys and String values (Both **Homogeneous**)

```
HashMap<Integer, String> hm = new HashMap<>();
```

Using Object to store different data types **(Heterogeneous)**

```
HashMap<Integer, Object> hm = new HashMap<>();
```

The **Iterator interface** allows **sequential access** to elements in a HashMap.

- **put(key, value)** → Adds or updates a key-value pair in the HashMap.
- **putIfAbsent(key, value)** → Adds the key-value pair only if the key does not already exist.
- **get(key)** →Retrieves the value associated with the given key.
- **getOrDefault(key, defaultValue)** → Returns the value for a key if it exists; otherwise, returns the provided default value.
- **remove(key)** → Removes a key-value pair using the key.
- **remove(key, value)** → Removes the key-value pair only if it matches the given value.

🔹 **Checking Elements:**

- **containsKey(key)** → Returns true if the key exists in the HashMap.
- **containsValue(value)** → Returns true if the specified value exists in the HashMap.

🔹 **Retrieving Keys, Values, and Entries:**

- **keySet()** → Returns a Set of all keys in the HashMap.
- **values()** → Returns a Collection of all values in the HashMap.
- **entrySet()** → Returns a Set of all key-value pairs (Map.Entry<K, V>).

🔹 **Size and Clearing:**

- **size()** → Returns the number of key-value pairs in the HashMap.
- **isEmpty()** → Returns true if the HashMap is empty.
- **clear()** → Removes all key-value pairs from the HashMap.

🔹 **Iterating Over HashMap:**

- Using **for-each** with keySet() → Iterates through all keys.
- Using **for-each** with entrySet() → Iterates through all key-value pairs.
- Using **Iterator** on entrySet() → Iterates using an Iterator.

Difference between ArrayList, HashSet, and HashMap:

Feature	ArrayList	HashSet	HashMap
Implements	List interface	Set interface	Map interface
Data Structure	Dynamic array	Hash table	Key-Value pairs stored in Hash table
Duplicates	✅ **Allowed**	❌ **Not Allowed**	✅ **Keys: Not Allowed** ✅ **Values: Allowed**
Insertion Order	✅ **Preserved** (Index-based)	❌ **Not Preserved**	❌ **Not Preserved** (Unordered)
Heterogeneous Data	✅ Allowed (if using ArrayList<Object>)	✅ Allowed (if using HashSet<Object>)	✅ Allowed (if using HashMap<Object, Object>)
Indexing	✅ **Allowed** (Can access via index)	❌ **Not Allowed**	❌ **Not Allowed** (Uses keys instead)
Access Time Complexity	🔹 **O(1) for get(index)** 🔹 **O(n) for contains(value)**	🔹 **O(1) for add/remove** 🔹 **O(1) for contains(value)**	🔹 **O(1) for put/get** 🔹 **O(n) for containsValue(value)**
Iteration Methods	for-loop, foreach, Iterator	foreach, Iterator	foreach, Iterator, Map.Entry
Null Values	✅ **Multiple Nulls Allowed**	✅ **One Null Allowed**	✅ **One Null Key & Multiple Null Values Allowed**
Usage	When **ordered** collection is needed	When **unique elements** are needed (won't show duplicate)	When **key-value mapping** is required
Important Methods	add(), add(index, element), get(), set(), remove(), contains(), size(), isEmpty(), clear()	add(), remove(), contains(), size(), isEmpty(), clear(), addAll(), removeAll(), retainAll()	put(), putIfAbsent(), get(), remove(), containsKey(), containsValue(), size(), isEmpty(), clear(), keySet(), values(), entrySet()

✔ contains(value) in HashSet → ✅ **O(1)**
✔ containsKey(key) in HashMap → ✅ **O(1)**
❌ containsValue(value) in HashMap → ⛔ **O(n) (Slowest)** Iterating through all elements is always O(n)

O(1) in ArrayList ?
✅ Accessing an element by index:
✅ ArrayList contains() is O(n) because it must search linearly.

O(1) in HashSet ?
✅ Checking if an element exists (contains())

O(1) in HashMap ?
✅ Getting a value by key (get())
✅ Checking if a key exists (containsKey())
✅ Inserting a key-value pair (put())