# Chapter 28:Mastering Inner Classes and Interfaces in Java: A Comprehensive Guide

Rohit Gawande


## Introduction

In this post, I, **Rohit Gawande**, will take you on a journey through **inner classes** and **interfaces** in Java. We will start by revisiting the core concepts of **Object-Oriented Programming (OOP)**, which are fundamental to understanding the structure and behavior of inner classes and interfaces. Along the way, I'll use detailed examples, memory maps, tables, and visual aids to ensure you grasp every concept. By the end of this post, you'll have a solid understanding of how to work with inner classes and interfaces in Java. Let's dive right in!

## Section 1: Revisiting Object-Oriented Programming (OOP)

Before we dive into **inner classes** and **interfaces**, let's quickly revise the core OOP concepts that underpin these topics.

## 1.1 Key Concepts of OOP

- **Encapsulation**:
  Bundling data (variables) and methods that operate on the data into a single unit (class). It allows you to protect your data by making it private and controlling its access via public methods (getters and setters).

- **Inheritance**:
  The mechanism by which one class inherits properties and behaviors from another. It promotes code reuse and allows classes to be extended for specialization.

- **Polymorphism**:
  The ability to treat objects of different classes as if they were objects of a common superclass. It supports method overloading and overriding, enabling flexibility in handling objects.

- **Abstraction**:
  Hiding the implementation details and showing only the essential features of an object. It allows you to define a blueprint (via abstract classes or interfaces) without worrying about the actual implementation.

## 1.2 Object, Variables, and Methods

In OOP, an object can use instance and local variables:

- **Instance variables** belong to the object and are tied to it.

- **Local variables** are created within methods and exist only during the method's execution.

If you want a variable to be accessible by all objects, you make it **static**. Static variables belong to the class itself rather than instances of it. We've already learned that instance variables are often made **private** to encapsulate data, and they can be accessed externally using **getters and setters** (public methods).

Instance variables can be initialized either via a **constructor** or a **setter method**, depending on when you want to set the value:

**Constructor Initialization:**

- **Use Case:** When you need to set the initial state of an object,Use a constructor when you need the value as soon as the object is created.

- **Advantages:** Ensures that an object is created with a valid state.

**Setter Initialization:**

- **Use Case:** Use a setter if you want to set the value later,When you need to modify the state of an object after creation.
- **Advantages:** Provides flexibility to change the state as needed.

## Constructors vs. Setters

Instance variables can be initialized using constructors or setters. Here's a comparison:

| Feature | Constructors | Setters |
|---|---|---|
| **Initialization Time** | Initialize at the time of object creation. | Initialize after object creation. |
| **Purpose** | Used for mandatory initialization. | Used for optional or late initialization. |
| **Best Use Case** | When values must be provided upon creation. | When values may change over time. |

## 1.2 **Example Code**: OOP Concepts in Action

```
// Demonstrating Encapsulation
class Car {
    private String model;

    // Getter for model
    public String getModel() {
        return model;
    }

    // Setter for model
    public void setModel(String model) {
        this.model = model;
    }
}

// Demonstrating Inheritance
class SportsCar extends Car {
    private int speed;

    // Getter and Setter for speed
    public int getSpeed() {
        return speed;
    }

    public void setSpeed(int speed) {
        this.speed = speed;
    }
}

// Demonstrating Polymorphism
class Test {
    public static void main(String[] args) {
        Car myCar = new SportsCar(); // Polymorphism: Treating SportsCar as Car
        myCar.setModel("Ferrari");
        System.out.println("Car Model: " + myCar.getModel());
    }
}
```

### Memory Map:

| Object | Memory Address | Instance Variables (Values) |
|---|---|---|
| `myCar` | `0x001` | `model = Ferrari` |

In the example above, the **SportsCar** class inherits the properties of the **Car** class, demonstrating inheritance. We use polymorphism by declaring `myCar` as a `Car` type while initializing it as a `SportsCar`. This allows the **Car** methods to be

used while hiding the specific details of **SportsCar**.

### Visual Representation

```
javaCopy codeClass A
  |
  +-- Instance Variables (Private)
  |      |
  |      +-- Getter and Setter (Public)
  |
  +-- Static Variable (Shared across all instances)
  |
  +-- Constructor or Setter (Initialization)
```

## Section 2:Understanding `System.out` and `PrintStream` in Java

In Java, `System.out` is a familiar way to print text to the console, but understanding the underlying mechanics can provide deeper insights into how Java handles output.

### What is `System.out`?

- `System` Class: `System` is a utility class in the `java.lang` package. It provides various methods and fields for interacting with system-level resources.
- `out` Field: `System.out` is a static field of the `System` class. This field is an instance of the `PrintStream` class, which is used for output operations.

### The `PrintStream` Class

- **Purpose**: `PrintStream` is a class in the `java.io` package designed to handle output operations. It provides methods to print various types of data, including strings, integers, and floating-point numbers.
- **Key Methods**: It includes methods such as `print()`, `println()`, and `printf()`, which allow for formatted output.

### Code Explanation

Consider the following code snippet:

```
PrintStream out = System.out;
out.println("hello");
```

Here's a step-by-step breakdown:

1. **Assignment**:

   ```
   PrintStream out = System.out;
   ```

   - This line assigns the `System.out` field to a new variable named `out`.
   - `out` is now a reference to the same `PrintStream` object that `System.out` points to.

2. **Printing**:

   ```
   out.println("hello");
   ```

   - The `println` method of the `PrintStream` class is called on the `out` object.
   - Since `out` is essentially `System.out`, this call prints the string `"hello"` to the console.

### Why Use `System.out`?

- **Standard Output**: `System.out` is the default output stream for the console. It is automatically initialized when the Java Virtual Machine (JVM) starts.
- **Convenience**: Using `System.out` directly is convenient for simple applications and debugging.

### Example Code

To illustrate:

```
PrintStream out = System.out; // Create a reference to the standard output stream
out.println("hello"); // Print "hello" to the console
```

**Output**:

```
hello
```

In this code, `out` and `System.out` are interchangeable. Both refer to the same `PrintStream` instance, so the output is identical regardless of which one you use.

## Conclusion

The `System.out` field is a widely used mechanism for printing output in Java. By understanding that `System.out` is a `PrintStream` object, you gain insight into how Java handles console output. The `PrintStream` class's methods, such as `println`, provide a simple way to produce formatted text output.

## Understanding Static and Non-Static Methods in Java

Before diving into inner classes, let's review a basic Java class with static and non-static methods. This will help in understanding how inner classes work.

### Example Code

```
public class First {
    int num; // Non-static variable

    public void show() {
        System.out.println("In show method: " + num); // Non-static method
    }

    public static void main(String[] args) {
        // If we directly call show() here, it will throw an error.
        // We can't call a non-static method from a static context directly.

        First obj = new First(); // Creating an instance of First
        obj.show(); // Calling the non-static method show() on the instance obj
    }
}
```

### Explanation

1. **Non-Static Methods and Variables**:
    - **Non-Static Variable**: `int num` is a non-static (instance) variable. It belongs to a specific instance of the `First` class.
    - **Non-Static Method**: `public void show()` is a non-static method. It can access instance variables and other instance methods. To call this method, you need an instance of the class.
2. **Static Methods and Variables**:
    - **Static Context**: `public static void main(String[] args)` is a static method. Static methods belong to the class itself rather than any instance of the class.
    - **Calling Non-Static Methods**: You cannot directly call a non-static method from a static method without an instance. This is because static methods do not have access to instance-specific data.

3. **Creating an Instance**:
    - To call the non-static `show()` method from the static `main` method, you need to create an instance of the `First` class using `First obj = new First();`.
    - Once you have an instance (`obj`), you can call `obj.show();` to execute the non-static method.

### Key Points

- **Static Methods**: Cannot directly access non-static methods or variables. They can only access other static methods and variables.
- **Non-Static Methods**: Can access both static and non-static methods and variables. They need an instance of the class to be called.

## Section 2: Introduction to Inner Classes in Java

An **Inner Class** is a class defined within another class. It provides a way to logically group classes that are only used in one place, increasing encapsulation and improving code readability. There are several types of inner classes, and we will cover each with detailed code examples, memory maps, and tables.

### 2.1 Why Use Inner Classes?

1. **Logical Grouping**: Inner classes allow for the grouping of related classes in a way that makes sense contextually.
2. **Encapsulation**: Inner classes can access the private members of the outer class, providing a tighter control of access.
3. **Scoping**: Inner classes allow you to limit the scope of a class definition to the class that uses it.

---

### 2.2 Types of Inner Classes

Java provides four types of inner classes:

1. **Non-Static Inner Class**
2. **Static Inner Class**
3. **Local Inner Class**

4. **Anonymous Inner Class**

Let's explore each of them in detail.

---

## 2.2.1. Non-Static Inner Class

A **non-static inner class** is associated with an instance of the outer class. This means that you need to create an object of the outer class before creating an object of the inner class.

### Understanding Non-Static Inner Classes in Java

In Java, non-static inner classes (also known as instance inner classes) are tied to an instance of the outer class. This means that you need to create an instance of the outer class before you can create an instance of the inner class.

#### Example Code

```
public class First {
    int num; // Instance variable of the outer class
    A obj1 = new A(); // Creating an instance of the inner class

    public void show() {
        System.out.println("In show method: " + num);
    }

    class A { // Non-static inner class
        public void config() {
            System.out.println("In config method");
            show(); // Accessing outer class method directly
        }
    }

    public static void main(String[] args) {
        First obj = new First(); // Creating an instance of the outer class
        obj.show(); // Calling the outer class method
        A innerObj = obj.new A(); // Creating an instance of the inner class
        innerObj.config(); // Calling the inner class method
    }
}
```

#### Memory Map

When the Java compiler processes this code, it creates a separate `.class` file for each class.

1. **Outer Class (`First`)**:
    - Contains instance variables (`num`, `obj1`).
    - Contains instance methods (`show()`, `main()`).
    - Contains the non-static inner class `A`.
2. **Inner Class (`A`)**:
    - Can access outer class instance variables and methods.
    - Exists in a separate `.class` file with a name that includes the outer class name (e.g., `First$A.class`).

**Memory Layout:**

| Object | Description |
| --- | --- |
| `First` | Instance of the outer class |
| `num` | Instance variable of `First` |
| `obj1` | Instance of the inner class `A` within `First` |
| `A` | Inner class `A`, has access to `First`'s fields |
| `innerObj` | Instance of the inner class `A` |

**Table: Compilation Output**

When you compile `First.java`, the following `.class` files are generated:

| .java File | .class File |
| --- | --- |
| `First.java` | `First.class` |
| | `First$A.class` (for the inner class `A`) |

**Why is the inner class file named differently?**

- The inner class file is named `First$A.class` to indicate that it is an inner class of `First`. The `$` symbol is used to separate the outer class name from the inner class name.

- This naming convention helps in identifying the inner class in relation to its outer class.

### Key Points

1. **Creating Inner Class Instances**:
    - To create an instance of a non-static inner class, you first need an instance of the outer class. In the code:

      ```
      First obj = new First(); // Create outer class instance
      A innerObj = obj.new A(); // Create inner class instance
      ```

    - This is because the inner class `A` has an implicit reference to the instance of the outer class `First`.
2. **Accessing Methods**:
    - Non-static inner classes can access all members (including private members) of the outer class.
    - Methods of the outer class can be called directly from within the inner class (e.g., `show()` method in the `config()` method).
3. **Compilation Details**:
    - Each class, including the inner class, is compiled into a separate `.class` file.
    - The inner class file name includes the outer class name, which helps in maintaining the relationship between the outer and inner classes.

Let's break down the code you provided and explain the concepts related to static and non-static inner classes, including why an outer class cannot be static.

## Code Explanation

```
    package innerClass;

    class A {
        int a = 1;

        public void disp2() {
            System.out.println("a = " + a);
        }

        class B {
            int b = 2;

            public void disp3() {
                System.out.println("b = " + b);
            }
        }
    }

    public class Basic {
        public static void main(String[] args) {
            A b = new A(); // Create an instance of outer class A
            b.disp2(); // Call method of outer class A

            A.B obj; // Declare a reference to inner class B
            obj = b.new B(); // Create an instance of inner class B using the
outer class instance
            obj.disp3(); // Call method of inner class B
        }
    }
```

## Explanation

1. **Outer Class** `A`:
    - **Instance Variable**: `int a = 1;`
    - **Instance Method**: `public void disp2() {...}`
2. **Inner Class** `B`:
    - **Instance Variable**: `int b = 2;`
    - **Instance Method**: `public void disp3() {...}`

3. **Creating Instances**:
    - **Outer Class Instance**: `A b = new A();` creates an instance of the outer class `A`.
    - **Calling Outer Class Method**: `b.disp2();` calls the method `disp2()` of the outer class `A`.
    - **Inner Class Instance**: `A.B obj = b.new B();` creates an instance of the non-static inner class `B` using the outer class instance `b`.
    - **Calling Inner Class Method**: `obj.disp3();` calls the method `disp3()` of the inner class `B`.

## Non-Static Inner Classes

### 1. Non-Static Inner Classes:

- Non-static inner classes (like `B` in the example) are associated with an instance of the outer class.
- To create an instance of a non-static inner class, you need an instance of the outer class. This is done using the syntax `outerInstance.new InnerClass()`, where `outerInstance` is an instance of the outer class.

**Example:**

```
A outer = new A();
A.B inner = outer.new B();
```

**Why?** Non-static inner classes have an implicit reference to the instance of the outer class. This allows them to access the outer class's members directly.

### Why Outer Classes Cannot Be Static

- **Static Context**: The concept of `static` is associated with the class itself, not instances. An outer class cannot be static because it exists independently and is not tied to any instance of another class.
- **Instance Association**: An outer class is a blueprint for creating instances, and the idea of it being static doesn't align with its role. Static is relevant for inner classes to manage their independence from outer class instances.

## Summary

- **Non-static Inner Classes**: Require an instance of the outer class to be created because they hold an implicit reference to that instance.
- **Outer Classes**: Cannot be static as they are the primary class structure for creating instances and defining the static context.

Non-static inner classes are closely associated with instances of the outer class and can access its members directly. To create an instance of a non-static inner class, you first need an instance of the outer class. The Java compiler generates separate `.class` files for both the outer and inner classes, with the inner class file named in a way that reflects its association with the outer class.

---

## 2.2.2 Static Inner Class

A **static inner class** is associated with the outer class itself rather than an instance of the outer class. You can instantiate the inner class without creating an instance of the outer class.

- Static inner classes (also known as nested static classes) are not tied to a specific instance of the outer class.

- You can create an instance of a static inner class without needing an instance of the outer class. Instead, you use the outer class name directly.

  **Example:**

```
class A {
    static class C {
        void disp() {
            System.out.println("Static Inner Class");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        A.C obj = new A.C(); // Create an instance of static inner class C
        obj.disp(); // Call method of static inner class C
    }
}
```

  Summary:-

  - **Why?** Static inner classes do not have an implicit reference to the outer class instance, so they can be created independently. They can only access the static members of the outer class.
  - **Static Inner Classes**: Do not require an outer class instance for creation and can be instantiated using the outer class name directly.

---

## 2.2.3 Local Inner Class

A **local inner class** is a class defined inside a method of the outer class. It can only be instantiated within the method where it is defined.

```
class OuterClass {
    void myMethod() {
        class InnerClass {
            void display() {
                System.out.println("Inside Local Inner Class");
            }
        }

        InnerClass inner = new InnerClass();
        inner.display();
    }

    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        outer.myMethod();
    }
}
```

**Explanation**:

> The `InnerClass` is defined within the method `myMethod()`. It cannot be accessed outside this method, making it local to the method.

## Method Overriding in Java

**Concept Overview:**

Method overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass. This feature enables runtime polymorphism and allows subclasses to modify or extend the behavior of methods inherited from their parent classes.

**Example Code:**

```
// Superclass
class Computer {
    public void  config()  {
        System.out.println("in Computer config");
    }
}

// Subclass
class AdvComputer extends Computer {
    @Override
    public void config() {
        System.out.println("in AdvComputer config");
    }
}

public class FirstCode4 {
    public static void main(String[] args) {
        // Create a reference of type Computer but instantiate it with AdvComputer
        Computer obj = new AdvComputer();
        // Calls the overridden method in AdvComputer
        obj.config(); // Output: in AdvComputer config
    }
}
```

### