**Object-Oriented Programming System**

Object-Oriented Programming System (OOPS) in Java is a programming paradigm that uses **objects** and **classes** to design and develop applications.

OOPS aims to implement real-world entities like **inheritance**, **polymorphism**, **encapsulation**, and **abstraction** in programming.

Here are the key concepts of OOPS in Java:

**1. Class**:

A class is a **blueprint** for creating objects.

It defines a datatype by bundling **data** and **methods** that work on the data into one single unit.

**2. Object**:

An object is an **instance** of a class.

It is a real-world entity that has a **state** and **behavior** represented by **fields** and **methods** respectively.

**3. Inheritance**:

Inheritance is a mechanism where one class acquires the properties (fields) and behaviors (methods) of another class.

The class that inherits is called the subclass (or derived class), and the class from which it inherits is called the superclass (or base class).

**4. Polymorphism**: Polymorphism allows methods to do different things based on the object it is acting upon.

It is of two types:

* **Compile-time polymorphism (Method Overloading)**: When multiple methods in the same class have the same name but different parameters.
* **Runtime polymorphism (Method Overriding)**: When a subclass provides a specific implementation of a method that is already defined in its superclass.

**5. Encapsulation**:

Encapsulation is the technique of **wrapping** the data (variables) and the code (methods) together as a single unit.

In encapsulation, the variables of a class are **hidden from** other classes and can be accessed **only through the methods** of their current class.

This is achieved through the use of **access modifiers** (private, public, protected, default).

**6. Abstraction**:

**Abstraction** is the process of **hiding the implementation** details and **showing only the functionality**.

Abstract classes and interfaces are used to achieve **abstraction** in Java.

These concepts form the foundation of object-oriented programming in Java, enabling developers to create modular, reusable, and maintainable code.

**What is a class?**

In Java, a class is a blueprint or prototype for creating objects.

It defines a data type by bundling data and methods that work on the data into one single unit.

A class can contain fields (variables) and methods (functions) to define the properties and behaviors of objects.

**Key Components of a Class**

1. **Fields (Attributes)**: Variables that represent the state or properties of an object.
2. **Methods (Functions)**: Blocks of code that define the behavior or actions of an object.
3. **Constructors**: Special methods used to initialize objects.

**Here's a simple example of a class in Java:**

public class Car {

// Fields (attributes)

private String color;

private int speed;

// Constructor

public Car(String color, int speed) {

this.color = color;

this.speed = speed;

}

// Method to accelerate

public void accelerate() {

speed += 10;

}

// Method to brake

public void brake() {

speed -= 10;

}

// Method to display car details

public void displayDetails() {

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

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

}

}

**Creating Objects from a Class**

An object is an instance of a class.

You can create objects from the class using the **new** keyword and call its methods and access its fields.

public class Main {

public static void main(String[] args) {

// Creating an object of the Car class

Car myCar = new Car("Red", 0);

// Accessing fields and methods

myCar.accelerate();

myCar.displayDetails(); // Outputs: Car color: Red, Car speed: 10

myCar.brake();

myCar.displayDetails(); // Outputs: Car color: Red, Car speed: 0

}

}

**What are access modifiers?**

**Access modifiers** in Java are keywords that determine the visibility and accessibility of classes, methods, and other members.

They help in **encapsulating** the data and controlling how and where certain parts of a program can be accessed.

There are four main access modifiers in Java:

1. **public**
2. **protected**
3. **default** (no modifier)
4. **private**

**1. public**

* **Class Level**: A class with **public** access modifier can be accessed from any other class.
* **Member Level**: Fields, methods, or constructors declared **public** can be accessed from any other class.

public class PublicClass {

public int publicField;

public void publicMethod() {

// some code

}

}

**2. protected**

* **Class Level**: The **protected** access modifier cannot be used at the class level.
* **Member Level**: Fields, methods, or constructors declared **protected** can be accessed within the **same package** and by **subclasses in other packages**.

class BaseClass {

protected int protectedField;

protected void protectedMethod() {

// some code

}

}

public class DerivedClass extends BaseClass {

public void someMethod() {

protectedField = 10; // accessible

protectedMethod(); // accessible

}

}

**3. default (no modifier)**

* **Class Level**: A class with no access modifier is accessible only within the same package.
* **Member Level**: Fields, methods, or constructors declared with no access modifier are accessible only within the same package.

