**Object Oriented programming**

**—------------------------------------------**

It is the another way to provide the solution for client problem

Class

—-----

Specification from group of objects. Each group representation is called class

java is Object- Oriented Programming:-

- In the real world, everything is an object like a pen, mouse, glass, etc.

- Every object has two things:

- An object knows something ( It has some properties)

- An object does something( It has a behaviour)

- Every object has a property and behaviour. It can also have multiple properties as well as multiple behaviours.

- To create an object, first, we need to create a class.

- Class act as a blueprint that contains the designing of an object.

#2

- JVM creates objects in Java.

- JVM creates an object but it requires a blueprint.

- Thus, you create a class file that class file gets compiled to create a byte code and that byte code goes to the JVM and then JVM creates an object..

Q)What are the differences between Class and Object?

—------------------------------------------------------------------------------------

Ans:

1. Class is a group of elements having common propertis and behaviours.

Object is an individual element in the group of elements having physical properties and Physical Behaviours.

2. class is virtual.

Object is Physical or real.

3. Class is the virtual encapsulation of properties and behaviours.

Object is the physical encapsulation of the properties and behaviours.

4. Class is Generalization

Object is Specialization

5. Class is a Model Or blue print for the Objects.

Object is an instance of the Class.

example of classes and objects

--------------------------------------------

**package** com.codegnan.oopprograming;

//Define a class called 'Car'

**class** Car {

// Instance variables (attributes)

String brand;

String model;

**int** year;

// Constructor method

**public** Car(String brand, String model, **int** year) {

**this**.brand = brand;

**this**.model = model;

**this**.year = year;

}

// Method to display car details

**public** **void** displayDetails() {

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

System.***out***.println("Model: " + model);

System.***out***.println("Year: " + year);

}

}

//Main class to demonstrate the usage of Car class

**public** **class** Main {

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

// Creating objects of Car class

Car car1 = **new** Car("Toyota", "Camry", 2020);

Car car2 = **new** Car("Honda", "Accord", 2019);

// Calling method to display details of car1

System.***out***.println("Car 1 Details:");

car1.displayDetails();

System.***out***.println();

// Calling method to display details of car2

System.***out***.println("Car 2 Details:");

car2.displayDetails();

}

}

METHODS

—-----------

In Java, a method is a block of code that performs a specific task and can be called (invoked) from other parts of the program. Methods are essential for organizing code, promoting reusability, and enhancing readability.

### **Considerations for Method Creation:**

· **Naming**: Use meaningful names that describe what the method does.

· **Parameters**: Choose appropriate types and numbers of parameters to pass data into methods.

· **Return Types**: Decide whether methods should return a value and choose the appropriate return type.

· **Access Modifiers**: Define whether methods are public, private, protected, or package-private based on their intended visibility and usage.

· **Static vs. Instance Methods**: Determine whether methods should be static (class-level) or instance methods (object-level).

### **Syntax to Create Methods in Java:**

//Method without parameters and without return value (void method)

**public** **void** methodName() {

// Method body

// Perform tasks here

}

//Method with parameters and without return value (void method)

**public** **void** methodNameWithParameters(**int** param1, String param2) {

// Method body using parameters

// Perform tasks with param1 and param2

}

//Method with parameters and with return value

**public** ReturnType methodNameWithReturnValue(**int** param1, String param2) {

// Method body using parameters

// Perform tasks and return a value of ReturnType

**return** returnValue;

}

### **Advantages of Methods:**

1. **Code Reusability**: Methods allow you to define functionality once and reuse it multiple times throughout your program.

2. **Modularity**: Methods promote a modular approach to programming by breaking down tasks into smaller, manageable units.

3. **Readability**: By giving meaningful names to methods, code becomes easier to understand and maintain.

4. **Encapsulation**: Methods encapsulate functionality, hiding implementation details and exposing only necessary interfaces.

5. **Debugging and Maintenance**: With methods, debugging and maintaining code becomes easier as logical units are isolated.

### **Disadvantages of Methods:**

1. **Overhead**: Creating multiple methods can add overhead in terms of memory and runtime performance, although this is usually negligible in modern systems.

2. **Abstraction Complexity**: If methods are poorly designed or overly complex, they can increase the cognitive load for developers trying to understand the code.

**public** **class** Example {

// Method without parameters and without return value

**public** **void** greet() {

System.***out***.println("Hello, World!");

}

// Method with parameters and without return value

**public** **void** printSum(**int** a, **int** b) {

**int** sum = a + b;

System.***out***.println("Sum: " + sum);

}

// Method with parameters and with return value

**public** **int** multiply(**int** a, **int** b) {

**return** a \* b;

}

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

Example example = **new** Example();

// Calling methods

example.greet();

example.printSum(5, 3);

**int** result = example.multiply(4, 6);

System.***out***.println("Multiplication result: " + result);

}

}

Constructors :

-----------------------

Object creation is not enough compulsory we should perform initialization then only the object is in a position to provide the response properly.

Whenever we are creating an object some piece of the code will be executed automatically to perform initialization of an object this piece of the code is nothing but constructor.

Hence the main objective of constructor is to perform initialization of an object.

example

----------

package com.codegnan.oopprograming;

public class Human {

String name;

int age;

public Human() {

name="malli";

age=25;

}

public Human(String name,int age) {

this.name=name;

this.age=age;

System.out.println("hello constructor");

}

public static void main(String[] args) {

Human human1 = new Human();

System.out.println("name : " + human1.name + " age : " + human1.age);

Human human2=new Human();

human2.name="codegnan";

human2.age=25;

System.out.println("name : " + human2.name + " age : " + human2.age);

Human human3=new Human();

System.out.println("name : " + human3.name + " age : " + human3.age);

Human human4=new Human("mahesh",20);

System.out.println(human4);

System.out.println("name : " + human4.name + " age : " + human4.age);

Human human5=new Human("balayya",50);

System.out.println(human5);

System.out.println("name : " + human5.name + " age : " + human5.age);

}

}

Rules to write constructors:

--------------------------------------

1. Name of the constructor and name of the class must be same.

2. Return type concept is not applicable for constructor even void also by mistake if we are declaring the return type for the constructor we won't get any compile time error and runtime error compiler simply treats it as a method.

Example:

class Test

{

void Test() //it is not a constructor and it is a method

{}

}

3. It is legal (but stupid) to have a method whose name is exactly same as class name.

4. The only applicable modifiers for the constructors are public, default, private, protected.

5. If we are using any other modifier we will get compile time error.

Example:

class Test

{

static Test()

{}

}

Output:

Modifier static not allowed here

Default constructor:

--------------------------

For every class in java including abstract classes also constructor concept is applicable.

If we are not writing at least one constructor then compiler will generate default constructor.

If we are writing at least one constructor then compiler won't generate any default constructor. Hence every class contains either compiler generated constructor (or) programmer written constructor but not both simultaneously.

example

-------

// Car.java

// Define the Car class

public class Car {

// Instance variables

String brand;

String model;

int year;

// Default constructor (provided by Java)

public Car() {

// No explicit initialization needed here, as Java initializes

// reference types (String) to null and numerical types (int) to 0 by default

}

// Method to display car details

public void displayDetails() {

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

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

System.out.println("Year: " + year);

}

// Main method to demonstrate usage

public static void main(String[] args) {

// Creating a Car object using the default constructor

Car defaultCar = new Car();

// Accessing instance variables

System.out.println("Default Car Details:");

System.out.println("Brand: " + defaultCar.brand); // Outputs: null

System.out.println("Model: " + defaultCar.model); // Outputs: null

System.out.println("Year: " + defaultCar.year); // Outputs: 0

// Since default constructor doesn't set any values, they remain null or 0

}

}

Parameterized constructors

-----------------------------

A parameterized constructor in Java is a constructor that allows you to initialize the objects of a class with specific values at the time of object creation.

Unlike the default constructor, which takes no arguments, a parameterized constructor accepts one or more parameters.

Example of Parameterized Constructor :

-------------------------------------

// Book.java

// Define the Book class

public class Book {

// Instance variables

private String title;

private String author;

private int pages;

// Parameterized constructor

public Book(String title, String author, int pages) {

this.title = title; // Initialize title

this.author = author; // Initialize author

this.pages = pages; // Initialize pages

}

// Method to display book details

public void displayDetails() {

System.out.println("Title: " + title);

System.out.println("Author: " + author);

System.out.println("Pages: " + pages);

}

// Main method to demonstrate usage

public static void main(String[] args) {

// Creating a Book object using the parameterized constructor

Book myBook = new Book("Java Programming", "John Doe", 400);

// Accessing and displaying book details using a method

myBook.displayDetails();

}

}

**copyConstructor**

A copy constructor in Java is a special type of constructor that initializes a new object as a copy of an existing object of the same class.

It accepts an object of the same class as a parameter and initializes the new object's state with the values from the existing object.

Example Program with Copy Constructor:

------------------------------------------------

// Person.java

// Define the Person class

public class Person {

// Instance variables

private String name;

private int age;

// Constructor to initialize name and age

public Person(String name, int age) {

this.name = name;

this.age = age;

}

// Copy constructor

public Person(Person anotherPerson) {

this.name = anotherPerson.name;

this.age = anotherPerson.age;

}

// Method to display person details

public void displayDetails() {

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

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

}

// Main method to demonstrate usage

public static void main(String[] args) {

// Creating a Person object using the constructor

Person person1 = new Person("Alice", 30);

// Display details of person1

System.out.println("Details of person1:");

person1.displayDetails();

System.out.println();

// Creating another Person object using the copy constructor

Person person2 = new Person(person1);

// Display details of person2

System.out.println("Details of person2 (copy of person1):");

person2.displayDetails();

}

}

**Overridethis() vs super()**

**----------------------**

this():

* Usage: this() is used to call another constructor within the same class.
* Placement: It must be the first statement in the constructor if used. It allows constructors to invoke other constructors to avoid code duplication and manage different initialization scenarios within the same class.
* Example: Useful when a class has multiple constructors with varying parameters, allowing them to call each other to ensure consistent initialization logic.

super():

* Usage: super() is used to call the constructor of the superclass (parent class) from the subclass (child class).
* Placement: It must be the first statement in the subclass constructor if used. If not explicitly written, the compiler automatically inserts a call to the no-argument constructor of the superclass.
* Example: Helps in initializing inherited properties or performing common initialization tasks defined in the superclass.

**Example program**

**--------------------**

**package** com.codegnan.oopprograming;

//Parent class (superclass) named Vehicle

**class** Vehicle {

**private** String brand;

// Constructor of Vehicle class

**public** Vehicle(String brand) {

**this**.brand = brand; // Initialize 'brand' attribute

System.***out***.println("Vehicle constructor is invoked");

}

// Method to display brand of the vehicle

**public** **void** displayBrand() {

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

}

}

//Child class (subclass) named Car extending Vehicle

**class** Car **extends** Vehicle {

**private** **int** year;

// Constructor of Car class with brand and year

**public** Car(String brand, **int** year) {

**super**(brand); // Call superclass constructor to initialize 'brand'

**this**.year = year; // Initialize 'year' attribute

System.***out***.println("Car constructor is invoked");

}

// Constructor of Car class with brand only

**public** Car(String brand) {

**this**(brand, 2020); // Call another constructor of the same class

}

// Method to display details of the car

**public** **void** displayDetails() {

**super**.displayBrand(); // Call superclass method to display brand

System.***out***.println("Year: " + year);

}

}

//Main class where program execution begins

**public** **class** Main {

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

// Creating objects of Car class using different constructors

Car myCar1 = **new** Car("Toyota", 2022);

Car myCar2 = **new** Car("Honda");

// Displaying details of cars

System.***out***.println("Car 1:");

myCar1.displayDetails();

System.***out***.println("\nCar 2:");

myCar2.displayDetails();

}

}

**this vs super**

**----------------**

### **this Keyword**

In Java, this is a reference variable that refers to the current object instance. It can be used in several contexts:

1. **To refer to instance variables**: When there is ambiguity between instance variables and local variables or method parameters, this is used to refer to the instance variable.

2. **To invoke current class method**: You can use this keyword to call another constructor from within a constructor. This is called constructor chaining.

3. **To pass as an argument in the method**: this can be passed as an argument in the method call

### **super Keyword**

In Java, super is used to refer to the superclass of the current object instance. It can be used in several contexts:

1. **To access superclass methods**: When a method is overridden in a subclass, super can be used to invoke the superclass version of the method.

**Example using super and this keywords**

**package** com.codegnan.oopprograming;

//Superclass

**class** Person {

**protected** String name;

**protected** **int** age;

**public** Person(String name, **int** age) {

**this**.name = name; // 'this' is used to refer to the instance variable 'name'

**this**.age = age; // 'this' is used to refer to the instance variable 'age'

}

**public** **void** displayInfo() {

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

}

}

//Subclass

**class** Student **extends** Person {

**private** **int** studentId;

**public** Student(String name, **int** age, **int** studentId) {

**super**(name, age); // Calls the superclass constructor using 'super'

**this**.studentId = studentId;

}

@Override

**public** **void** displayInfo() {

**super**.displayInfo(); // Calls the superclass method 'displayInfo' using 'super'

System.***out***.println("Student ID: " + **this**.studentId);

}

}

//Main class

**public** **class** Main {

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

Student student = **new** Student("Alice", 20, 12345);

student.displayInfo();

}

}

**What is the difference super(),this() constructors and super,this keywords**

**---------------------------------------------------**

| **super(), this()** | **super, this** |
| --- | --- |
| These are constructors calls. | These are keywords |
| We can use these to invoke super class & current constructors directly | We can use refers parent class and current class instance members. |
| We should use only inside constructors as first line, if we are using outside of constructor we will get compile time error. | We can use anywhere (i.e., instance area) except static area , other wise we will get compile time error . |

Example:

class Test

{

public static void main(String[] args)

{

System.out.println(super.hashCode());

}

}

Output:

Compile time error.

Non-static variable super cannot be referenced from a static context.

Constructor chaining

-------------------------------

Constructor chaining in Java refers to the concept where one constructor can call another constructor of the same class or its superclass. This allows for code reuse and helps in initializing objects with different sets of parameters or defaults without duplicating initialization logic

**Example program of constructor chaining**

**------------------------------------------**

**package** com.codegnan.oopprograming;

//Parent class

**class** Person {

**private** String name;

**private** **int** age;

**private** **char** gender;

// Constructor with all parameters

**public** Person(String name, **int** age, **char** gender) {

**this**.name = name;

**this**.age = age;

**this**.gender = gender;

System.***out***.println("Creating Person: " + name + ", Age: " + age + ", Gender: " + gender);

}

}

//Child class inheriting from Person

**class** Employee **extends** Person {

**private** **int** employeeId;

**private** String department;

// Constructor with all parameters

**public** Employee(String name, **int** age, **char** gender, **int** employeeId, String department) {

**super**(name, age, gender); // Call constructor of superclass (Person)

**this**.employeeId = employeeId;

**this**.department = department;

System.***out***.println("Creating Employee with ID: " + employeeId + ", Department: " + department);

}

// Constructor with basic parameters (defaults employeeId and department)

**public** Employee(String name, **int** age, **char** gender) {

**this**(name, age, gender, 0, "Unknown"); // Call another constructor with default values

}

// Constructor with minimal parameters (defaults age, gender, employeeId,

// department)

**public** Employee(String name) {

**this**(name, 0, 'U'); // Call another constructor with default age, gender

}

// Main method to demonstrate usage

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

// Create Employee objects using different constructors

Employee employee1 = **new** Employee("John Doe", 30, 'M', 1001, "IT");

Employee employee2 = **new** Employee("Alice Smith", 25, 'F');

Employee employee3 = **new** Employee("Bob Johnson");

}

}

### **Overloaded constructors :**

A class can contain more than one constructor and all these constructors having the same name but different arguments and hence these constructors are considered as overloaded constructors.

Example

----------------

**package** com.codegnan.oopprograming;

**public** **class** Book {

**private** String title;

**private** String author;

**private** String isbn;

**private** **double** price;

// Constructor with all parameters

**public** Book(String title, String author, String isbn, **double** price) {

**this**.title = title;

**this**.author = author;

**this**.isbn = isbn;

**this**.price = price;

}

// Constructor with title, author, and isbn (default price)

**public** Book(String title, String author, String isbn) {

**this**(title, author, isbn, 0); // Call another constructor with default price

}

// Constructor with only title and author (default isbn and price)

**public** Book(String title, String author) {

**this**(title, author, "Unknown", 0); // Call another constructor with default isbn and price

}

// Method to display information about the book

**public** **void** displayInfo() {

System.***out***.println("Title: " + title + ", Author: " + author + ", ISBN: " + isbn + ", Price: $" + price);

}

// Main method to demonstrate usage

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

// Create different books using different constructors

Book book1 = **new** Book("Java Programming", "John Doe", "978-0321995546", 39.99);

book1.displayInfo();

Book book2 = **new** Book("Clean Code", "Robert C. Martin", "978-0132350884");

book2.displayInfo();

Book book3 = **new** Book("Design Patterns", "Gang of Four");

book3.displayInfo();

}

}

**Q)What are the differences between Methods and Constructors in Java?**

**---------------------------------------------------------**

**Ans:**

**----**

1. Java Method is a set of instructions representing a particular action of an entity.

Constructor is a java feature, it can be used to provide initializations inside the Object..

2. In Java, we must provide return type to the methods. In Java,

we must not provide return type to the constructors.

3. Java methods are able to allow the access modifiers like static, final, abstract,....

Java constuctors are not allowing the access modifiers like static, final, abstract,....

4. To access a method we must create object for the respective class and we must use the generated reference variable.

In Java applications, constructors will be accessed either with 'new' keyword or with 'this' or with 'super' keyword.

Note: In Java applications, when we access constructor with new keyword then only JVM will create object at heap memory, but, if we access constructor with this keyword and with super keyword then JVM will execute the respective constructor only, JVM will not create Object.

5. In general, methods will be executed when we access the methods. If we are not accesing the method then JVM will not execute the method.

In Java applications, Constructor will be executed automatically as part of Object creation.

6. In Java applications, we can access methods at any part of the program and JVM will execute the methods as per the method call.

In Java aplications, constructors will be executed automatiically exactly at the time of creating objects , not before creating objects and not after creating objects.

7. If we use methods to provide initializations inside the objects then the provided data will be stored inside the objects as second data, not as first data.

If we use constructors to provide initializations inside the objects then the provided data will be stored inside the objects as initial data that is first data, not as second data.

8. In Java applications, Default constructors are possible , that is, compiler will add a 0-arg constructor to the class when we are not providing any constructor explicitly.

In case of Java, no default method is existed as like default constructors.

Note: Default methods are existed in java inside the interfaces from Java 8 version onwards, but they are totally in different context, not same as default constructor.

9. In Java applications, it is not mandatory to provide the same class name as method name, method names may be or may not be same as the respective class name.

In Java applicautions,it is mandatory to provide the same class name as the constructor name

**Getter and Setter methods:**

**============================**

In Java, getters and setters are methods used to access and modify the private fields (instance variables) of a class, respectively. They provide a way to encapsulate the access to class fields, allowing for better control over how the fields are accessed and modified. Getters are used to retrieve the value of a private field, while setters are used to update the value of a private field.

**Getter Methods**:

* Getter methods are used to retrieve the value of a private field (attribute). They are typically declared with the public access modifier and have a return type that matches the type of the attribute being accessed.
* The naming convention for a getter method usually starts with get followed by the name of the attribute (in camelCase).

// Setter method

public int getMyField() {

return age;

}



**Setter Methods**:

o Setter methods are used to modify the value of a private field (attribute). They are also declared with the public access modifier and have a void return type.

o The naming convention for a setter method usually starts with set followed by the name of the attribute (in camelCase), and takes a parameter that specifies the new value to be set.

