**What is oops:**

**Object-Oriented Programming (OOP**) is a way of designing and writing software where you group related tasks and data together into "objects.

Here's a simpler breakdown:

**Class:** Think of a class as a blueprint for creating objects. For example, a blueprint for a "Car" might include properties like color and speed, and methods like drive and stop.

**Object:** An object is an instance of a class. Or we can say object is a real world entity Using the "Car" blueprint, you can create an actual car with specific color and speed.

Example:

public class Car {

// Properties of the Car

String color;

int speed;

String model;

// Method to display the details of the Car

void displayDetails() {

System.out.println("Car model: " + model);

System.out.println("Car color: " + color);

System.out.println("Car speed: " + speed + " km/h");

}

// Method to accelerate the Car

void accelerate(int increment) {

speed += increment;

}

// Method to brake the Car

void brake(int decrement) {

speed -= decrement;

}

}

public class Main {

public static void main(String[] args) {

// Creating first Car object

Car car1 = new Car();

car1.color = "Red";

car1.speed = 100;

car1.model = "Toyota";

// Display details of the first Car

car1.displayDetails();

System.out.println();

// Accelerate the first Car

car1.accelerate(20);

System.out.println("After accelerating, speed of car1: " + car1.speed + " km/h");

System.out.println();

// Creating second Car object

Car car2 = new Car();

car2.color = "Blue";

car2.speed = 80;

car2.model = "Honda";

// Display details of the second Car

car2.displayDetails();

System.out.println();

// Brake the second Car

car2.brake(30);

System.out.println("After braking, speed of car2: " + car2.speed + " km/h");

}

}

**4 pillars of java:**

1. Encapsulation : It is defined as wrapping up of data under a single unit is encapsulation. Its also implements data hiding.

Why do we need encapsulation: To avoid data directly exposed to outside world.

(Using private data member we can use data hiding)

Example

public class Person {

// Private fields

private String name;

private int age;

// Public getter method for name

public String getName() {

return name;

}

· The fields name and age are private, meaning they can't be accessed directly from outside the class.

· Public getter and setter methods (getName, setName, getAge, setAge) provide controlled access to these fields.

// Public setter method for name

public void setName(String name) {

this.name = name;

}

// Public getter method for age

public int getAge() {

return age;

}

// Public setter method for age

public void setAge(int age) {

if (age > 0) {

this.age = age;

} else {

System.out.println("Age must be positive.");

}

}

}

public class Main {

public static void main(String[] args) {

// Create a Person object

Person person = new Person();

// Set the person's name and age using setter methods

person.setName("Alice");

person.setAge(30);

// Get the person's name and age using getter methods

System.out.println("Name: " + person.getName());

System.out.println("Age: " + person.getAge());

// Attempt to set a negative age

person.setAge(-5); // This will trigger a validation message

// Get the person's age again to see if it changed

System.out.println("Age after trying to set a negative value: " + person.getAge());

}

}

1. **Inheritance**: When the poperties & methods of base class are passed into derived class (or one class accquiring the properties of other class.)

Why do we need inheritance: code resuability, polymorphism, abstraction.

Inheritance represents the ****IS-A relationship**** which is also known as a parent-child relationship.

EXAMPLE :

class vehicle

{

}

Class bike extends vehicle // bike IS- A vehicle

{

}

Example

**Simple Inheritace example:**

**// Base Class (Parent Class)**

class Vehicle {

// Properties of the Vehicle

String brand;

int speed;

// Method to display the details of the Vehicle

void displayDetails() {

System.out.println("Brand: " + brand);

System.out.println("Speed: " + speed + " km/h");

}

}

// Derived Class (Child Class)

class Car extends Vehicle {

// Additional property for Car

int numberOfDoors;

// Method to display the details of the Car

void displayCarDetails() {

displayDetails(); // Call the method from the parent class

System.out.println("Number of Doors: " + numberOfDoors);

}

}

// Main Class to Test the Inheritance

public class Main {

public static void main(String[] args) {

// Create a Car object

Car myCar = new Car();

// Set properties directly

myCar.brand = "Toyota";

myCar.speed = 120;

myCar.numberOfDoors = 4;

// Display the details of the Car

myCar.displayCarDetails();

}

}

**Single Inheritance:** In single inheritance, a class inherits from one base (parent) class.

// Base Class

class Animal {

void eat() {

System.out.println("Eating...");

}

}

// Derived Class

class Dog extends Animal {

void bark() {

System.out.println("Barking...");

}

}

// Main Class to Test Single Inheritance

public class Main {

public static void main(String[] args) {

Dog dog = new Dog();

dog.eat(); // Inherited method

dog.bark(); // Own method

}

}

**Multilevel Inheritance:**  In multilevel inheritance, a class is derived from another class, which is also derived from another class.