class DefaultClass {

int defaultField;

void defaultMethod() {

// some code

}

}

class AnotherClass {

public void someMethod() {

DefaultClass obj = new DefaultClass();

obj.defaultField = 10; // accessible within the same package

obj.defaultMethod(); // accessible within the same package

}

}

**4. private**

* **Class Level**: The **private** access modifier cannot be used at the class level.
* **Member Level**: Fields, methods, or constructors declared **private** can be accessed **only within the same class.**

public class PrivateClass {

private int privateField;

private void privateMethod() {

// some code

}

public void someMethod() {

privateField = 10; // accessible within the same class

privateMethod(); // accessible within the same class

}

}

public class AnotherClass {

public void someMethod() {

PrivateClass obj = new PrivateClass();

// obj.privateField = 10; // not accessible

// obj.privateMethod(); // not accessible

}

}

**Summary of Access Levels**

| **Modifier** | **Class** | **Package** | **Subclass** | **Global** |
| --- | --- | --- | --- | --- |
| **Public** | Yes | Yes | Yes | Yes |
| **Protected** | No | Yes | Yes | No |
| **Default** | Yes | Yes | No | No |
| **Private** | No | No | No | No |

Understanding and using access modifiers effectively allows you to implement proper encapsulation, which is a core principle of object-oriented programming.

**Example 01: Person Class**

This example defines a **Person** class with private fields for encapsulation, public methods to access and modify these fields, and a constructor to initialize the object.

// Define the Person class

public class Person {

// Private fields (attributes)

private String name;

private int age;

// Constructor to initialize the Person object

public Person(String name, int age) {

this.name = name;

this.age = age;

}

// Public method to get the name

public String getName() {

return name;

}

// Public method to set the name

public void setName(String name) {

this.name = name;

}

// Public method to get the age

public int getAge() {

return age;

}

// Public method to set the age

public void setAge(int age) {

if (age > 0) { // Adding a simple validation

this.age = age;

}

else {

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

}

}

// Public method to display person details

public void displayDetails() {

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

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

}

}

// Class with the main method to test the Person class

public class Main {

public static void main(String[] args) {

// Create a new Person object

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

// Access and modify fields using public methods

person.displayDetails(); // Outputs: Name: Alice, Age: 30

person.setName("Bob");

person.setAge(35);

// Display updated details

person.displayDetails(); // Outputs: Name: Bob, Age: 35

// Attempt to set an invalid age

person.setAge(-5); // Outputs: Age must be positive.

}

}

**Explanation**

1. **Class Definition**:
   * The **Person** class is defined with the **public** access modifier, making it accessible from other classes.
2. **Fields (Attributes)**:
   * The **name** and **age** fields are private, ensuring they can only be accessed and modified within the **Person** class.
3. **Constructor**:
   * The constructor **Person(String name, int age)** initializes the **name** and **age** fields when a new **Person** object is created.
4. **Methods**:
   * **getName()**, **setName(String name)**, **getAge()**, and **setAge(int age)** are public methods that provide controlled access to the private fields.
   * **displayDetails()** is a public method that prints the **Person** object's details.
5. **Encapsulation**:
   * The private fields and public getter and setter methods illustrate encapsulation, allowing controlled access to the object's properties.
6. **Main Method**:
   * The **Main** class contains the **main** method, which creates and manipulates a **Person** object to demonstrate the use of the **Person** class.

This example showcases the basic structure and usage of a class in Java, including object creation, encapsulation, and method invocation.

**Example 02: Student class**

Let's create a more detailed example with additional fields.

This example will define a **Student** class with several fields representing different attributes of a student, and it will include methods to interact with these fields.

This example defines a **Student** class with fields for the student's name, age, ID, major, and GPA.

It also includes a constructor to initialize these fields, as well as getter and setter methods for each field and toString() method.