**Syntax:**

// Setter method

public void setMyField(int newValue) {

myField = newValue;

}

Example-1

-----------

**package** com.codegnan.oopprograming;

**class** Human {

/\*

\* private int age=11; private String name="malli";

\*/

**private** **int** age = 13;

**private** String name = "codegnan";

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

}

**public** **class** Demo {

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

Human obj = **new** Human();

/\*

\* obj = age = 11; obj.name = "malli";

\* System.out.printrln(obj.name+"-----"+obj.age)

\*/

// System.out.println(obj.getAge() + "-----" + obj.getName());

obj.setAge(30);

obj.setName("praveen");

System.***out***.println(obj.getAge() + obj.getName());

}

}

**Example-2**

**----------**

**package** com.codegnan.oopprograming;

**public** **class** Person {

**private** String name;

**private** **int** age;

**private** **char** gender;

// Constructor

**public** Person(String name, **int** age, **char** gender) {

**this**.name = name;

**this**.age = age;

**this**.gender = gender;

}

// Getter for name

**public** String getName() {

**return** name;

}

// Setter for name

**public** **void** setName(String name) {

**this**.name = name;

}

// Getter for age

**public** **int** getAge() {

**return** age;

}

// Setter for age

**public** **void** setAge(**int** age) {

**this**.age = age;

}

// Getter for gender

**public** **char** getGender() {

**return** gender;

}

// Setter for gender

**public** **void** setGender(**char** gender) {

**this**.gender = gender;

}

// Method to display information about the person

**public** **void** displayInfo() {

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

}

// Main method to demonstrate usage

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

Person person = **new** Person("Alice", 25, 'F');

// Display initial information

person.displayInfo();

// Update information using setters

person.setName("Bob");

person.setAge(30);

person.setGender('M');

// Display updated information

person.displayInfo();

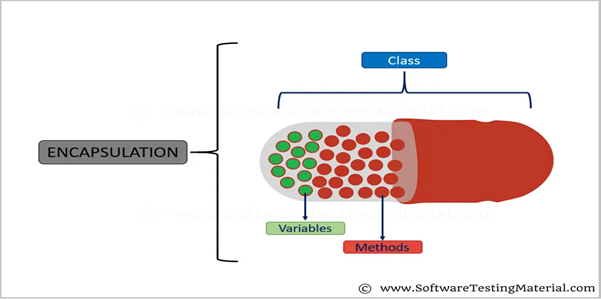
}

}

**Encapsulation**

**---------------**

Encapsulation is the bundling of data (attributes) and methods (functions that operate on the data) that manipulate the data, into a single unit called a class. It is a fundamental principle of object-oriented programming that promotes data hiding and abstraction.



EXAMPLE-1 tablet capsules example with encapsulation

------------------------------------------------------

**package** com.codegnan.oopprograming;

//Class representing a Tablet Capsule

**class** TabletCapsule {

// Private variables (attributes) of the TabletCapsule class

**private** String color; // Color of the capsule

**private** String size; // Size of the capsule

**private** **int** strength; // Strength of the capsule in milligrams

// Constructor to initialize the TabletCapsule object

**public** TabletCapsule(String color, String size, **int** strength) {

**this**.color = color;

**this**.size = size;

**this**.strength = strength;

}

// Getter method to retrieve the color of the capsule

**public** String getColor() {

**return** color;

}

// Setter method to set the color of the capsule

**public** **void** setColor(String color) {

**this**.color = color;

}

// Getter method to retrieve the size of the capsule

**public** String getSize() {

**return** size;

}

// Setter method to set the size of the capsule

**public** **void** setSize(String size) {

**this**.size = size;

}

// Getter method to retrieve the strength of the capsule

**public** **int** getStrength() {

**return** strength;

}

// Setter method to set the strength of the capsule

**public** **void** setStrength(**int** strength) {

**this**.strength = strength;

}

// Method to display information about the capsule

**public** **void** display() {

System.***out***.println("Tablet Capsule Information:");

System.***out***.println("Color: " + color);

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

System.***out***.println("Strength: " + strength + " mg");

}

}

//Main class to demonstrate encapsulation with TabletCapsule class

**public** **class** Demo {

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

// Creating an instance of TabletCapsule

TabletCapsule capsule = **new** TabletCapsule("Red", "Medium", 500);

// Displaying initial information using display method

capsule.display();

// Changing the color of the capsule using setter method

capsule.setColor("Blue");

// Changing the strength of the capsule using setter method

capsule.setStrength(750);

// Displaying updated information using display method

capsule.display();

}

}

Example-2: Simple Interest Calculator with Encapsulation

**package** com.codegnan.oopprograming;

// SimpleInterestCalculator.java

// Class representing a Simple Interest Calculator

**public** **class** SimpleInterestCalculator {

// Private variables (attributes) encapsulated within the class

**private** **double** principal; // principal amount

**private** **double** rate; // annual interest rate

**private** **int** time; // time period in years

// Constructor to initialize the SimpleInterestCalculator object

**public** SimpleInterestCalculator(**double** principal, **double** rate, **int** time) {

**this**.principal = principal;

**this**.rate = rate;

**this**.time = time;

}

// Method to calculate simple interest based on encapsulated variables

**public** **double** calculateSimpleInterest() {

**double** interest = (principal \* rate \* time) / 100;

**return** interest;

}

// Getter method for principal

**public** **double** getPrincipal() {

**return** principal;

}

// Setter method for principal

**public** **void** setPrincipal(**double** principal) {

**this**.principal = principal;

}

// Getter method for rate

**public** **double** getRate() {

**return** rate;

}

// Setter method for rate

**public** **void** setRate(**double** rate) {

**this**.rate = rate;

}

// Getter method for time

**public** **int** getTime() {

**return** time;

}

// Setter method for time

**public** **void** setTime(**int** time) {

**this**.time = time;

}

// Main method to demonstrate usage

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

// Creating an instance of SimpleInterestCalculator

SimpleInterestCalculator calculator = **new** SimpleInterestCalculator(1000, 5, 2);

// Calculate and display simple interest

**double** interest = calculator.calculateSimpleInterest();

System.***out***.println("Simple Interest: " + interest);

// Update principal amount using setter and calculate interest again

calculator.setPrincipal(1500);

interest = calculator.calculateSimpleInterest();

System.***out***.println("Updated Simple Interest: " + interest);

}

}

Example-3

—---------

2) Imagine you are developing a Car Rental System, and you need to design a class

named Car to represent individual cars within the system. The Car class should have attributes

model and year. Additionally, you need methods to get and set these attributes.

Attributes:

model: Represents the specific model of the car (e.g., Camry, Accord).

year: Represents the manufacturing year of the car.

Methods:

getModel(): Returns the model of the car.

getYear(): Returns the manufacturing year of the car.

setModel(String model): Sets the model of the car.

setYear(int year): Sets the manufacturing year of the car.

Code:

package q22919;

import java.util.Scanner;

public class Car {

// write your code here..

private String model;

private int year;

public Car(String model,int year){

this.model=model;

this.year=year;

}

public String getModel(){

return model;

}

public void setModel(String model){

this.model=model;

}

public int getYear(){

return year;

}

public void setYear(int year){

this.year=year;

}

public static void main(String[] args){

Scanner scanner=new Scanner(System.in);

Car rentalCar=new Car(&quot;Toyota Camry&quot;,2022);

System.out.print(&quot;Model of car: &quot;);

String userEnteredModel = scanner.nextLine();

rentalCar.setModel(userEnteredModel);

System.out.print(&quot;Manufacturing year of car: &quot;);

int userEnteredYear = scanner.nextInt();

rentalCar.setYear(userEnteredYear);

// Display details

System.out.println(&quot;Details of the Rental Car:&quot;);

System.out.println(&quot;Model: &quot; + rentalCar.getModel());

System.out.println(&quot;Year: &quot; + rentalCar.getYear());

// close the scanner

scanner.close();

}

}

### **Encapsulation Benefits:**

· **Data Hiding**: The private variables (principal, rate, time) are hidden from external classes, preventing direct access and ensuring that their state is managed internally by the class.

· **Abstraction**: Users of the SimpleInterestCalculator class interact with it through its public methods (calculateSimpleInterest(), getters, and setters), abstracting away the details of how simple interest is calculated or how the variables are stored.

· **Flexibility and Control**: The encapsulated methods (calculateSimpleInterest(), getters, and setters) provide a controlled way to interact with and modify the object's state, allowing for validation, error checking, or additional operations if needed.

**Inheritance**

**----------------**

inheritance is a fundamental concept in object-oriented programming (OOP) where a class (subclass or derived class) can inherit attributes and methods from another class (superclass or base class). This allows classes to be organized in a hierarchical structure, where common behaviors and characteristics are defined in a higher-level class and specialized behaviors are defined in lower-level classes.

### **Syntax in Java:**

In Java, inheritance is implemented using the extends keyword.

### **Benefits of extends Keyword:**

· **Code Reusability**: Inheritance allows subclasses to reuse code from a superclass, reducing redundancy and promoting modular code design.

· **Polymorphism**: Subclasses can be treated as instances of their superclass, allowing for flexibility in programming and facilitating polymorphic behavior.

· **Hierarchy and Structure**: Inheritance helps in organizing classes into a hierarchical structure based on common characteristics and behaviors, enhancing code readability and maintainability.

**Type Casting in Inheritance**

**----------------------------------------**

In Java, type casting is the process of converting a reference of one type to another. When dealing with inheritance, there are two main types of casting: upcasting and downcasting.

1. **Upcasting**:

o Upcasting is casting from a subclass to a superclass.

o It's implicit and doesn't require an explicit cast operator ((SuperClass)).

o Upcasting is safe and always allowed because a subclass object can always be treated as a superclass object.

2. **Downcasting**:

o Downcasting is casting from a superclass to a subclass.

o It's explicit and requires an explicit cast operator ((SubClass)).

o Downcasting is potentially unsafe because it involves converting a superclass reference to a subclass reference. It can lead to ClassCastException at runtime if the object being cast is not actually an instance of the subclass.

Example:

class Parent {

public void methodOne(){ }

}

class Child extends Parent {

public void methodTwo() { }

}

**Conclusion :**

1. Whatever the parent has by default available to the child but whatever the child has by default not available to the parent. Hence on the child reference we can call both parent and child class methods. But on the parent reference we can call only methods available in the parent class and we can't call child specific methods.

2. Parent class reference can be used to hold child class object but by using that reference we can call only methods available in parent class and child specific methods we can't call.

3. Child class reference cannot be used to hold parent class object.

### **Example Program:**

Let's demonstrate upcasting and downcasting with a meaningful example using classes Animal, Dog (subclass of Animal), and Cat (subclass of Animal).