// Base Class

class Animal {

void eat() {

System.out.println("Eating...");

}

}

// Derived Class

class Dog extends Animal {

void bark() {

System.out.println("Barking...");

}

}

// Another Derived Class

class Puppy extends Dog {

void weep() {

System.out.println("Weeping...");

}

}

// Main Class to Test Multilevel Inheritance

public class Main {

public static void main(String[] args) {

Puppy puppy = new Puppy();

puppy.eat(); // Inherited from Animal

puppy.bark(); // Inherited from Dog

puppy.weep(); // Own method

}

}

1. **Hierarchical Inheritance:** In hierarchical inheritance, multiple classes inherit from a single base class.

// Base Class

class Animal {

void eat() {

System.out.println("Eating...");

}

}

// Derived Class

class Dog extends Animal {

void bark() {

System.out.println("Barking...");

}

}

// Another Derived Class

class Cat extends Animal {

void meow() {

System.out.println("Meowing...");

}

}

// Main Class to Test Hierarchical Inheritance

public class Main {

public static void main(String[] args) {

Dog dog = new Dog();

Cat cat = new Cat();

dog.eat(); // Inherited from Animal

dog.bark(); // Own method

cat.eat(); // Inherited from Animal

cat.meow(); // Own method

}

}

**Multiple Inheritance** (Java does not support directly) ---> We can do it by using interface.

Reason : To reduce the time complexity and simplify the language.

Lets says A,B,C classes . if A and B have same method and when u call it from child their might be amibiguty.

Like example:

Class A{

Void msg(){sysout(“hello”)

}

Class B{

Void msg(){sysout(“hii”)

}

Class C extends A,B

{

PSVM()

{

C objc = new c();

Objc.msg.

}

}

//----> This will give complie time error

// First Interface

interface Printable {

void print();

}

// Second Interface

interface Showable {

void show();

}

// Class implementing both interfaces

class Demo implements Printable, Showable {

public void print() {

System.out.println("Printing...");

}

public void show() {

System.out.println("Showing...");

}

}

// Main Class to Test Multiple Inheritance using Interfaces

public class Main {

public static void main(String[] args) {

Demo demo = new Demo();

demo.print();

demo.show();

}

}

**Hybrid inheritance** is a combination of two or more types of inheritance. In Java, hybrid inheritance is not directly supported due to the restriction against multiple inheritance of classes. However, hybrid inheritance can be achieved using interfaces.

interface Animal {

void eat();

}

interface Pet {

void play();

}

class Mammal {

void sleep() {

System.out.println("Mammal is sleeping");

}

}

// Dog class demonstrates hybrid inheritance by implementing multiple interfaces and extending a class

class Dog extends Mammal implements Animal, Pet {

public void eat() {

System.out.println("Dog is eating");

}

public void play() {

System.out.println("Dog is playing");

}

}

public class HybridInheritanceExample {

public static void main(String[] args) {

Dog dog = new Dog();

dog.eat(); // From Animal interface

dog.play(); // From Pet interface

dog.sleep(); // From Mammal class

}

}

1. Polymorphism: Polymorphism in simple terms means "many forms." It's a concept in programming that allows one interface or method to be used in multiple ways.

There are two types of polymorphism

1. **Compile time or static ploymorphism:** (Trick CSL) Here we use method overloading

(Overloading -----------> means same multiple functions with same name but with different parameter)

EXAMPLE

class Calculator {

// Method to add two integers

int add(int a, int b) {

return a + b;

}

// Overloaded method to add three integers

int add(int a, int b, int c) {

return a + b + c;

}

}

public class Main {

public static void main(String[] args) {

Calculator calculator = new Calculator();

System.out.println("Sum of 2 and 3: " + calculator.add(2, 3));

System.out.println("Sum of 1, 2, and 3: " + calculator.add(1, 2, 3));

}

}

1. **Runtime or dynamic polymorphsim :** Here we use overridding method.

(Overidding------------------> means when the child class have the same method as that of parent class then it is known as overloading

Or

Child and parent class have same method but different defination

Example: class animal()

Void eat() ----> eats any thing

|

|

|

Class Deer()

Void eat() ----> eats only grass.