// Define the Student class

public class Student {

// Private fields (attributes)

private String name;

private int age;

private String studentID;

private String major;

private double gpa;

// default constructor

public Student() {

}

// Constructor to initialize the Student object

public Student(String name, int age, String studentID, String major, double gpa) {

this.name = name;

this.age = age;

this.studentID = studentID;

this.major = major;

this.gpa = gpa;

}

// Public method to get the name

public String getName() {

return name;

}

// Public method to set the name

public void setName(String name) {

this.name = name;

}

// Public method to get the age

public int getAge() {

return age;

}

// Public method to set the age

public void setAge(int age) {

if (age > 0) { // Adding a simple validation

this.age = age;

} else {

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

}

}

// Public method to get the student ID

public String getStudentID() {

return studentID;

}

// Public method to set the student ID

public void setStudentID(String studentID) {

this.studentID = studentID;

}

// Public method to get the major

public String getMajor() {

return major;

}

// Public method to set the major

public void setMajor(String major) {

this.major = major;

}

// Public method to get the GPA

public double getGpa() {

return gpa;

}

// Public method to set the GPA

public void setGpa(double gpa) {

if (gpa >= 0.0 && gpa <= 4.0) { // Adding a simple validation

this.gpa = gpa;

} else {

System.out.println("GPA must be between 0.0 and 4.0.");

}

}

// Overriding the toString() method

@Override

public String toString() {

return "Name: " + name + ", Age: " + age + ", Student ID: " + studentID

+ ", Major: " + major + ", GPA: " + gpa;

}

}

// Class with the main method to test the Student class

public class Main {

public static void main(String[] args) {

// Create a new Student object

Student student = new Student("Alice", 20, "S12345", "Computer Science", 3.8);

// Access and modify fields using public methods

// Display the updated student's details using the toString() method

System.out.println(student); // Outputs: Name: Alice, Age: 20, Student ID: S12345, Major: Computer Science, GPA: 3.8

student.setName("Bob");

student.setAge(21);

student.setStudentID("S54321");

student.setMajor("Mathematics");

student.setGpa(3.9);

// Display updated details

System.out.println(student); // Outputs: Name: Bob, Age: 21, Student ID: S54321, Major: Mathematics, GPA: 3.9

// Attempt to set an invalid GPA

student.setGpa(4.5); // Outputs: GPA must be between 0.0 and 4.0.

}

}

**Explanation**

1. **Class Definition**:
   * The **Student** class is defined with the **public** access modifier, making it accessible from other classes.
2. **Fields (Attributes)**:
   * The **name**, **age**, **studentID**, **major**, and **gpa** fields are private, ensuring they can only be accessed and modified within the **Student** class.
3. **Constructor**:
   * The constructor **Student(String name, int age, String studentID, String major, double gpa)** initializes the fields when a new **Student** object is created.
4. **Getter and Setter Methods**:
   * Each field has a corresponding getter and setter method that provides controlled access to the private fields.
5. **toString() Method**:
   * The **toString()** method is overridden to return a string in the format: **Name: [name], Age: [age], Student ID: [studentID], Major: [major], GPA: [gpa]**.
6. **Main Method**:
   * The **main** method within the **Student** class creates and manipulates a **Student** object to demonstrate the use of the **Student** class and the overridden **toString()** method.

By using the **toString()** method, you can easily print a string representation of the **Student** object, which is particularly useful for debugging and logging.

**Example 03: Book class**

This time we'll define a **Book** class with both a default constructor and a parameterized constructor.

The **Book** class will have fields for the title, author, ISBN, publisher, and price.

We will also include getters and setters for each field and override the **toString()** method to provide a formatted string representation of a **Book** object.

public class Book {

// Private fields (attributes)

private String title;

private String author;

private String isbn;

private String publisher;

private double price;

// Default constructor

public Book() {

this.title = "Unknown Title";

this.author = "Unknown Author";

this.isbn = "Unknown ISBN";

this.publisher = "Unknown Publisher";

this.price = 0.0;

}

// Parameterized constructor to initialize the Book object

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

this.title = title;

this.author = author;

this.isbn = isbn;

this.publisher = publisher;

this.price = price;

}

// Getter for title

public String getTitle() {

return title;

}

// Setter for title

public void setTitle(String title) {

this.title = title;

}

// Getter for author

public String getAuthor() {

return author;

}

// Setter for author

public void setAuthor(String author) {

this.author = author;

}

// Getter for ISBN

public String getIsbn() {

return isbn;

}

// Setter for ISBN

public void setIsbn(String isbn) {

this.isbn = isbn;

}

// Getter for publisher

public String getPublisher() {

return publisher;

}

// Setter for publisher

public void setPublisher(String publisher) {

this.publisher = publisher;