**package** com.codegnan.oopprograming;

//Animal class (superclass)

**class** Animal {

**public** **void** makeSound() {

System.***out***.println("Some sound");

}

}

//Dog class (subclass of Animal)

**class** Dog **extends** Animal {

**public** **void** makeSound() {

System.***out***.println("Bark");

}

**public** **void** fetch() {

System.***out***.println("Fetching ball");

}

}

//Cat class (subclass of Animal)

**class** Cat **extends** Animal {

**public** **void** makeSound() {

System.***out***.println("Meow");

}

**public** **void** scratch() {

System.***out***.println("Scratching furniture");

}

}

//Main class to demonstrate upcasting and downcasting

**public** **class** Demo {

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

// Upcasting

Animal animal1 = **new** Dog(); // Upcasting Dog to Animal

Animal animal2 = **new** Cat(); // Upcasting Cat to Animal

animal1.makeSound(); // Outputs "Bark" - calls Dog's overridden makeSound() method

animal2.makeSound(); // Outputs "Meow" - calls Cat's overridden makeSound() method

// Downcasting

**if** (animal1 **instanceof** Dog) {

Dog dog = (Dog) animal1; // Downcasting Animal to Dog

dog.fetch(); // Valid because animal1 is actually referencing a Dog object

}

// Attempting unsafe downcasting

// Animal animal3 = new Animal();

// Dog dog2 = (Dog) animal3; // This would throw ClassCastException at runtime

}

}

**Inheritance with respective to the final keyword**

**-------------------------------------------------**

**Inheritance and final Keyword**

* **Inheritance**: In Java, inheritance allows a subclass (child class) to inherit fields and methods from a superclass (parent class). This promotes code reuse and allows for a hierarchical organization of classes.
* **Final Keyword**: Final is the modifier applicable for classes, methods and variables.

**Final variables:**

**Final instance variables:**

* If the value of a variable is varied from object to object such type of variables are called instance variables.
* For every object a separate copy of instance variables will be created.

DIAGRAM:

For the instance variables it is not required to perform initialization explicitly jvm will always provide default values.

Example:

class Test

{

int i;

public static void main(String args[]){

Test t=new Test();

System.out.println(t.i);

}}

Output:

D:\Java>javac Test.java

D:\Java>java Test

0

If the instance variable declared as the final compulsory we should perform initialization explicitly and JVM won't provide any default values.  
 whether we are using or not otherwise we will get compile time error.

**Example:** Program 1:

class Test

{

int i;

}

Output:

D:\Java>javac Test.java

D:\Java>

Program 2:

class Test

{

final int i;

}

Output:

Compile time error.

D:\Java>javac Test.java

Test.java:1: variable i might not have been initialized

class Test

**Rule:**

For the final instance variables we should perform initialization before constructor completion. That is the following are various possible places for this.

**1) At the time of declaration:**

Example:

class Test

{

final int i=10;

}

Output:

D:\Java>javac Test.java

D:\Java>

**2) Inside instance block:**

Example:

class Test

{

final int i;

{

i=10;

}}

Output:

D:\Java>javac Test.java

D:\Java>

**3) Inside constructor:**

Example:

class Test

{

final int i;

Test()

{

i=10;

}}

Output:

D:\Java>javac Test.java

D:\Java>

Example program for all thye three combinations

-----------------------------------------------------

sIf we are performing initialization anywhere else we will get compile time error.

Example:

class Test

{

final int i;

public void methodOne(){

i=10;

}}

Output:

Compile time error.

D:\Java>javac Test.java

Test.java:5: cannot assign a value to final variable i

i=10;

**Final static variables:**

* If the value of a variable is not varied from object to object such type of variables is not recommended to declare as the instance variables. We have to declare those variables at class level by using static modifier.
* In the case of instance variables for every object a seperate copy will be created but in the case of static variables a single copy will be created at class level and shared by every object of that class.
* For the static variables it is not required to perform initialization explicitly jvm will always provide default values.

Example:

class Test

{

static int i;

public static void main(String args[]){

System.out.println("value of i is :"+i);

}}

Output:

D:\Java>javac Test.java

D:\Java>java Test

Value of i is: 0

If the static variable declare as final then compulsory we should perform initialization explicitly whether we are using or not otherwise we will get compile time error.(The JVM won't provide any default values)

Example:  
  
  
  
  
 **Rule:**

For the final static variables we should perform initialization before class loading completion otherwise we will get compile time error. That is the following are possible places.

**1) At the time of declaration:**

Example:

class Test

{

final static int i=10;

}

Output:

D:\Java>javac Test.java

D:\Java>

**2) Inside static block:**

Example:

class Test

{

final static int i;

static

{

i=10;

}}

Output:

Compile successfully.

If we are performing initialization anywhere else we will get compile time error.

Example:

class Test

{

final static int i;

public static void main(String args[]){

i=10;

}}

Output:

Compile time error.

D:\Java>javac Test.java

Test.java:5: cannot assign a value to final variable i

i=10;

**Final local variables:**

* To meet temporary requirement of the Programmer sometime we can declare the variable inside a method or block or constructor such type of variables are called local variables.
* For the local variables jvm won't provide any default value compulsory we should perform initialization explicitly before using that variable.

Example:

class Test

{

public static void main(String args[]){

int i;

System.out.println("hello");

}}

Output:

D:\Java>javac Test.java

D:\Java>java Test

Hello

Example:

class Test

{

public static void main(String args[]){

int i;

System.out.println(i);

}}

Output:

Compile time error.

D:\Java>javac Test.java

Test.java:5: variable i might not have been initialized

System.out.println(i);

Even though local variable declared as the final before using only we should perform initialization.

Example:

class Test

{

public static void main(String args[]){

final int i;

System.out.println("hello");

}}

Output:

D:\Java>javac Test.java

D:\Java>java Test

hello

**Note:** The only applicable modifier for local variables is final if we are using any other modifier we will get compile time error.

Example:

Output:

Compile time error.

D:\Java>javac Test.java

Test.java:5: illegal start of expression

private int x=10;

**Final Methods:**

* Whatever the methods parent has by default available to the child.
* If the child is not allowed to override any method, that method we have to declare with final in parent class. That is final methods cannot overridden.

**Example:** Program 1:

class Parent

{

public void property(){

System.out.println("cash+gold+land");

}

public final void marriage(){

System.out.println("subbalakshmi");

}}

Program 2:

class child extends Parent

{

public void marriage(){

System.out.println("Thamanna");

}}

OUTPUT:

Compile time error.

D:\Java>javac Parent.java

D:\Java>javac child.java

child.java:3: marriage() in child cannot override marriage() in Parent;

overridden method is final

public void marriage(){

### **Final Class:**

If a class declared as the final then we cann't creates the child class that is inheritance concept is not applicable for final classes.

**Example:** Program 1:

final class Parent

{

}

Program 2:

class child extends Parent

{

}

OUTPUT:

Compile time error.

D:\Java>javac Parent.java

D:\Java>javac child.java

child.java:1: cannot inherit from final Parent

class child extends Parent

**Note:** Every method present inside a final class is always final by default whether we are declaring or not. But every variable present inside a final class need not be final.

**Packages in java**

**-------------------**

**Predefined packages**

**---------------------------------**

· A **package** is a group of similar types of classes, interfaces and sub-packages.

· Package in java can be categorized in two forms,

- built-in package and

- user-defined package.

· There are many built-in packages such as java.lang, awt, javax, swing, net, io, util, sql etc.

* A package is a way to organise your Java projects.
* consider packages as folders.
* Packages contain group of related classes
* **Companies use their domain names reversed as a package.**
* So codegnan.com., which is our website, becomes com.codegnan as a package.

Packages - Advantages

* Java package is used to categorize the classes and interfaces so that they can be easily maintained.
* Java package provides access protection.
* Java package removes naming collision.

Banking application

* Account Management

· Account creation - AccountCreation.java,

· AccountUpdation.java

· Withdraw and deposits

·

User registration

-activating and deactivation of users

-authentication and authorization

codegnan.com

com.codegnan.bankngapp.acctmangmt

Domain name: codegnan.com - com.codegnan.banking.accmngmt.acctcreation

Syntax of Package

**package packagename;**

This must be the first line of a class.

* When we want to access a class which is inside the same package as your class, then we don't need any import statement.
* But if we are accessing a class from outside the package, then we need import statement.

Syntax of import statement

import packagename.classname;

import packagename.\*;

Usage of wildcard characters

* Wildcard characters like asterisk (\*) can be used to import all classes in a specific package.
* Eg. import com.codegnan.usermanagement.\*;

But it is always a good practice to avoid wild card to import a class. We must import the specific class.

Eg. import com.codegnan.usermamnagement.User;

1. **java.lang**:

o This package is automatically imported into all Java programs.

o It contains fundamental classes like Object, String, Integer, Boolean, etc.

o Provides basic language support such as exceptions, threads, and the core classes of Java.

2. **java.util**:

o Contains utility classes such as data structures (ArrayList, HashMap, etc.) and utilities for handling dates, times, and other common functionalities.

o Includes the Collections framework for managing and manipulating collections of objects.

3. **java.io**:

o Provides classes for input and output operations.

o Includes classes for reading/writing files (File, InputStream, OutputStream, Reader, Writer) and handling streams (BufferedReader, PrintWriter, etc.).

4. **java.net**:

o Contains classes and interfaces for networking operations.

o Supports networking protocols (like TCP/IP) and provides classes like URL, URLConnection, Socket, ServerSocket, etc.

5. **java.awt** and **javax.swing**:

o These packages are used for creating graphical user interfaces (GUIs).

o java.awt (Abstract Window Toolkit) provides basic GUI components.

o javax.swing provides enhanced components and utilities built on top of java.awt.

6. **java.sql**:

o Provides classes and interfaces for JDBC (Java Database Connectivity).

o Allows Java applications to interact with databases using SQL queries.

7. **java.text**:

o Contains classes for formatting and parsing textual data, including dates, numbers, and messages.

o Includes classes like DateFormat, NumberFormat, and MessageFormat.

8. **java.security**:

o Provides classes and interfaces for security-related operations.

o Includes cryptography, secure random number generation, permissions, and authentication.

9. **java.lang.reflect**:

o Provides classes and interfaces for obtaining information about classes and objects at runtime.

o Allows dynamic loading of classes, introspection, and reflection capabilities.

10. **java.util.concurrent**:

o Contains utility classes for concurrent programming.

o Includes classes like Executor, ThreadPoolExecutor, CountDownLatch, Semaphore, etc., for managing concurrent tasks and synchronization.

These predefined packages cover a wide range of functionalities, making Java a versatile and powerful language for various types of application development. By importing these packages, developers can leverage pre-written code to save time and effort while building robust Java applications.

## **Import statement:**

class Test{

public static void main(String args[]){

ArrayList l=new ArrayList();

}

}

Output:

Compile time error.

D:\Java>javac Test.java

Test.java:3: cannot find symbol

symbol : class ArrayList

location: class Test

ArrayList l=new ArrayList();

* We can resolve this problem by using fully qualified name "**java.util.ArrayList l=new java.util.ArrayList();**". But problem with using fully qualified name every time is it increases length of the code and reduces readability.
* We can resolve this problem by using import statements.

Example:

import java.util.ArrayList;

class Test{

public static void main(String args[]){

ArrayList l=new ArrayList();

}

}

Output:

D:\Java>javac Test.java

Hence whenever we are using import statement it is not require to use fully qualified names we can use short names directly. This approach decreases length of the code and improves readability.

### **Case 1: Types of Import Statements:**

There are 2 types of import statements.

1) **Explicit class import** 2) **Implicit class import.**

### **Explicit class import:**

Example: **Import java.util.ArrayList**

* This type of import is highly recommended to use because it improves readability of the code.
* Best suitable for Hi-Tech city where readability is important.

### **Implicit class import:**

Example: **import java.util.\*;**

* It is never recommended to use because it reduces readability of the code.
* Best suitable for Ameerpet where typing is important.

**Modifiers related to the packages:**

**==================================**

**Default modifier:**

**----------------------**

The default access modifier in Java, often referred to as "package-private", restricts visibility of classes, methods, and fields to within the same package. Here’s an example program that demonstrates the default access modifier within packages:

1. MyClass.java (in package com.example):

// MyClass.java in package com.example

package com.example;

// Default access modifier (package-private)

class MyClass {

void display() {

System.out.println("Hello from MyClass");

}

}

o Comments:

MyClass is declared with default access (class MyClass { ... }).

display() method is also declared with default access (void display() { ... }).

Both MyClass and display() are accessible only within the same package (com.example).

2. AnotherClass.java (in the same package com.example):

// AnotherClass.java in package com.example

package com.example;

public class AnotherClass {

public static void main(String[] args) {

MyClass obj = new MyClass();

obj.display(); // Accessing default method from the same package

}

}





### **Explanation:**

* **Package Structure**:
  + Both MyClass and AnotherClass are in the com.example package.
  + MyClass is not explicitly declared as public, protected, or private, so it has default access, meaning it can only be accessed within the com.example package.

**Public modifier**

**================**

In Java, the public access modifier allows classes, methods, and fields to be accessed from any other class in any package. Here’s how you can use the public modifier within packages, along with example programs:

### 

### **1. PublicClass.java (in package com.example):**

### **// PublicClass.java in package com.example**

### **package com.example;**

### 

### **// Public access modifier**

### **public class PublicClass {**

### **public void display() {**

### **System.out.println("Hello from PublicClass");**

### **}**

### **}**

### **o Comments:**

### **PublicClass is declared with public access (public class PublicClass { ... }).**

### **display() method is also declared with public access (public void display() { ... }).**

### **Both PublicClass and display() method can be accessed from any other class, regardless of the package.**

### **2. AnotherClass.java (in a different package com.anotherpackage):**

### **// AnotherClass.java in package com.anotherpackage**

### **package com.anotherpackage;**

### 

### **import com.example.PublicClass;**

### 

### **public class AnotherClass {**

### **public static void main(String[] args) {**

### **PublicClass obj = new PublicClass();**

### **obj.display(); // Accessing public method from a different package**

### **}**

### **}**

### **o Comments:**

### **AnotherClass is declared with public access (public class AnotherClass { ... }).**

### **main() method is the entry point of the program.**

### **AnotherClass imports PublicClass from the com.example package using import com.example.PublicClass;.**

### **AnotherClass can instantiate PublicClass and call its display() method because PublicClass is declared public and accessible from any package.**

### **Explanation:**

* **Package Structure**:
  + PublicClass is in the com.example package.
  + AnotherClass is in the com.anotherpackage package.
* **Accessing Public Classes and Methods**:
  + AnotherClass can import PublicClass from the com.example package and access its public methods (display() in this case).
  + This demonstrates how public access modifier allows classes and methods to be widely accessible across packages, facilitating code reuse and modularity.

**Private modifier**

**===================**

In Java, the private access modifier restricts access to the declared class, method, or field within the same class. It is the most restrictive access level. Here's how private works with classes, methods, and fields, along with example programs:

1. PrivateClass.java:

// PrivateClass.java

package com.example;

public class PrivateClass {

private int privateField;

private void privateMethod() {

System.out.println("Private method called");

}

public void accessPrivateMembers() {

privateField = 10; // Accessing private field within the same class

privateMethod(); // Accessing private method within the same class

}

}

o Comments:

PrivateClass is declared with public access (public class PrivateClass { ... }).

privateField is a private instance variable (private int privateField;).

privateMethod() is a private method (private void privateMethod() { ... }).

accessPrivateMembers() is a public method that can access privateField and privateMethod() within the same class (PrivateClass).

2. AnotherClass.java (in a different package or same package):

// AnotherClass.java

package com.anotherpackage;

import com.example.PrivateClass;

public class AnotherClass {

public static void main(String[] args) {

PrivateClass obj = new PrivateClass();

// obj.privateField; // Error: privateField has private access in PrivateClass

// obj.privateMethod(); // Error: privateMethod() has private access in PrivateClass

obj.accessPrivateMembers(); // Accessing public method that internally accesses private members

}

}

o Comments:

AnotherClass is declared with public access (public class AnotherClass { ... }).

main() method is the entry point of the program.

AnotherClass imports PrivateClass from the com.example package using import com.example.PrivateClass;.

AnotherClass can create an instance of PrivateClass, but it cannot directly access privateField or privateMethod() due to their private access.

### **Explanation:**

· **Private Access Modifier**:

* + private members (privateField and privateMethod() in PrivateClass) are accessible only within the same class (PrivateClass).
  + They cannot be accessed from any other class, not even subclasses or classes in the same package (AnotherClass in this case).

· **Indirect Access**:

* + Public methods (accessPrivateMembers() in PrivateClass) can access private members internally.
  + This allows controlled access to private members through public methods, maintaining encapsulation and ensuring data integrity.

**Protected modifier**

**======================**

In Java, the protected access modifier allows access to the member (method or field) within the same package and by subclasses, regardless of whether they are in the same package or different packages. Here’s how protected works with classes, methods, and fields, along with example programs:

1. ParentClass.java (in package com.example):

// ParentClass.java in package com.example

package com.example;

// Protected access modifier

public class ParentClass {

protected void protectedMethod() {

System.out.println("Protected method called");

}

}

o Comments:

ParentClass is declared with public access (public class ParentClass { ... }).

protectedMethod() is declared with protected access (protected void protectedMethod() { ... }).

protectedMethod() can be accessed within the same package (com.example) and by subclasses, regardless of the package.

2. ChildClass.java (in a different package com.anotherpackage):

// ChildClass.java in package com.anotherpackage

package com.anotherpackage;

import com.example.ParentClass;

public class ChildClass extends ParentClass {

public static void main(String[] args) {

ChildClass obj = new ChildClass();

obj.protectedMethod(); // Accessing protected method from subclass

}

}

o Comments:

ChildClass extends ParentClass, inheriting its protected members (protectedMethod()).

main() method is the entry point of the program.

ChildClass imports ParentClass from the com.example package using import com.example.ParentClass;.

ChildClass can access protectedMethod() because it inherits it from ParentClass and protectedMethod() is protected.

### **Explanation:**

· **Protected Access Modifier**:

* + protected members (protectedMethod() in ParentClass) are accessible within the same package (com.example) and by subclasses (ChildClass in com.anotherpackage).

Polymorphism:

--------------

Polymorphism is a Greak Word, where Poly means Many and Morphism means Structures or forms. If one thing is existed in more than one form then it is called as Polymorphism. The main advantage of Polymorphism is "Flexibility" to design application.

Example 1: We can use same abs() method for int type, long type, float type etc.

Example:

1. abs(int)

2. abs(long)

3. abs(float)

Example 2:  
 We can use the parent reference to hold any child objects.  
 We can use the same List reference to hold ArrayList object, LinkedList object, Vector object, or Stack object.

Example:

1. List l=new ArrayList();

2. List l=new LinkedList();

3. List l=new Vector();

4. List l=new Stack();

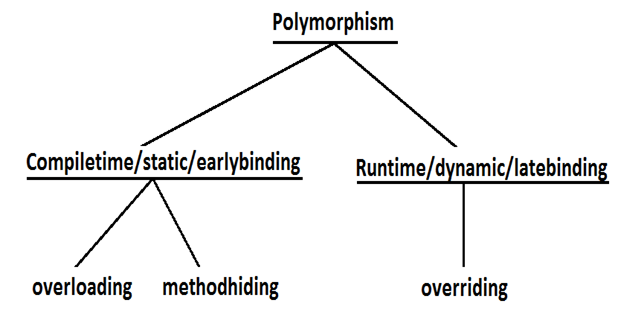
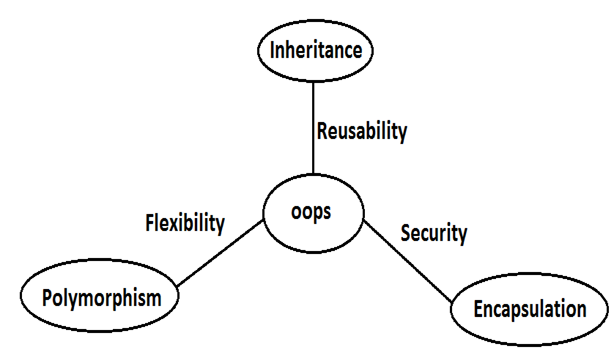
Diagram:  


Diagram: **3 Pillars of OOPS  
  
**

1) Inheritance talks about reusability.  
 2) Polymorphism talks about flexibility.  
 3) Encapsulation talks about security.

**Beautiful definition of polymorphism:**

A boy starts love with the word friendship, but girl ends love with the same word friendship, word is the same but with different attitudes. This concept is nothing but polymorphism.

**Overloading**

**------------------**

Method overloading in Java allows a class to have more than one method having the same name if their parameter lists are different. It is a compile-time polymorphism technique where the method to be executed is determined by the compiler based on the method signature (name and parameters).

In method overloading:

* Methods must have the same name.
* Methods must have different parameters (number or type).
* Overloaded methods can have different return types, but return type alone is not sufficient to distinguish overloaded methods (this is not recommended for readability).