)

Example:

// Base Class

class Animal {

// Method to make a sound

void sound() {

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

}

}

// Derived Class

class Dog extends Animal {

// Overridden method to make a sound specific to Dog

@Override

void sound() {

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

}

}

// Derived Class

class Cat extends Animal {

// Overridden method to make a sound specific to Cat

@Override

void sound() {

System.out.println("Cat meows");

}

}

public class Main {

public static void main(String[] args) {

// Create an Animal reference and Dog object

Animal myDog = new Dog();

// Create an Animal reference and Cat object

Animal myCat = new Cat();

// Call the sound method on both objects

myDog.sound(); // Outputs "Dog barks"

myCat.sound(); // Outputs "Cat meows"

}

}

1. **Abstraction :** Hiding all the unnessary details and showing only the important part to the user is known as abstraction:

Example, you know how to use a TV remote without knowing how it works inside.

There are two ways to achieve abstraction in java

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

Abstract Classes: In Java, an abstract class is a class that cannot be instantiated on its own and can include abstract methods (methods without a body) that must be implemented by subclasses.

Example:

// Abstract Class

abstract class Animal {

// Abstract method (does not have a body)

abstract void makeSound();

// Regular method

void eat() {

System.out.println("This animal eats food.");

}

}

// Subclass (Concrete Class)

class Dog extends Animal {

// Implementing the abstract method

@Override

void makeSound() {

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

}

}

class Cat extends Animal {

// Implementing the abstract method

@Override

void makeSound() {

System.out.println("Cat meows");

}

}

public class Main {

public static void main(String[] args) {

// Creating instances of Dog and Cat

Animal myDog = new Dog();

Animal myCat = new Cat();

// Calling the abstract method

myDog.makeSound(); // Outputs "Dog barks"

myCat.makeSound(); // Outputs "Cat meows"

// Calling the regular method

myDog.eat(); // Outputs "This animal eats food."

myCat.eat(); // Outputs "This animal eats food."

}

}

**Interfaces:** An interface in Java is a reference type that can contain only abstract methods (from Java 8 onward, interfaces can also have default and static methods). Classes that implement the interface must provide implementations for the abstract methods.

Why we use interface

* It is used to achieve abstraction.
* By interface, we can support the functionality of multiple inheritance.

// Interface

interface Shape {

// Abstract method

void draw();

}

// Implementing the Interface

class Circle implements Shape {

@Override

public void draw() {

System.out.println("Drawing a Circle");

}

}

class Rectangle implements Shape {

@Override

public void draw() {

System.out.println("Drawing a Rectangle");

}

}

public class Main {

public static void main(String[] args) {

// Creating instances of Circle and Rectangle

Shape circle = new Circle();

Shape rectangle = new Rectangle();

// Calling the abstract method

circle.draw(); // Outputs "Drawing a Circle"

rectangle.draw(); // Outputs "Drawing a Rectangle"

}

}

Inheritance: This allows you to create a new class based on an existing class. For example, you can create a "ElectricCar" class that inherits all the properties and methods of the "Car" class but adds new ones like battery life.

Encapsulation: This means keeping the internal details of how an object works hidden from the outside. You interact with the object through its methods without knowing the complex details inside.

Polymorphism: This allows you to use a single method in different ways for different objects. For example, the method "start" might do different things for a "Car" and a "Bike."

Abstraction: This means focusing on the essential features of an object and ignoring the complex details. For example, you know how to use a TV remote without knowing how it works inside.

**Call by Value**

**(Java only supports call by value)**

means that when you pass a variable to a method, the method gets a copy of the variable's value. Any changes made to the variable inside the method do not affect the original variable outside the method.\

public class Main {

public static void main(String[] args) {

int num = 10; // Original piece of paper

System.out.println("Before call: " + num); // Outputs 10

changeValue(num); // Give a copy of the paper to the method

System.out.println("After call: " + num); // Still outputs 10

}

void changeValue(int number) {

number = 20; // Change the copy, not the original

}

}

**Object Slicing**

JDK (Java Development Kit): For developing and running Java programs (includes JRE and development tools).

JRE (Java Runtime Environment): For running Java programs (includes JVM and runtime libraries).

JVM (Java Virtual Machine): For executing Java bytecode (part of JRE).

**Difference between JDK, JRE, and JVMs**

**JDK (Java Development Kit)**

What It Is: A toolkit for Java developers.

What's Inside: JDK contains the JRE (Java Runtime Environment) plus tools like the compiler (javac), debugger, and other development tools.

Use Case: Needed for writing, compiling, and running Java programs.

**JRE (Java Runtime Environment)**

What It Is: The environment needed to run Java applications.

What's Inside: JRE contains the JVM (Java Virtual Machine) plus libraries and other components needed to run Java applications.

Use Case: Needed for running Java applications but not for developing them.

**JVM (Java Virtual Machine)**

What It Is: A virtual machine that runs Java bytecode.