}

// Getter for price

public double getPrice() {

return price;

}

// Setter for price

public void setPrice(double price) {

if (price >= 0.0) { // Adding a simple validation

this.price = price;

} else {

System.out.println("Price must be non-negative.");

}

}

// Method to calculate total price for a given quantity of books

public double calculateTotalPrice(int quantity) {

if (quantity < 0) {

System.out.println("Quantity must be non-negative.");

return -1.0; // Return a negative value to indicate error

}

return price \* quantity;

}

// Method to check if the book is expensive based on a threshold price

public boolean isExpensive(double thresholdPrice) {

return price > thresholdPrice;

}

// Overriding the toString() method

@Override

public String toString() {

return "Title: " + title + ", Author: " + author + ", ISBN: " + isbn + ", Publisher: " + publisher + ", Price: $" + price;

}

}

**BookTest.java**

public class BookTest {

public static void main(String[] args) {

// Create a new Book object using the default constructor

Book defaultBook = new Book();

// Display the default book's details using the toString() method

System.out.println(defaultBook);

// Create a new Book object using the parameterized constructor

Book book = new Book("1984", "George Orwell", "978-0451524935", "Signet Classic", 9.99);

// Display the book's details using the toString() method

System.out.println(book);

// Modify the book's details using setters

book.setTitle("Animal Farm");

book.setAuthor("George Orwell");

book.setIsbn("978-0451526342");

book.setPublisher("Signet Classic");

book.setPrice(7.99);

// Display the updated book's details using the toString() method

System.out.println(book);

// Calculate total price for a given quantity

int quantity = 5;

double totalPrice = book.calculateTotalPrice(quantity);

if (totalPrice >= 0) {

System.out.println("Total price for " + quantity + " books: $" + totalPrice);

}

// Check if the book is expensive based on a threshold price

double thresholdPrice = 10.0;

if (book.isExpensive(thresholdPrice)) {

System.out.println("The book is expensive (price > $" + thresholdPrice + ").");

}

else {

System.out.println("The book is not expensive (price <= $" + thresholdPrice + ").");

}

}

}

**Explanation**

1. **Class Definition**:
   * The **Book** class is defined with the **public** access modifier, making it accessible from other classes.
2. **Fields (Attributes)**:
   * The **title**, **author**, **isbn**, **publisher**, and **price** fields are private, ensuring they can only be accessed and modified within the **Book** class.
3. **Constructors**:
   * The default constructor **Book()** initializes the fields with default values.
   * The parameterized constructor **Book(String title, String author, String isbn, String publisher, double price)** initializes the fields with the provided values.
4. **Getter and Setter Methods**:
   * Each field has a corresponding getter and setter method that provides controlled access to the private fields.
5. **toString() Method**:
   * The **toString()** method is overridden to return a string in the format: **Title: [title], Author: [author], ISBN: [isbn], Publisher: [publisher], Price: $[price]**.
6. **Main Method**:
   * The **main** method within the **Book** class creates and manipulates **Book** objects to demonstrate the use of both the default and parameterized constructors, as well as the overridden **toString()** method.

This example demonstrates the complete definition of a class with both a default and a parameterized constructor, along with encapsulated fields and a custom **toString()** method.

**Example 04: Product class**

We'll define a **Product** class with various attributes representing a product. We'll include a default constructor, a parameterized constructor, getters and setters for each field, and an overridden **toString()** method.

public class Product {

// Private fields (attributes)

private String name;

private String description;

private String category;

private double price;

private int stockQuantity;

private String manufacturer;

private String brand;

private String color;

private String size;

private boolean isAvailable;

// Default constructor

public Product() {

this.name = "Unknown Product";

this.description = "No description available";

this.category = "Other";

this.price = 0.0;

this.stockQuantity = 0;

this.manufacturer = "Unknown Manufacturer";

this.brand = "Unknown Brand";

this.color = "Unknown";

this.size = "One Size";

this.isAvailable = false;

}

// Parameterized constructor to initialize the Product object

public Product(String name, String description, String category, double price, int stockQuantity,

String manufacturer, String brand, String color, String size, boolean isAvailable) {

this.name = name;

this.description = description;

this.category = category;

this.price = price;

this.stockQuantity = stockQuantity;

this.manufacturer = manufacturer;

this.brand = brand;

this.color = color;

this.size = size;

this.isAvailable = isAvailable;