Example program using overloading

--------------------------------------------------

**package** com.codegnan.relations;

**public** **class** OverloadingExample {

// Method to calculate the sum of two integers

**public** **int** add(**int** a, **int** b) {

**return** a + b;

}

// Overloaded method to calculate the sum of three integers

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

**return** a + b + c;

}

// Overloaded method to calculate the sum of two doubles

**public** **double** add(**double** a, **double** b) {

**return** a + b;

}

// Overloaded method to calculate the sum of an array of integers

**public** **int** add(**int**[] numbers) {

**int** sum = 0;

**for** (**int** num : numbers) {

sum += num;

}

**return** sum;

}

// Overloaded method with different return type (not recommended for clarity)

**public** **double** add(**double** a, **double** b, **double** c) {

**return** a + b + c;

}

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

OverloadingExample example = **new** OverloadingExample();

// Using the add methods

System.***out***.println("Sum of 5 and 3: " + example.add(5, 3)); // Output: Sum of 5 and 3: 8

System.***out***.println("Sum of 2, 4, and 6: " + example.add(2, 4, 6)); // Output: Sum of 2, 4, and 6: 12

System.***out***.println("Sum of 2.5 and 3.7: " + example.add(2.5, 3.7)); // Output: Sum of 2.5 and 3.7: 6.2

**int**[] numbers = { 1, 2, 3, 4, 5 };

System.***out***.println("Sum of array elements: " + example.add(numbers)); // Output: Sum of array elements: 15

}

}

**Conclusion :** In overloading compiler is responsible to perform method resolution(decision) based on the reference type(but not based on run time object). Hence overloading is also considered as compile time polymorphism(or) static polymorphism (or)early biding.

Example-2

**package** com.codegnan.relations;

**public** **class** PaymentProcessor {

// Method to process payment in USD

**public** **void** processPayment(**double** amount) {

System.***out***.println("Payment processed in USD: $" + amount);

// Logic to process payment in USD

}

// Overloaded method to process payment in EUR

**public** **void** processPayment(**double** amount, String currency) {

**if** (currency.equalsIgnoreCase("EUR")) {

System.***out***.println("Payment processed in EUR: €" + amount);

// Logic to process payment in EUR

} **else** {

System.***out***.println("Unsupported currency: " + currency);

// Handle unsupported currency

}

}

// Overloaded method to process payment with additional fee

**public** **void** processPayment(**double** amount, **double** fee) {

**double** totalAmount = amount + fee;

System.***out***.println("Payment processed with fee: $" + totalAmount);

// Logic to process payment with fee

}

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

PaymentProcessor processor = **new** PaymentProcessor();

// Process payment in USD

processor.processPayment(100.0);

// Process payment in EUR

processor.processPayment(80.0, "EUR");

// Process payment with additional fee

processor.processPayment(50.0, 5.0);

}

}

## **Overriding**

1. Whatever the Parent has by default available to the Child through inheritance, if the Child is not satisfied with Parent class method implementation then Child is allow to redefine that Parent class method in Child class in its own way this process is called overriding.

2. The Parent class method which is overridden is called overridden method.

3. The Child class method which is overriding is called overriding method.

**package** com.codegnan.relations;

//Parent class defining properties and behaviors of a parent

**class** Parent {

// Method to display parent's properties

**public** **void** property() {

System.***out***.println("cash + land + gold");

}

// Method to display parent's preferred choice for marriage

**public** **void** marry() {

System.***out***.println("subbalakshmi"); // Original method in Parent class

}

}

//Child class inheriting from Parent class, demonstrating method overriding

**class** Child **extends** Parent {

// Method overriding the marry() method in Parent class

@Override

**public** **void** marry() {

System.***out***.println("Trisha / Nayanatara / Anushka"); // Overridden method in Child class

}

}

//Test class to demonstrate method overriding and polymorphism

**public** **class** Main {

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

Parent p = **new** Parent();

p.marry(); // Output: subbalakshmi (parent method)

Child c = **new** Child();

c.marry(); // Output: Trisha / Nayanatara / Anushka (child method)

Parent p1 = **new** Child(); // Reference of Parent type pointing to a Child object it is called dynamic dispatch method.

p1.marry(); // Output: Trisha / Nayanatara / Anushka (child method)

}

}

In overriding method resolution is always takes care by JVM based on runtime object hence overriding is also considered as runtime polymorphism or dynamic polymorphism or late binding.

The process of overriding method resolution is also known as dynamic method dispatch.

**Note**: In overriding runtime object will play the role and reference type is dummy.

**Example-1**

**—-------------**

import java.util.Scanner;

// write your code here..

class shape{

public void calculateVolume(){

}

}

class Cube extends shape{

public void calculateVolume(){

Scanner scanner=new Scanner(System.in);

System.out.print("Side of the cube: ");

double side=scanner.nextDouble();

double volume=Math.pow(side,3);

System.out.printf("Volume of Cube: %.2f\n", volume);

}

}

class Cuboid extends shape{

public void calculateVolume(){

Scanner scanner=new Scanner(System.in);

System.out.print("Length of the cuboid: ");

double length=scanner.nextDouble();

System.out.print("Width of the cuboid: ");

double width=scanner.nextDouble();

System.out.print("Height of the cuboid: ");

double height=scanner.nextDouble();

double volume=length\*width\*height;

System.out.printf("Volume of Cuboid: %.2f\n", volume);

}

}

public class ShapeCalculator {

public static void main(String[] args) {

// Creating instances of the subclasses

Cube cubeInstance = new Cube();

Cuboid cuboidInstance = new Cuboid();

// Calling calculateVolume() for each instance

cubeInstance.calculateVolume();

cuboidInstance.calculateVolume();

}

}

Example-2

—----------------

3.Create a class named Appointment with a method schedule(). This method should print "Scheduling an appointment".

Create a subclass DoctorAppointment that extends Appointment. Override the schedule() method in the DoctorAppointment class to prompt the user for input. Ask the user to input the specialization (String) and print "Doctor appointment scheduled for {specialization}".

Create another subclass DentistAppointment that also extends Appointment. Override the schedule() method in the DentistAppointment class to prompt the user for input. Ask the user to input the dental procedure (String) and print "Dentist appointment scheduled for {procedure}".

Note: The main class has been provided to you in the editor.

TestCase - 1 (Execution Time: 272ms) Correct Output

Expected Output

Scheduling·an·appointment

Specialization: Orthopedics

Doctor·appointment·scheduled·for·Orthopedics

Dental·procedure: Root Canal

Solution:

AppointmentTest.java

==================================

package q23491;

import java.util.Scanner;

// write your code here..

class Appointment {

void schedule() {

System.out.println("Scheduling an appointment");

}

}

class DoctorAppointment extends Appointment {

void schedule(){

Scanner scanner = new Scanner(System.in);

System.out.print("Specialization: ");

String specialization = scanner.nextLine();

System.out.println("Doctor appointment scheduled for " + specialization);

}

}

class DentistAppointment extends Appointment {

void schedule(){

Scanner scanner = new Scanner(System.in);

System.out.print("Dental procedure: ");

String procedure = scanner.nextLine();

System.out.println("Dentist appointment scheduled for " + procedure);

}

}

public class AppointmentTest {

public static void main(String[] args) {

Appointment genericAppointment = new Appointment();

genericAppointment.schedule();

DoctorAppointment doctorAppointment = new DoctorAppointment();

doctorAppointment.schedule();

DentistAppointment dentistAppointment = new DentistAppointment();

dentistAppointment.schedule();

}

}

Q)What are the differences between Method Overloading and Method Overriding?

-----------------------------------------------------------------

Ans:

----

1. Method overloading is the process of extending existed methodfunctionality up to new functionality.

Method Overriding is the process of providing replacement for the existed method

functionality with some other new functionality.

1. We can perform method overloading with or with out inheritance.

Method overriding is required Inheritance.

2. In Method overloading all method must have same name and differentparameter list, that is , different method signature.

In method overriding, both super class method and sub class method must have samemethod prototype.

3. Method over hiding is not possible with Method Overloading.

Method over hiding is possible with method overriding in the case of static methods.

4. We can overload static methods directly.

We can not override static methods

5. We can overload final methods.

We can not override final methods.

6. We can overload methods which are having different scopes.

In Method overriding, overriding method must have either same scope of the super classmethod or more scope than the super class method.

7. In Java applications, constructor overloading is possible.

In Java applications, constructor overriding is not possible.

**Abstract Methods:**

Even though we don't have implementation still we can declare a method with abstract modifier.  
 That is abstract methods have only declaration but not implementation.  
 Hence abstract method declaration should compulsory ends with semicolon.

Example:

Child classes are responsible to provide implementation for parent class abstract methods.

**Example:** Program:

* The main advantage of abstract methods is , by declaring abstract method in parent class we can provide **guide lines** to the child class such that which methods they should compulsory implement.
* Abstract method never talks about implementation whereas if any modifier talks about implementation then the modifier will be enemy to abstract and that is always illegal combination for methods.

**Abstract class:**

For any java class if we are not allow to create an object such type of class we have to declare with abstract modifier that is for abstract class instantiation is not possible.

Example:

abstract class Test

{

public static void main(String args[]){

Test t=new Test();

}}

Output:

Compile time error.

**What is the difference between abstract class and abstract method ?**

* If a class contain at least on abstract method then compulsory the corresponding class should be declare with abstract modifier. Because implementation is not complete and hence we can't create object of that class.
* Even though class doesn't contain any abstract methods still we can declare the class as abstract that is an abstract class can contain zero no of abstract methods also.