What's Inside: The JVM is a part of the JRE and is responsible for converting Java bytecode into machine-specific code so that it can be executed by the computer.

Use Case: Executes Java programs, provides a runtime environment

# Data Types in Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. ****Primitive data types:**** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. ****Non-primitive data types:**** The non-primitive data types include [Classes](https://www.javatpoint.com/object-and-class-in-java), [Interfaces](https://www.javatpoint.com/interface-in-java), and [Arrays](https://www.javatpoint.com/array-in-java).



# Operators in Java

# **Operator** in [Java](https://www.javatpoint.com/java-tutorial) is a symbol that is used to perform operations. For example: +, -, \*, / etc.

There are many types of operators in Java which are given below:

* Unary Operator,
* Shift Operator,
* Arithmetic Operator,
* Assignment Operator.
* Relational Operator,
* Bitwise Operator,
* Logical Operator,
* Ternary Operator and

|  |  |  |
| --- | --- | --- |
| **Operator Type** | **Category** | **Precedence** |
| Unary | postfix | expr++ expr-- |
| prefix | ++expr --expr +expr -expr ~ ! |
| Arithmetic | multiplicative | \* / % |
| additive | + - |
| Shift | shift | << >> >>> |
| Relational | comparison | < > <= >= instanceof |
| equality | == != |
| Bitwise | bitwise AND | & |
| bitwise exclusive OR | ^ |
| bitwise inclusive OR | | |
| Logical | logical AND | && |
| logical OR | || |
| Ternary | ternary | ? : |
| Assignment | assignment | = += -= \*= /= %= &= ^= |= <<= >>= >>>= |

## Naming Conventions of the Different Identifiers

CLASS , INTERFACE, CONSTANTS ------> UPPER CASE

Method, variable, package -------> Lower case

Access Specifiers

**Within class** |**within package** |**outside package by subclass** |**outside package**

Private yes no no no

Default yes yes no no

Protected yes yes yes no

Public yes yes yes yes

# Constructors in Java

It is a special method which is automatically created which is automatically created at the of object creation,

-> It has the same name as that of the class name.

-> It doesnot have any return type.

-> A Java constructor cannot be abstract, static, final, and synchronized

## Types of Java constructors

There are two types of constructors in Java:

1. Default constructor.
2. Parameterized constructor

Default constructor: A default constructor is a constructor that does not take any arguments. If you don't explicitly define any constructor in your class, Java provides a default constructor automatically

Parameterized constructor : A parameterized constructor is a constructor that takes one or more arguments. It allows you to initialize an object with specific values at the time of creation.

Exampple:

public class Person {

String name;

int age;

// Default Constructor

public Person() {

name = "Unknown";

age = 0;

}

// Parameterized Constructor

public Person(String name, int age) {

this.name = name;

this.age = age;

}

public void display() {

System.out.println("Name: " + name);

System.out.println("Age: " + age);

}

public static void main(String[] args) {

// Using the Default Constructor

Person person1 = new Person();

person1.display(); // Outputs: Name: Unknown, Age: 0

// Using the Parameterized Constructor

Person person2 = new Person("Alice", 30);

person2.display(); // Outputs: Name: Alice, Age: 30

}

}

## Constructor Overloading in Java

class Student5{

int id;

String name;

int age;

//creating two arg constructor

Student5(int i,String n){

id = i;

name = n;

}

//creating three arg constructor

Student5(int i,String n,int a){

id = i;

name = n;

age=a;

}

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

public static void main(String args[]){

Student5 s1 = new Student5(111,"Karan");

Student5 s2 = new Student5(222,"Aryan",25);

s1.display();

s2.display();

}

}

# Copy Constructor

A copy constructor is a constructor used to create a new object as a copy of an existing object.

class Student {

String name;

int age;

// Parameterized constructor

Student(String name, int age) {

this.name = name;

this.age = age;

}

// Copy constructor

Student(Student student) {

this.name = student.name;

this.age = student.age;

}

void display() {

System.out.println("Name: " + name + ", Age: " + age);

}

public static void main(String[] args) {

// Creating an object using the parameterized constructor

Student student1 = new Student("Alice", 20);

student1.display(); // Outputs: Name: Alice, Age: 20

// Creating a copy of student1 using the copy constructor

Student student2 = new Student(student1);

student2.display(); // Outputs: Name: Alice, Age: 20

}

}

# Static keyword

****static keyword**** in [Java](https://www.javatpoint.com/java-tutorial) is used for memory management.The static variable can be used to refer to the common property of all objects.

Example:

Lets say we are having a class called student. Which has int rollno, String name, and static string =”kiit”.

So, here college name will be same for all the students. Irrespective of their roll number and name.

1. **class** Student{
2. **int** rollno;//instance variable
3. String name;
4. **static** String college ="ITS";//static variable
5. //constructor
6. Student(**int** r, String n){
7. rollno = r;
8. name = n;
9. }
10. //method to display the values
11. **void** display (){System.out.println(rollno+" "+name+" "+college);}
12. }
13. //Test class to show the values of objects
14. **public** **class** TestStaticVariable1{
15. **public** **static** **void** main(String args[]){
16. Student s1 = **new** Student(111,"Karan");
17. Student s2 = **new** Student(222,"Aryan");
18. //we can change the college of all objects by the single line of code
19. //Student.college="BBDIT";
20. s1.display();
21. s2.display();
22. }
23. }

# ‘this’ keyword:

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

It is also used to get rid of shadowing problem:

What is Shadowing problem: when the local variable and instant variable having the same name inside the method, then it would be a name clash.In this case jvm will give the first preference to its local variable.

Example:

Class student {

Private int age, // ----------> Instance variabe

Private string name;

Public void setdata(int a ) //----> a is local variable

{

age=a // clash

// to avoid this we use this.age = a

}

}

**Instant variable and Instant variable:**

EXAMPLE:

public class Car {

// Instance variables

String model;

int year;

// Constructor to initialize instance variables

public Car(String model, int year) {

this.model = model; // Assigning values to instance variables

this.year = year;

}

// Method to display car details

public void displayDetails() {

// Local variable

String info = "Car Model: " + model + ", Year: " + year; // Local variable

System.out.println(info); // Accessing the local variable

}

public static void main(String[] args) {

// Local variables in the main method

String myModel = "Toyota";

int myYear = 2020;

// Creating an instance of Car

Car myCar = new Car(myModel, myYear);

// Displaying the details of myCar

myCar.displayDetails();

// Outputs:

// Car Model: Toyota, Year: 2020

}

}

* **Instance Variables** are declared in a class and are part of the object's state. Each object has its own copy of these variables.
* **Local Variables** are declared inside a method, constructor, or block and are only accessible within that method, constructor, or block. They must be initialized before use.

**Aggregation:**

Aggregation is a special form of association where one class contains a reference to another class. It's a " " relationship. For example, an Employee "has an" Address. In aggregation, the contained objects (like Address) can exist independently of the container object (like Employee).

### Example of Aggregation

// Address class

class Address {

String city;

String state;

String country;

String zipcode;

// Constructor to initialize Address

public Address(String city, String state, String country, String zipcode) {

this.city = city;

this.state = state;

this.country = country;

this.zipcode = zipcode;

}

// Method to display address details

public void displayAddress() {

System.out.println("City: " + city);

System.out.println("State: " + state);

System.out.println("Country: " + country);

System.out.println("Zipcode: " + zipcode);

}

}

// Employee class

class Employee {

int id;

String name;

String emailId;

Address address; // Aggregation (Employee "has an" Address)

// Constructor to initialize Employee

public Employee(int id, String name, String emailId, Address address) {

this.id = id;

this.name = name;

this.emailId = emailId;

this.address = address;

}

// Method to display employee details

public void displayEmployee() {

System.out.println("ID: " + id);

System.out.println("Name: " + name);

System.out.println("Email ID: " + emailId);

address.displayAddress(); // Display the address details

}

public static void main(String[] args) {

// Creating an Address object

Address address = new Address("New York", "NY", "USA", "10001");

// Creating an Employee object with the Address object

Employee employee = new Employee(1, "John Doe", "john.doe@example.com", address);

// Displaying the details of the employee

employee.displayEmployee();

// Outputs:

// ID: 1

// Name: John Doe

// Email ID: john.doe@example.com

// City: New York

// State: NY

// Country: USA

// Zipcode: 10001

}

}

# Super Keyword:

The **super** keyword is a reference variable which is used to refer immediate parent class object.

**Example**

class Animal

{

void eat(){System.out.println("eating...");}

}

class Dog extends Animal

{

void eat(){System.out.println("eating bread...");}

void bark(){System.out.println("barking...");}

void work()

{

super.eat();

bark();

}

}

class TestSuper2{

public static void main(String args[]){

Dog d=new Dog();

d.work();

}}

# Final Keyword

The **final keyword** in java is used to restrict the user.Final can be:

1. variable
2. method
3. Class

Once a class,method and a variable is declared as final, it cannot be manuplated.

|  |  |  |
| --- | --- | --- |
| final variable | final method | final class |
| 1. **class** Bike9{ 2. **final** **int** speedlimit=90;//final variable 3. **void** run(){ 4. speedlimit=400; 5. } 6. **public** **static** **void** main(String args[]){ 7. Bike9 obj=**new**  Bike9(); 8. obj.run(); 9. } 10. }//end of class | 1. **class** Bike{ 2. **final** **void** run(){System.out.println("running");} 3. } 5. **class** Honda **extends** Bike{ 6. **void** run(){System.out.println("running safely with 100kmph");} 8. **public** **static** **void** main(String args[]){ 9. Honda honda= **new** Honda(); 10. honda.run(); 11. } 12. } | 1. **final** **class** Bike{} 3. **class** Honda1 **extends** Bike{ 4. **void** run(){System.out.println("running safely with 100kmph");} 6. **public** **static** **void** main(String args[]){ 7. Honda1 honda= **new** Honda1(); 8. honda.run(); 9. } 10. } |

### Dynamic binding

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

### Example of dynamic binding

class Animal {

void eat() {

System.out.println("animal is eating...");

}

}

class Dog extends Animal {

@Override

void eat() {

System.out.println("dog is eating...");

}

public static void main(String args[]) {

Animal a = new Dog();

a.eat(); // Outputs: dog is eating...

}

}

### static binding

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

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

### Example of static binding

class Dog{

private void eat(){System.out.println("dog is eating...");}

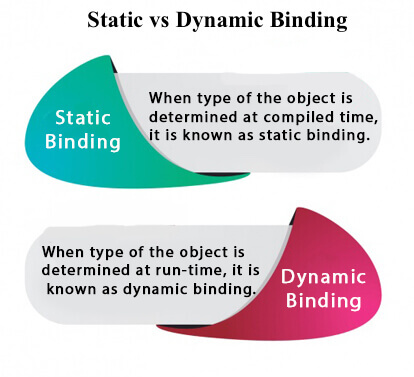
public static void main(String args[]){

Dog d1=new Dog();

d1.eat();

}

}



**Array List:** ArrayList is a part of java collection framework.

It is a collection of data elements, where the elements are stored dynamically, without any contiguous memory locations.

It does not have any fixed size unlinke array.

import java.util.ArrayList;

import java.util.Iterator;

public class ArrayListExample {

public static void main(String[] args) {

// Creating an ArrayList of Integer

ArrayList<Integer> intList = new ArrayList<>();

// Adding elements to the ArrayList

intList.add(10);

intList.add(20);

intList.add(30);

intList.add(40);

// Printing the ArrayList

System.out.println("Integer ArrayList: " + intList);

// Accessing an element from the ArrayList

int element = intList.get(2);

System.out.println("Element at index 2: " + element);

// Removing an element from the ArrayList by index

intList.remove(1);

System.out.println("After removing element at index 1: " + intList);

// Removing an element from the ArrayList by value

intList.remove(Integer.valueOf(30));

System.out.println("After removing element 30: " + intList);

// Checking if an element exists in the ArrayList

boolean containsElement = intList.contains(10);

System.out.println("ArrayList contains 10: " + containsElement);

// Getting the size of the ArrayList

int size = intList.size();

System.out.println("Size of ArrayList: " + size);

// Iterating over the elements in the ArrayList

System.out.println("Iterating over the elements:");

for (int num : intList) {

System.out.println(num);

}

// Using an iterator to iterate over the elements

Iterator<Integer> iterator = intList.iterator();

System.out.println("Using Iterator to iterate over elements:");

while (iterator.hasNext()) {

System.out.println(iterator.next());

}

// Clearing all elements from the ArrayList

intList.clear();

System.out.println("After clearing, size of ArrayList: " + intList.size());

// Creating an ArrayList of String

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

strList.add("Aditya");

strList.add("Ranjan");

strList.add("Pradhan");

strList.add("Shiv");

// Printing the ArrayList of Strings

System.out.println("String ArrayList: " + strList);

// Using forEach with Lambda expression

System.out.println("Using forEach with Lambda expression:");

strList.forEach(name -> System.out.println(name));

}

}

Example 1:

import java.util.ArrayList;

import java.util.\*;

public class A2List

{

    public static void main(String[] args)

    {

        ArrayList<Integer> al = new ArrayList<>();

        al.add(1);

        al.add(2);

        al.add(3);

        System.out.println(al);

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

        as.add("Pradhan");

        as.add("Shiv");

        as.add("Aditya");

        as.add("Ranjan");

        System.out.println(as);

        // Iterator<String> itr = as.iterator();

        // while(itr.hasNext())

        // {

        //     System.out.println(itr.next());

        // }

        System.out.println("Ans: "+as.get(1));

        as.set(3, "Prakash");