}

// Getter and setter methods for each field

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getDescription() {

return description;

}

public void setDescription(String description) {

this.description = description;

}

public String getCategory() {

return category;

}

public void setCategory(String category) {

this.category = category;

}

public double getPrice() {

return price;

}

public void setPrice(double price) {

this.price = price;

}

public int getStockQuantity() {

return stockQuantity;

}

public void setStockQuantity(int stockQuantity) {

this.stockQuantity = stockQuantity;

}

public String getManufacturer() {

return manufacturer;

}

public void setManufacturer(String manufacturer) {

this.manufacturer = manufacturer;

}

public String getBrand() {

return brand;

}

public void setBrand(String brand) {

this.brand = brand;

}

public String getColor() {

return color;

}

public void setColor(String color) {

this.color = color;

}

public String getSize() {

return size;

}

public void setSize(String size) {

this.size = size;

}

public boolean isAvailable() {

return isAvailable;

}

public void setAvailable(boolean available) {

isAvailable = available;

}

// Business methods

public void restock(int quantity) {

stockQuantity += quantity;

}

public void sell(int quantity) {

if (stockQuantity >= quantity) {

stockQuantity -= quantity;

System.out.println("Sold " + quantity + " " + name);

} else {

System.out.println("Insufficient stock to sell " + quantity + " " + name);

}

}

// Business method to calculate the total value of stock

public double calculateTotalValue() {

return price \* stockQuantity;

}

// Business method to check if the product is in stock

public boolean isInStock() {

return stockQuantity > 0;

}

// Business method to order additional stock

public void orderStock(int quantity) {

stockQuantity += quantity;

}

// Business method to adjust the price of the product

public void adjustPrice(double newPrice) {

price = newPrice;

}

// Business method to update availability based on stock quantity

public void updateAvailability() {

isAvailable = stockQuantity > 0;

}

// Overriding the toString() method

@Override

public String toString() {

return "Name: '" + name + "', " +

"Description: '" + description + "', " +

"Category: '" + category + "', " +

"Price: $" + price + ", " +

"Stock Quantity: " + stockQuantity + ", " +

"Manufacturer: '" + manufacturer + "', " +

"Brand: '" + brand + "', " +

"Color: '" + color + "', " +

"Size: '" + size + "', " +

"Available: " + isAvailable;

}

}

**ProductTest class**

public class ProductTest {

public static void main(String[] args) {

// Create a new Product object using the default constructor

Product defaultProduct = new Product();

// Display the default product's details using the toString() method

System.out.println("Default Product:");

System.out.println(defaultProduct);

// Create a new Product object using the parameterized constructor

Product product = new Product("Laptop", "High-performance laptop with SSD", "Electronics", 999.99, 10,

"Manufacturer X", "Brand Y", "Silver", "15 inches", true);

// Display the product's details using the toString() method

System.out.println("\nProduct:");

System.out.println(product);

// Test business methods

System.out.println("\nTotal value of stock: $" + product.calculateTotalValue());

System.out.println("Is the product in stock? " + (product.isInStock() ? "Yes" : "No"));

product.orderStock(5);

System.out.println("Stock quantity after ordering 5 units: " + product.getStockQuantity());

product.adjustPrice(899.99);

System.out.println("Price after adjustment: $" + product.getPrice());

product.updateAvailability();

System.out.println("Updated availability: " + (product.isAvailable() ? "Available" : "Not Available"));

}

}

**Explanation**

1. **Product Class**:
   * The **Product** class represents a product with various attributes such as name, description, category, price, stock quantity, manufacturer, brand, color, size, and availability.
   * It provides constructors to initialize the product with default values or with specific values.
   * It includes getter and setter methods for each attribute to control access to the fields and facilitate data manipulation.
   * Additionally, it contains business methods such as **calculateTotalValue()**, **isInStock()**, **orderStock()**, **adjustPrice()**, and **updateAvailability()**.
2. **ProductTest Class**:
   * The **ProductTest** class serves as a test harness for the **Product** class.
   * In the **main()** method of **ProductTest**, we:
     + Create instances of **Product** using both the default and parameterized constructors.
     + Display the details of the products using the **toString()** method.
     + Test the business methods of the **Product** class:
       - **calculateTotalValue()**: Calculates the total value of the stock.
       - **isInStock()**: Checks if the product is in stock.
       - **orderStock()**: Orders additional stock.
       - **adjustPrice()**: Adjusts the price of the product.
       - **updateAvailability()**: Updates the availability based on the stock quantity.
   * Each step of the testing process is accompanied by relevant output statements to display the results.
3. **Output**:
   * The output of the program demonstrates the functionality of the **Product** class and its business methods.
   * It includes details of the default product, the parameterized product, the total value of the stock, the availability status, the stock quantity after ordering additional units, the updated price, and the updated availability.