Example program

---------------------

**package** com.codegnan.oopprograming;

//Abstract class Shape

**abstract** **class** Shape {

// Abstract method to calculate area

**public** **abstract** **double** calculateArea();

}

//Concrete subclass Circle inheriting from Shape

**class** Circle **extends** Shape {

**private** **double** radius;

// Constructor

**public** Circle(**double** radius) {

**this**.radius = radius;

}

// Override abstract method to calculate area for circle

@Override

**public** **double** calculateArea() {

**return** Math.***PI*** \* radius \* radius;

}

}

//Concrete subclass Rectangle inheriting from Shape

**class** Rectangle **extends** Shape {

**private** **double** length;

**private** **double** width;

// Constructor

**public** Rectangle(**double** length, **double** width) {

**this**.length = length;

**this**.width = width;

}

// Override abstract method to calculate area for rectangle

@Override

**public** **double** calculateArea() {

**return** length \* width;

}

}

//Main class to test shapes and area calculation

**public** **class** AreaCalculator {

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

// Create a Circle instance

Circle circle = **new** Circle(5.0);

System.***out***.println("Area of Circle: " + circle.calculateArea()); // Output: Area of Circle: 78.53981633974483

// Create a Rectangle instance

Rectangle rectangle = **new** Rectangle(4.0, 6.0);

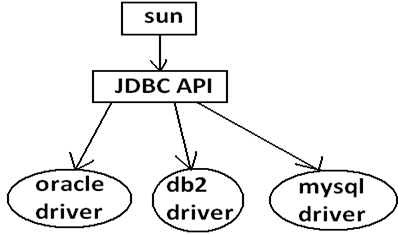
System.***out***.println("Area of Rectangle: " + rectangle.calculateArea()); // Output: Area of Rectangle: 24.0

}

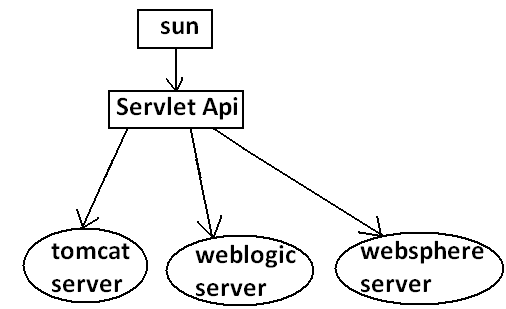
}

## **Interfaces:**

Def1: Any service requirement specification (srs) is called an interface.  
 Example1: Sun people responsible to define JDBC API and database vendor will provide implementation for that.

Diagram:  
  


Example2: Sun people define Servlet API to develop web applications web server vendor is responsible to provide implementation.

Diagram:  


Def2: From the client point of view an interface define the set of services what is expecting. From the service provider point of view an interface defines the set of services what is offering. Hence an interface is considered as a contract between client and service provider.  
 Example: ATM GUI screen describes the set of services what bank people offering, at the same time the same GUI screen the set of services what customer is expecting hence this GUI screen acts as a contract between bank and customer.

Def3: Inside interface every method is always abstract whether we are declaring or not hence interface is considered as 100% pure abstract class.

Summery def: Any service requirement specification (SRS) or any contract between client and service provider or 100% pure abstract classes is considered as an interface.

### 

### **Declaration and implementation of an interface:**

Note1:

Whenever we are implementing an interface compulsory for every method of that interface we should provide implementation otherwise we have to declare class as abstract in that case child class is responsible to provide implementation for remaining methods.

Note2:

Whenever we are implementing an interface method compulsory it should be declared as public otherwise we will get compile time error.

Example:

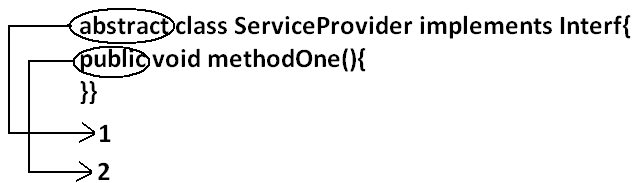
interface Interf

{

void methodOne();

void methodTwo();

}



class SubServiceProvider extends ServiceProvider

{

}

Extends vs implements:

A class can extends only one class at a time.

Example:

class One{

public void methodOne(){

}

}

class Two extends One{

}

A class can implements any no. Of interfaces at a time.

Example:

interface One{

public void methodOne();

}

interface Two{

public void methodTwo();

}

class Three implements One,Two{

public void methodOne(){

}

public void methodTwo(){

}

}

A class can extend a class and can implement any no. Of interfaces simultaneously.

interface One{

void methodOne();

}

class Two

{

public void methodTwo(){

}

}

class Three extends Two implements One{

public void methodOne(){

}

}

An interface can extend any no. Of interfaces at a time.

Example:

interface One{

void methodOne();

}

interface Two{

void methodTwo();

}

interface Three extends One,Two

{

}

Which of the following is true?

1. A class can extend any no.Of classes at a time.
2. An interface can extend only one interface at a time.
3. A class can implement only one interface at a time.
4. A class can extend a class and can implement an interface but not both simultaneously.
5. An interface can implement any no.Of interfaces at a time.
6. None of the above.

Ans: 6

Interface methods:

Every method present inside interface is always public and abstract whether we are declaring or not. Hence inside interface the following method declarations are equal.

void methodOne();

public Void methodOne();

abstract Void methodOne(); Equal

public abstract Void methodOne();

public: To make this method available for every implementation class.  
 abstract: Implementation class is responsible to provide implementation .

As every interface method is always public and abstract we can't use the following modifiers for interface methods.  
 Private, protected, final, static, synchronized, native, strictfp.

Inside interface which method declarations are valid?

1. public void methodOne(){}
2. private void methodOne();
3. public final void methodOne();
4. public static void methodOne();
5. public abstract void methodOne();

Ans: 5

Interface variables:

* An interface can contain variables
* The main purpose of interface variables is to define requirement level constants.
* Every interface variable is always public static and final whether we are declaring or not.

Example:

interface interf

{

int x=10;

}

public: To make it available for every implementation class.  
 static: Without existing object also we have to access this variable.  
 final: Implementation class can access this value but cannot modify.

Hence inside interface the following declarations are equal.

int x=10;

public int x=10;

static int x=10;

final int x=10; Equal

public static int x=10;

public final int x=10;

static final int x=10;

public static final int x=10;

* As every interface variable by default public static final we can't declare with the following modifiers.
  + Private
  + Protected
  + Transient
  + Volatile
* For the interface variables compulsory we should perform initialization at the time of declaration only otherwise we will get compile time error.

Example:

interface Interf

{

int x;

}

Output:

Compile time error.

D:\Java>javac Interf.java

Interf.java:3: = expected

int x;

Which of the following declarations are valid inside interface ?

1. int x;
2. private int x=10;
3. public volatile int x=10;
4. public transient int x=10;
5. public static final int x=10;

Ans: 5

**What is the difference between interface, abstract class and concrete class?  
 When we should go for interface, abstract class and concrete class?**

* If we don't know anything about implementation just we have requirement specification then we should go for interface.
* If we are talking about implementation but not completely (partial implementation) then we should go for abstract class.
* If we are talking about implementation completely and ready to provide service then we should go for concrete class

**What is the Difference between interface and abstract class ?**

| **interface** | **Abstract class** |
| --- | --- |
| If we don't' know anything about implementation just we have requirement specification then we should go for interface. | If we are talking about implementation but not completely (partial implementation) then we should go for abstract class. |
| Every method present inside interface is always public and abstract whether we are declaring or not. | Every method present inside abstract class need not be public and abstract. |
| We can't declare interface methods with the modifiers private, protected, final, static, synchronized, native, strictfp. | There are no restrictions on abstract class method modifiers. |
| Every interface variable is always public static final whether we are declaring or not. | Every abstract class variable need not be public static final. |
| Every interface variable is always public static final we can't declare with the following modifiers. Private, protected, transient, volatile. | There are no restrictions on abstract class variable modifiers. |
| For the interface variables compulsory we should perform initialization at the time of declaration otherwise we will get compile time error. | It is not require to perform initialization for abstract class variables at the time of declaration. |
| Inside interface we can't take static and instance blocks. | Inside abstract class we can take both static and instance blocks. |
| Inside interface we can't take constructor. | Inside abstract class we can take constructor. |

**We can't create object for abstract class but abstract class can contain constructor what is the need ?**

abstract class constructor will be executed when ever we are creating child class object to perform initialization of child object.

**Interface example program**

**—----------------------------------------**

**package** com.codegnan.oopprogramming;

**interface** Shape {

**public** **abstract** **double** calculateArea();

**public** **abstract** **double** calculateVolume();

}

**class** Circle **implements** Shape {

**double** radius;

//constructor

**public** Circle(**double** radius) {

**this**.radius = radius;

}

@Override

**public** **double** calculateArea() {

**return** Math.***PI*** \* radius \* radius;

}

@Override

**public** **double** calculateVolume() {

**return** 0;// circle has no volume

}

}

**class** Rectangle **implements** Shape {

**double** length;

**double** breadth;

**public** Rectangle(**double** length, **double** breadth) {

**this**.length = length;

**this**.breadth = breadth;

}

@Override

**public** **double** calculateArea() {

**return** length \* breadth;

}

@Override

**public** **double** calculateVolume() {

**return** 0;// rectangle has no volume

}

}

**class** Cylinder **implements** Shape {

**double** radius;

**double** height;

**public** Cylinder(**double** radius, **double** height) {

**super**();

**this**.radius = radius;

**this**.height = height;

}

**public** **double** calculateArea() {

**return** 2 \* Math.***PI*** \* radius \* (radius + height);

}

@Override

**public** **double** calculateVolume() {

**return** Math.***PI*** \* radius \* radius \* height;

}

}

**public** **class** Main {

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

Shape circle = **new** Circle(5);

Shape rectangle = **new** Rectangle(4, 6);

Shape cylinder = **new** Cylinder(3, 7);

// print the circle class details

System.***out***.println("Circle Area : " + circle.calculateArea());

System.***out***.println("Circle Volume : " + circle.calculateVolume());

// rectangle

System.***out***.println("Rectangle Area : " + rectangle.calculateArea());

System.***out***.println("Rectangle Volume : " + rectangle.calculateVolume());

// Cylinder

System.***out***.println("Cylinder Area : " + cylinder.calculateArea());

System.***out***.println("Cylinder Volume : " + cylinder.calculateVolume());

}

}

You are working on a password-strength checker application. The application needs to evaluate the strength of a given password based on two criteria: length and complexity. Your task is to design and implement the necessary Java classes and interface for this application.

Define an interface named PasswordChecker with the following methods:

checkLength(String password): This method should determine the length of the given password and display the length.

checkComplexity(String password): This method should analyze the complexity of the password and return a string indicating its strength. If the length is less than 8 characters, the password is considered "Weak." If the length is exactly 8 characters, it is labeled as "Medium." If the length is greater than 8 characters, it is marked as "Strong."

Implement a class named SimplePasswordChecker that implements the PasswordChecker interface. In this class, provide the necessary logic for both the checkLength and checkComplexity methods.

/CheckPassword.java

—-------------------------

**package** com.codegnan.oopprogramming;

**import** java.util.Scanner;

**interface** PasswordChecker {

**public** **abstract** String checkLength(String password);

**public** **abstract** String checkComplexity(String password);

}

**class** SimplePasswordChecker **implements** PasswordChecker {

@Override

**public** String checkLength(String password) {

**return** ("Length: " + password.length());

}

@Override

**public** String checkComplexity(String password) {

**if** (password.length() < 8) {

**return** "weak";

} **else** **if** (password.length() == 8) {

**return** "Medium";

} **else** {

**return** "Strong";

}

}

}

**public** **class** CheckPassword {

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

Scanner scanner = **new** Scanner(System.***in***);

System.***out***.println("Enter a Password : ");

String password = scanner.nextLine();

//create an instance of the simplepasswordchecker class.

PasswordChecker simpleChecker = **new** SimplePasswordChecker();

System.***out***.println(simpleChecker.checkLength(password));

System.***out***.println(simpleChecker.checkComplexity(password));

}

}

Multiple interface implementation in multiple inheritance(don’t use the terminantion word multiple inheritance)

---------------------------------------------

**package** com.codegnan.oopprogramming;

**interface** CanRun {

**public** **abstract** **void** run();

}

**interface** CanClimb {

**public** **abstract** **void** climb();

}

**class** Monkey **implements** CanRun, CanClimb {

@Override

**public** **void** climb() {

System.***out***.println("Monkey can Climb...");

}

@Override

**public** **void** run() {

System.***out***.println("Monkey is Running");

}

}

**class** Cheetah **implements** CanRun {

@Override

**public** **void** run() {

System.***out***.println("Cheetah is Running...");

}

}

**public** **class** Main2 {

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

Monkey monkey = **new** Monkey();

Cheetah cheetah = **new** Cheetah();

monkey.run();

monkey.climb();

cheetah.run();

}

}

Course.java

—---------------

**package** com.codegnan.university.managment;

// Represents a course in the university

**public** **class** Course {

**private** String title; // Title of the course

// Constructor to initialize the course with a title

**public** Course(String title) {

**this**.title = title; // Set the course title

}

// Getter method to retrieve the course title

**public** String getTitle() {

**return** title; // Return the course title

}

// Override toString method to provide a string representation of the course

@Override

**public** String toString() {

**return** title; // Return the course title as the string representation

}

}

Professor.java

—------------------

package com.codegnan.university.managment;

import java.util.ArrayList;

import java.util.List;

// Represents a professor in the university

public class Professor {

private String name; // Name of the professor

private List<Course> assignedCourses; // List to hold courses assigned to the professor

// Constructor to initialize the professor with a name and an empty list of assigned courses

public Professor(String name) {

this.name = name; // Set the professor's name

this.assignedCourses = new ArrayList<>(); // Initialize the list of assigned courses

}

// Getter method to retrieve the professor's name

public String getName() {

return name; // Return the professor's name

}

// Method to assign a course to the professor

public void assignCourse(Course course) {

// Check if the course is not already in the list of assigned courses

if (!assignedCourses.contains(course)) {

assignedCourses.add(course); // Add the course to the list if not already present

}

}

// Getter method to retrieve the list of courses assigned to the professor

public List<Course> getAssignedCourses() {

return assignedCourses; // Return the list of assigned courses

}

// Override toString method to provide a string representation of the professor

@Override

public String toString() {

return name; // Return the professor's name as the string representation

}

}

Student.java

—--------------------

package com.codegnan.university.managment;

import java.util.ArrayList;

import java.util.List;

// Represents a student in the university

public class Student {

private String name; // Name of the student

private List<Course> enrolledCourses; // List to hold courses the student is enrolled in

// Constructor to initialize the student with a name and an empty list of enrolled courses

public Student(String name) {

this.name = name; // Set the student's name

this.enrolledCourses = new ArrayList<>(); // Initialize the list of enrolled courses

}

// Getter method to retrieve the student's name

public String getName() {

return name; // Return the student's name

}

// Method to enroll the student in a course

public void enrollInCourse(Course course) {

// Check if the course is not already in the list of enrolled courses

if (!enrolledCourses.contains(course)) {

enrolledCourses.add(course); // Add the course to the list

}

}

// Getter method to retrieve the list of courses the student is enrolled in

public List<Course> getEnrolledCourses() {

return enrolledCourses; // Return the list of enrolled courses

}

// Override toString method to provide a string representation of the student

@Override

public String toString() {

return name; // Return the student's name as the string representation

}

}

UniversityManagment.java

—-------------------------------------

package com.codegnan.university.managment;

import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;

// Manages the university's students, professors, and courses

public class UniversityManagement {

private List<Student> students; // List to hold students

private List<Professor> professors; // List to hold professors

private List<Course> courses; // List to hold courses

// Constructor initializes empty lists for students, professors, and courses

public UniversityManagement() {

students = new ArrayList<>();

professors = new ArrayList<>();

courses = new ArrayList<>();

}

// Method to add a student

public void addStudent(String name) {

students.add(new Student(name));

}

// Method to add a professor

public void addProfessor(String name) {

professors.add(new Professor(name));

}

// Method to add a course

public void addCourse(String title) {

courses.add(new Course(title));

}

// Method to enroll a student in a course

public void enrollStudentInCourse(String studentName, String courseTitle) {

Student student = findStudentByName(studentName); // Find the student by name

Course course = findCourseByTitle(courseTitle); // Find the course by title

if (student != null && course != null) {

student.enrollInCourse(course); // Enroll the student in the course

} else {

System.out.println("Student or course not found."); // Error message if not found

}

}

// Method to assign a course to a professor

public void assignCourseToProfessor(String professorName, String courseTitle) {

Professor professor = findProfessorByName(professorName); // Find the professor by name

Course course = findCourseByTitle(courseTitle); // Find the course by title

if (professor != null && course != null) {

professor.assignCourse(course); // Assign the course to the professor

} else {

System.out.println("Professor or course not found."); // Error message if not found

}

}

// Method to list all students

public void listStudents() {

System.out.println("Students:");

for (Student student : students) {

System.out.println(student); // Print each student's name

}

}

// Method to list all professors

public void listProfessors() {

System.out.println("Professors:");

for (Professor professor : professors) {

System.out.println(professor); // Print each professor's name

}

}

// Method to list all courses

public void listCourses() {

System.out.println("Courses:");

for (Course course : courses) {

System.out.println(course); // Print each course's title

}

}

// Method to display the courses a student is enrolled in

public void displayStudentCourses(String studentName) {

Student student = findStudentByName(studentName); // Find the student by name

if (student != null) {

System.out.println("Courses for student " + studentName + ":");

for (Course course : student.getEnrolledCourses()) {

System.out.println(course); // Print each enrolled course

}

} else {

System.out.println("Student not found."); // Error message if student not found

}

}

// Method to display the courses assigned to a professor

public void displayProfessorCourses(String professorName) {

Professor professor = findProfessorByName(professorName); // Find the professor by name

if (professor != null) {

System.out.println("Courses assigned to professor " + professorName + ":");

for (Course course : professor.getAssignedCourses()) {

System.out.println(course); // Print each assigned course

}

} else {

System.out.println("Professor not found."); // Error message if professor not found

}

}

// Helper method to find a student by name

private Student findStudentByName(String name) {

for (Student student : students) {

if (student.getName().equalsIgnoreCase(name)) {

return student; // Return the student if found

}

}

return null; // Return null if student not found

}

// Helper method to find a professor by name

private Professor findProfessorByName(String name) {

for (Professor professor : professors) {

if (professor.getName().equalsIgnoreCase(name)) {

return professor; // Return the professor if found

}

}

return null; // Return null if professor not found

}

// Helper method to find a course by title

private Course findCourseByTitle(String title) {

for (Course course : courses) {

if (course.getTitle().equalsIgnoreCase(title)) {

return course; // Return the course if found

}

}

return null; // Return null if course not found

}

// Main method to interact with the system through a menu

public static void main(String[] args) {

UniversityManagement management = new UniversityManagement();

Scanner scanner = new Scanner(System.in); // Scanner for user input

boolean running = true; // Flag to control the while loop

// Loop to display the menu and process user choices

while (running) {

System.out.println("\nUniversity Management System");

System.out.println("1. Add Student");

System.out.println("2. Add Professor");

System.out.println("3. Add Course");

System.out.println("4. Enroll Student in Course");

System.out.println("5. Assign Course to Professor");

System.out.println("6. List Students");

System.out.println("7. List Professors");

System.out.println("8. List Courses");

System.out.println("9. Display Student Courses");

System.out.println("10. Display Professor Courses");

System.out.println("11. Exit");

System.out.print("Choose an option: ");

int choice = scanner.nextInt(); // Read user choice

scanner.nextLine(); // Consume the newline character

// Switch case to handle user choice

switch (choice) {

case 1:

System.out.print("Enter student name: ");

String studentName = scanner.nextLine();

management.addStudent(studentName); // Add the student

break;

case 2:

System.out.print("Enter professor name: ");

String professorName = scanner.nextLine();

management.addProfessor(professorName); // Add the professor

break;

case 3:

System.out.print("Enter course title: ");

String courseTitle = scanner.nextLine();

management.addCourse(courseTitle); // Add the course

break;

case 4:

System.out.print("Enter student name: ");

String enrollStudent = scanner.nextLine();

System.out.print("Enter course title: ");

String enrollCourse = scanner.nextLine();

management.enrollStudentInCourse(enrollStudent, enrollCourse); // Enroll student in course

break;

case 5:

System.out.print("Enter professor name: ");

String assignProfessor = scanner.nextLine();

System.out.print("Enter course title: ");

String assignCourse = scanner.nextLine();

management.assignCourseToProfessor(assignProfessor, assignCourse); // Assign course to professor

break;

case 6:

management.listStudents(); // List all students

break;

case 7:

management.listProfessors(); // List all professors

break;

case 8:

management.listCourses(); // List all courses

break;

case 9:

System.out.print("Enter student name: ");

String displayStudent = scanner.nextLine();

management.displayStudentCourses(displayStudent); // Display courses for a student

break;

case 10:

System.out.print("Enter professor name: ");

String displayProfessor = scanner.nextLine();

management.displayProfessorCourses(displayProfessor); // Display courses for a professor

break;

case 11:

running = false; // Exit the loop

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

break;

default:

System.out.println("Invalid choice. Please enter a number between 1 and 11."); // Handle invalid input

break;

}

}

scanner.close(); // Close the scanner

}

}