        System.out.println(as);

        Collections.sort(as);

        System.out.println(as);

    }

}

**Example: 2:**

class student

{

    int rollno;

    int name;

    public student(int rollno, int name) {

        this.rollno = rollno;

        this.name = name;

    }

    public int getRollno() {

        return rollno;

    }

    public void setRollno(int rollno) {

        this.rollno = rollno;

    }

    public int getName() {

        return name;

    }

    public void setName(int name) {

        this.name = name;

    }

    @Override

    public String toString() {

        return "student [rollno=" + rollno + ", name=" + name + "]";

    }

}

public class A2List

{

    public static void main(String[] args)

    {

        ArrayList<student> sl = new ArrayList<>();

        sl.add(new student(1, 1123));

        sl.add(new student(2, 2234));

        sl.add(new student(3, 4456));

        Iterator <student> sitr = sl.iterator();

        while(sitr.hasNext())

        {

            student s = sitr.next();

            System.out.println(s);

        }

    }

}

Example 3:

import java.util.Iterator;

import java.util.\*;

class Employee

{

    int empid;

    String empname;

    Employee(int empid, String empname)

    {

        this.empid = empid;

        this.empname = empname;

    }

    public int getEmpid()

    {

        return empid;

    }

    public void setempid(int empid)

    {

        this.empid=empid;

    }

    public String getName()

    {

        return empname;

    }

    public void setName(String empname)

    {

        this.empname=empname;

    }

    @Override

    public String toString()

    {

        return "empid: "+empid+"empname: "+empname;

    }

}

public class A2List

{

    public static void main(String[] args)

    {

        ArrayList<Employee> el = new ArrayList<>();

        el.add(new Employee(1, "Adi"));

        el.add(new Employee(2,"Shiv"));

        // TreeSet<Employee> el = new TreeSet<Employee>();

        // el.add(new Employee(1, "Adi"));

        // el.add(new Employee(2,"Shiv"));

        Iterator <Employee> eitr = el.iterator();

        while(eitr.hasNext())

        {

            Employee e = eitr.next();

            System.out.println(e);

        }

    }

}

TreeSet:

import java.util.Iterator;

import java.util.TreeSet;

class Employee implements Comparable<Employee> {

int empid;

String empname;

Employee(int empid, String empname) {

this.empid = empid;

this.empname = empname;

}

public int getEmpid() {

return empid;

}

public void setEmpid(int empid) {

this.empid = empid;

}

public String getName() {

return empname;

}

public void setName(String empname) {

this.empname = empname;

}

@Override

public String toString() {

return "empid: " + empid + ", empname: " + empname;

}

@Override

public int compareTo(Employee other) {

return Integer.compare(this.empid, other.empid);

}

}

public class A2List {

public static void main(String[] args) {

TreeSet<Employee> el = new TreeSet<>();

el.add(new Employee(1, "Adi"));

el.add(new Employee(2, "Shiv"));

Iterator<Employee> eitr = el.iterator();

while (eitr.hasNext()) {

Employee e = eitr.next();

System.out.println(e);

}

}

}

**Linked List:**

Linked list is a linear data structure which consists of a sequence of elements called as Node.Where each node consist of data and reference to the next node.

There are various types of linked list:

1. Single Linked List: each node points to next node.
2. Doubly Linked list: each node point to next node as well as the previous node.

class Node {

    int data;

    Node next;

    Node(int data) {

        this.data = data;

        this.next = null;

    }

}

class LinkedList {

    Node head;

    LinkedList() {

        head = null;

    }

    public void insert(int data) {

        Node newnNode = new Node(data);

        if (head == null) {

            head = newnNode;

        } else {

            Node currentNode = head;

            while (currentNode.next != null) {

                currentNode = currentNode.next;

            }

            currentNode.next = newnNode;

        }

    }

    public void print()

    {

        if(head==null)

        {

            System.out.println("Linked List is Empty ... ");

        }

        else

        {

            Node curr = head;

            while(curr!=null)

            {

                System.out.print(curr.data+" --> ");

                curr = curr.next;

            }

            System.out.println("null");

        }

    }

}

public class A3LinkedList {

    public static void main(String[] args)

    {

        LinkedList l = new LinkedList();

        l.insert(1);

        l.insert(2);

        l.insert(3);

        l.print();

    }

}

Example 2:

class Node

{

    int data;

    Node next;

    Node(int data)

    {

        this.data = data;

        this.next=null;

    }

}

class LinkedList

{

    Node head;

    LinkedList(int data)

    {

        head = null;

    }

    void insertatbeginning(int data)

    {

        Node newnode = new Node(data);

        newnode.next=head;

        head=newnode;

