Unit-02

Introducing Classes, Objects and Methods

2.1 Class Fundamentals

In Java, a class is a blueprint for creating objects. It defines the attributes (fields) and behaviors (methods) that the objects created from the class will have. It encapsulates data for the object and methods to manipulate that data.

Class Declaration: A class is declared using the class keyword followed by the class name. The class body is enclosed in curly braces {}.

Fields: Fields (also known as instance variables) are variables that hold the state of an object. They are declared inside the class but outside any method.

Methods: Methods define the behaviors of an object. They are functions that are defined inside a class and operate on the fields of the class.

Constructors: Constructors are special methods that are called when an object is instantiated. They initialize the object's fields. A constructor has the same name as the class and no return type.

Creating Objects: Objects are instances of a class. They are created using the new keyword followed by a call to the constructor.

Access Modifiers: Access modifiers determine the visibility of classes, fields, and methods. The main access modifiers are public, private, protected, and the default (package-private).

Static Members: The static keyword is used to declare fields and methods that belong to the class, rather than to any specific instance. Static members can be accessed without creating an instance of the class.

Explain the principles of the object-oriented programming

Object-oriented programming (OOP) is a programming model based on the concept of "objects," which can contain data and code to manipulate that data. The four fundamental principles of OOP are encapsulation, inheritance, polymorphism, and abstraction.

1. Encapsulation

Encapsulation is the principle of bundling the data (fields) and methods (functions) that operate on the data into a single unit or class. It restricts direct access to some of an object's components, which is a means of preventing accidental interference and misuse of the data. Encapsulation is achieved through access modifiers (private, protected, public).

2. Inheritance

Inheritance is the mechanism by which one class (subclass or derived class) can inherit fields and methods from another class (superclass or base class). This promotes code reuse and establishes a natural hierarchical relationship between classes.

3. Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common super class. It is most often expressed as the ability of different classes to respond to the same message (method call) in different ways, typically through method overriding or method overloading.

4. Abstraction

Abstraction is the principle of hiding the complex implementation details and showing only the necessary features of an object. It simplifies the complexity of the system by allowing the programmer to focus on interactions at a higher level.

2.2 Object Creation

In Java, an object is created from a class using the 'new' keyword, which allocates memory for the object and calls a constructor to initialize the object.

Steps to Create an Object

1. **Declare a Reference Variable**: This step creates a variable of the class type, which will hold the reference to the object.

   ```
   ClassName referenceVariable;
   ```

2. **Instantiate the Object**: This step uses the new keyword followed by a call to the class constructor to create an object.

   ```
   referenceVariable = new ClassName();
   ```

3. **Combine the Declaration and Instantiation**: Typically, the declaration and instantiation are combined into a single step.

   ```
   ClassName referenceVariable = new ClassName();
   ```

Create programs with methods, constructors, nested and inner classes

```
public class University {
    private String name;

    // Constructor
    public University(String name) {
        this.name = name;
    }

    // Method to display university details
    public void displayUniversity() {
        System.out.println("University: " + name);
    }

    // Static nested class Department
    public static class Department {
        private String deptName;

        // Constructor
        public Department(String deptName) {
            this.deptName = deptName;
        }

        // Method to display department details
        public void displayDepartment() {
            System.out.println("Department: " + deptName);
        }
        // Inner class Student
        public class Student {
            private String studentName;
            private int studentId;

            // Constructor
            public Student(String studentName, int studentId) {
                this.studentName = studentName;
                this.studentId = studentId;
            }

            // Method to display student details
            public void displayStudent() {
                System.out.println("Student Name: " + studentName + ", ID: " + studentId);
            }
        }
    }

    public static void main(String[] args) {
        // Create a University object
        University university = new University("ABC University");
        university.displayUniversity();

        // Create a Department object
        University.Department department = new University.Department("Computer Science");
        department.displayDepartment();
        // Create a Student object
        University.Department.Student student = department.new Student("John Doe", 12345);
        student.displayStudent();
    }
}
```

2.3 Methods

In Java, a method is a block of code within a class that performs a specific task. Methods are used to execute a sequence of statements, encapsulate code for reuse, and make programs more modular and manageable.

Key Features of Methods

4. **Encapsulation**: Methods help in encapsulating the functionality within a class, promoting modularity.
5. **Reusability**: Methods allow code reuse, making it easy to call the same block of code multiple times.
6. **Parameters**: Methods can accept input parameters to process data.
7. **Return Type**: Methods can return a value after execution.

Syntax:

```
public returnType methodName(parameterList)
{
   // Method body
}
```

2.4 Command Line Arguments

Command line arguments in Java are used to pass information into a program when it is executed. They allow users to specify configuration options or input data directly from the command line without changing the code. These arguments are passed to the main method as an array of String objects.

```
public class CommandLineArgument {
    public static void main(String[] args) {
        // Check if an argument is provided
        if (args.length == 0) {
            System.out.println("Please provide an argument.");
            return;
        }

        // Print the provided argument
        System.out.println("Argument provided: " + args[0]);
    }
}
```

2.5 Constructors

In Java, constructors are special methods used for initializing objects. They have the same name as the class and do not have a return type. When an object is created using the new keyword, a constructor is invoked to initialize the object's state.

Key Features of Constructors

8. **Initialization**: Constructors initialize the newly created object.
9. **Same Name as Class**: Constructors have the same name as the class they belong to.
10. **No Return Type**: Constructors do not have a return type, not even void.
11. **Overloading**: Like other methods, constructors can be overloaded, allowing multiple constructors with different parameter lists.
12. **Implicit Default Constructor**: If no constructor is explicitly defined, Java provides a default constructor with no parameters.

2.6 Garbage Collection

Garbage collection is a feature in Java that automatically manages memory by reclaiming memory occupied by objects that are no longer reachable or in use by the program. It prevents memory leaks and allows developers to focus on writing code without worrying about memory management.

2.7 This keyword

The "this' keyword in Java is a reference variable that refers to the current object. It is used within an instance method or a constructor to refer to the current object whose method or constructor is being invoked. The 'this' keyword is particularly useful for differentiating between instance variables and parameters with the same name and for invoking other constructors in the same class.

Key Uses of this Keyword

→ **Referencing Instance Variables**: When instance variables and parameters have the same name, this is used to distinguish between them.

→ **Invoking Current Class Methods**: this can be used to invoke methods of the current class.

→ **Calling Other Constructors**: this() can be used to call another constructor in the same class. This is known as constructor chaining.

→ **Returning the Current Object**: this can be used to return the current instance from a method.

→ **Passing the Current Object as a Parameter**: this can be passed as an argument in the method call.

2.8 Static Fields and Methods

In Java, the static keyword is used to indicate that a particular member (field or method) belongs to the class itself rather than to instances of the class. This means that static fields and methods can be accessed without creating an instance of the class.

Static Fields

Static fields, also known as class variables, are shared among all instances of a class. There is only one copy of a static field, regardless of how many objects of the class are created.

Static Methods

Static methods, also known as class methods, can be called on the class itself rather than on instances of the class. They can only directly access other static fields and methods in the same class.

2.10 Variable Length Arguments

Variable Length Arguments (varargs) in Java allow methods to accept zero or more arguments of a specified type. This feature provides flexibility and convenience, enabling methods to handle a variable number of inputs without overloading methods for different numbers of parameters.

Key Features of Varargs

13. **Syntax**: Varargs are declared by appending an ellipsis (...) to the parameter type.
14. **Usage**: Inside the method, varargs behave like an array of the specified type.
15. **Single Varargs Parameter**: A method can only have one varargs parameter, and it must be the last parameter in the method's parameter list.

```
public class VarargsExample {



   // Method with variable length arguments

   public static void printNumbers(int... numbers) {

      for (int number : numbers) {

         System.out.print(number + " ");

      }

      System.out.println();

   }



   public static void main(String[] args) {

      // Calling the method with different numbers of arguments

      printNumbers(1, 2, 3);      // Output: 1 2 3

       printNumbers(4, 5);

      printNumbers(6);

      printNumbers();        } }
```