Overall, the **ProductTest** program showcases how to create and manipulate instances of the **Product** class, utilize its business methods, and verify the results through output statements. It serves as a comprehensive test suite for the **Product** class's functionality.

**Example 05**

Here's a separate class named **ProductManager** with a **main()** method where an array of **Product** objects is used:

public class ProductManager {

public static void main(String[] args) {

// Create an array of Product objects

Product[] products = new Product[3];

// Populate the array with Product objects

products[0] = new Product("Laptop", "High-performance laptop with SSD", "Electronics", 999.99, 10,

"Manufacturer X", "Brand Y", "Silver", "15 inches", true);

products[1] = new Product("Smartphone", "Flagship smartphone with dual cameras", "Electronics", 699.99, 20,

"Manufacturer Z", "Brand X", "Black", "6 inches", true);

products[2] = new Product("Headphones", "Wireless headphones with noise cancellation", "Electronics", 199.99, 15,

"Manufacturer A", "Brand B", "White", "One Size", true);

// Display details of each product in the array

System.out.println("Product Details:");

for (Product product : products) {

System.out.println(product);

}

// Test business methods on a product from the array

Product sampleProduct = products[1];

System.out.println("\nSample Product: " + sampleProduct.getName());

System.out.println("Total value of stock: $" + sampleProduct.calculateTotalValue());

System.out.println("Is the product in stock? " + (sampleProduct.isInStock() ? "Yes" : "No"));

sampleProduct.orderStock(5);

System.out.println("Stock quantity after ordering 5 units: " + sampleProduct.getStockQuantity());

sampleProduct.adjustPrice(649.99);

System.out.println("Price after adjustment: $" + sampleProduct.getPrice());

sampleProduct.updateAvailability();

System.out.println("Updated availability: " + (sampleProduct.isAvailable() ? "Available" : "Not Available"));

}

}

In **main()** method:

* An array of **Product** objects is created with a size of 3.
* Each element of the array is initialized with different **Product** objects using the parameterized constructor.
* The details of each product in the array are displayed using the **toString()** method.
* Business methods of the **Product** class are tested on one of the products from the array (**sampleProduct**). The methods tested include **calculateTotalValue()**, **isInStock()**, **orderStock()**, **adjustPrice()**, and **updateAvailability()**.
* Output statements are used to display the results of each test.

This **ProductManager** class demonstrates how to work with an array of **Product** objects and test their functionality using business methods.

**Example 06**

Let's modify the **ProductManager** class to use an array of **Product** objects and create products dynamically within the loop:

import java.util.Scanner;

public class ProductManager {

public static void main(String[] args) {

// Scanner object for user input

Scanner scanner = new Scanner(System.in);

// Array to store Product objects

Product[] products = new Product[5]; // Assuming a maximum of 5 products

// Index to keep track of the next available position in the array

int index = 0;

// Loop for user interaction

boolean exit = false;

do {

System.out.println("\nOptions:");

System.out.println("1. Create Product");

System.out.println("2. Display Product Details");

System.out.println("3. Test Business Methods");

System.out.println("4. Exit");

System.out.print("Enter your choice: ");

int choice = scanner.nextInt();

switch (choice) {

case 1:

// Create a new Product object

if (index < products.length) {

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

String name = scanner.next();

System.out.print("Enter product description: ");

String description = scanner.next();

System.out.print("Enter product category: ");

String category = scanner.next();

System.out.print("Enter product price: ");

double price = scanner.nextDouble();

System.out.print("Enter product stock quantity: ");

int stockQuantity = scanner.nextInt();

System.out.print("Enter product manufacturer: ");

String manufacturer = scanner.next();

System.out.print("Enter product brand: ");

String brand = scanner.next();

System.out.print("Enter product color: ");