    }

    void insertatend(int data)

    {

        Node newnNode = new Node(data);

        if(head == null)

        {

            head = newnNode;

        }

        else

        {

            Node currnode = head;

            while(currnode.next!=null)

            {

                currnode = currnode.next;

            }

            currnode.next = newnNode;

        }

    }

    void insertatgivenindex(int index,int data)

    {

        if(index==0)

        {

            insertatbeginning(data);

            return;

        }

        else

        {

            Node newNode = new Node(data);

            Node currNode = head;

            for(int i=0;i<index-1;i++)

            {

                if(currNode==null)

                {

                    throw new IndexOutOfBoundsException("Index out of bounds");

                }

                currNode = currNode.next;

            }

            newNode.next = currNode.next;

            currNode.next=newNode;

        }

    }

    void display()

    {

        if(head == null)

        {

            System.out.println("Linked list is empty ... ");

        }

        else

        {

            Node curNode = head;

            while(curNode!=null)

            {

                System.out.print(curNode.data+"-->");

                curNode = curNode.next;

            }

            System.out.println("null");

        }

    }

}

public class A3LinkedListp

{

    public static void main(String[] args)

    {

        LinkedList l1 = new LinkedList(0);

        l1.insertatend(1);

        l1.insertatend(2);

        l1.display();

        l1.insertatbeginning(1111);

        l1.display();

        l1.insertatgivenindex(1, 222);

        l1.display();

    }

}

class Node

{

    int data;

    Node next;

    Node(int data)

    {

        this.data=data;

        this.next = null;

    }

}

class LinkedList

{

    Node head;

    LinkedList()

    {

        head=null;

    }

    void insertatbeginning(int data)

    {

        Node newnNode = new Node(data);

        newnNode.next = head;

        head = newnNode;

    }

    void insertatend(int data)

    {

        Node newnode = new Node(data);

        if(head == null)

        {

            head = newnode;

        }

        else

        {

            Node curNode = head;

            while(curNode.next!=null)

            {

                curNode=curNode.next;

            }

            curNode.next=head;

        }

    }

    void insertatgivenindex(int index, int data)

    {

        if(index == 0)

        {

            insertatbeginning(data);

            return;

        }

        else

        {

            Node newnNode = new Node(data);

            Node curNode = head;

            for(int i=0;i<index-1;i++)

            {

                if(curNode==null)

                {

                    throw new IndexOutOfBoundsException("Index out of bounds");

                }

                else

                {

                    curNode = curNode.next;

                }

            }

            newnNode.next = curNode.next;

            curNode.next=newnNode;

        }

    }

    void remove(int value)

    {

        //CASE:1 if empty

        if(head==null)

        {

            System.out.println("Linked List is Empty .. ");

        }

        //CASE:2 if i want to remove the head

        if(head.data == value)

        {

            head = head.next;

            return;

        }

        //CASE:3 if i wannt to remove anyother data;

        Node curNode = head;

        while(curNode.next!=null)

        {

            if(curNode.next.data == value)

            {

                curNode.next = curNode.next.next;

                return;

            }

            curNode = curNode.next;

        }

        //CASE 4: if value is not present

        System.out.println("No value "+value+" found");

    }

    void display()

    {

        if(head == null)

        {

            System.out.println("LinkedList is Emplty ... ");

        }

        else

        {

            Node currNode = head;

            while (currNode!=null)

            {

                System.out.print(currNode.data+"-->");

                currNode = currNode.next;

            }

            System.out.println("Null");

        }

    }

}

public class A3LinkedListp

{

    public static void main(String[] args)

    {

        LinkedList ll = new LinkedList();

        ll.insertatend(22);

        ll.display();

        ll.insertatbeginning(11);

        ll.display();

        ll.insertatgivenindex(1, 9999);

        ll.display();

        ll.remove(11);

        ll.display();

        ll.remove(100);

    }

}

Vector:

Vector is a type of dynamic array that can grow and shrink in size as needed. It is part of the Java Collections Framework and is used to store a list of objects

import java.util.Vector;

public class VectorExample {

public static void main(String[] args) {

// Create a Vector to hold Integer values

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

// Add elements to the Vector

vector.add(10);

vector.add(20);

vector.add(30);

// Print the Vector

System.out.println("Vector elements: " + vector);

// Access an element by index

System.out.println("Element at index 1: " + vector.get(1));

// Remove an element by index

vector.remove(1); // Removes the element at index

// Print the Vector again

System.out.println("Vector elements after removal: " + vector);

// Check if the Vector contains a specific element

System.out.println("Contains 30? " + vector.contains(30));

}

}