String color = scanner.next();

System.out.print("Enter product size: ");

String size = scanner.next();

System.out.print("Enter product availability (true/false): ");

boolean isAvailable = scanner.nextBoolean();

// Create a Product object with user input

products[index] = new Product(name, description, category, price, stockQuantity,

manufacturer, brand, color, size, isAvailable);

System.out.println("Product created successfully: " + products[index]);

index++;

} else {

System.out.println("Maximum limit reached. Cannot add more products.");

}

break;

case 2:

// Display details of each product in the array

System.out.println("\nProduct Details:");

for (int i = 0; i < index; i++) {

System.out.println(products[i]);

}

break;

case 3:

// Test business methods on a sample product (if any)

if (index > 0) {

Product sampleProduct = products[0];

System.out.println("\nSample Product: " + sampleProduct.getName());

System.out.println("Total value of stock: $" + sampleProduct.calculateTotalValue());

System.out.println("Is the product in stock? " + (sampleProduct.isInStock() ? "Yes" : "No"));

sampleProduct.orderStock(5);

System.out.println("Stock quantity after ordering 5 units: " + sampleProduct.getStockQuantity());

sampleProduct.adjustPrice(649.99);

System.out.println("Price after adjustment: $" + sampleProduct.getPrice());

sampleProduct.updateAvailability();

System.out.println("Updated availability: " + (sampleProduct.isAvailable() ? "Available" : "Not Available"));

} else {

System.out.println("No products created yet.");

}

break;

case 4:

exit = true;

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

break;

default:

System.out.println("Invalid choice. Please enter a valid option.");

}

} while (!exit);

// Close the scanner

scanner.close();

}

}

**Explanation:**

* An array **products** of **Product** objects is initialized with a maximum size of 5 (you can adjust this as needed).
* The variable **index** keeps track of the next available position in the array to add a new product.
* When the user chooses to create a product, if there's space available in the array, the user is prompted to enter details for the new product, and it's added to the array.
* The option to display product details now iterates through the array up to the current **index** value to print only the products that have been created.
* Similarly, the option to test business methods (option 3) operates on the first product in the array, if any products have been created.

**What is an object?**

In Java, an object is an instance of a class.

Objects are the fundamental building blocks of object-oriented programming (OOP).

They encapsulate data and behavior together.

Here's a more detailed explanation:

**Key Concepts of an Object:**

1. **State (Attributes/Fields):**
   * An object holds data, which is represented by attributes or fields.
   * These fields store the object's current state.
   * Example: For a **Car** object, fields might include **color**, **model**, **year**, and **speed**.
2. **Behavior (Methods):**
   * An object has behavior, which is defined by methods. These methods define the operations or functions that the object can perform.
   * Example: For a **Car** object, methods might include **drive()**, **brake()**, **accelerate()**, and **honk()**.
3. **Identity:**
   * Every object has a unique identity, which distinguishes it from other objects, even if they have the same state. The identity is typically represented by the memory address where the object is stored.

**Example of an Object in Java:**

Consider a simple **Person** class and its objects:

public class Person {

// Fields (attributes)

private String name;

private int age;

// Constructor

public Person(String name, int age) {

this.name = name;

this.age = age;

}

// Method (behavior)

public void displayInfo() {

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

}

// Getters and Setters

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

}

public class Main {

public static void main(String[] args) {

// Creating objects of the Person class

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

Person person2 = new Person("Bob", 25);

// Using methods on the objects

person1.displayInfo(); // Output: Name: Alice, Age: 30

person2.displayInfo(); // Output: Name: Bob, Age: 25

// Changing the state of an object

person1.setAge(31);

person1.displayInfo(); // Output: Name: Alice, Age: 31

}

}

**Explanation:**

* **Class Definition:**
  + **Person** class defines the blueprint for creating **Person** objects.
  + It contains fields **name** and **age** to represent the state of a person.
  + It includes a constructor to initialize these fields when a new **Person** object is created.
  + It has a method **displayInfo()** to display the information of a person.
  + It also includes getter and setter methods for accessing and modifying the fields.
* **Creating Objects:**
  + **Person person1 = new Person("Alice", 30);** creates a new object **person1** of type **Person** with the name "Alice" and age 30.
  + **Person person2 = new Person("Bob", 25);** creates another object **person2** with the name "Bob" and age 25.
* **Using Methods:**
  + The **displayInfo()** method is called on **person1** and **person2** to display their information.
  + The **setAge()** method is used to change the age of **person1** to 31, demonstrating how the state of an object can be modified.

Objects in Java enable the creation of reusable and modular code by encapsulating data and behavior together, which aligns with the principles of object-oriented programming.

**What is static keyword?**

The **static** keyword in Java is used to indicate that a particular member belongs to the type itself, rather than to instances of that type.

This can be applied to variables, methods, blocks, and nested classes.

Let's explore each usage in detail:

**1. Static Variables (Class Variables)**

A static variable is shared among all instances of a class.

It belongs to the class, not to any specific instance.

**Example:**

public class Counter {

public static int count = 0; // static variable

public Counter() {

count++; // increment the static variable

}

}

public class Main {

public static void main(String[] args) {

Counter c1 = new Counter();

Counter c2 = new Counter();

Counter c3 = new Counter();

System.out.println(Counter.count); // Output: 3

}

}

In this example, **count** is a static variable that is incremented every time a new **Counter** object is created.

Since it is static, all instances share the same **count** variable.

**2. Static Methods**

A static method belongs to the class rather than to any specific instance. It can be called without creating an instance of the class. Static methods can only access other static members (variables and methods) directly.

**Example:**

public class MathUtil {

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

return a + b;

}

}

public class Main {

public static void main(String[] args) {

int sum = MathUtil.add(5, 3); // Calling the static method

System.out.println("Sum: " + sum); // Output: Sum: 8

}

}

In this example, **add** is a static method that can be called directly using the class name **MathUtil** without creating an instance of the class.

**3. Static Blocks**

Static blocks are used to initialize static variables.

They are executed when the class is loaded into memory.

**Example:**

public class StaticBlockExample {

public static int num;

static {

num = 10;

System.out.println("Static block executed.");

}

public static void main(String[] args) {

System.out.println("Value of num: " + StaticBlockExample.num); // Output: Static block executed. Value of num: 10

}

}

In this example, the static block initializes the static variable **num** and prints a message when the class is loaded.

**4. Static Nested Classes**

A static nested class is a nested class that is declared static.

It can access static members of the enclosing class, but it cannot access non-static members without an instance of the enclosing class.

**Example:**

public class OuterClass {

static int outerStaticVar = 10;

static class NestedStaticClass {

void display() {

System.out.println("Outer static variable: " + outerStaticVar);

}

}

public static void main(String[] args) {

OuterClass.NestedStaticClass nested = new OuterClass.NestedStaticClass();

nested.display(); // Output: Outer static variable: 10

}

}

In this example, **NestedStaticClass** is a static nested class that can access the static variable **outerStaticVar** of the enclosing class **OuterClass**.

**Summary**

* **Static Variables**: Shared among all instances of the class.
* **Static Methods**: Can be called without creating an instance of the class.
* **Static Blocks**: Used for initializing static variables; executed when the class is loaded.
* **Static Nested Classes**: Can access static members of the enclosing class; do not require an instance of the enclosing class.

The **static** keyword is useful for defining class-level members that should be shared among all instances, or utility methods that don't depend on instance variables.

**Constructor:**

1. Constructor is a method.
2. Constructor should have the name as class name.
3. Constructor should **not return any value** including **void**.
4. Constructor executes when we create object.
5. Constructor can be overloaded.
   1. Overloaded means it can have more than one method with the same name.
   2. Overloading means the difference should be in case of parameters(count, type).
   3. Count: the number of parameters
   4. Type: if number of parameters are same then it will find change in the data type of the parameter.

Overloading:

1. The class can have more than one method with the same name.
2. The difference should be case of parameters count or parameter’s type.

What is inheritance?

Deriving a new class from an existing class.

In java if you have not inherited a class means that class automatically inherits from the class named Object.

The class Object is part of JDK.

**Overriding:**

If you have defined a method in the class which is already defined in the base (parent) class.

Base (2)

method1()

method2()

Derived (4)

method3()

method4()

method2() (overriding in the derived class)

Object class will be having some methods. Out of those we need override the toString() method.

We can do overriding by using @Override annotation.

To override toString() method in the derived class

@Override

public String toString() {

return “Something”;

}

There is no need to call toString() method explicitly.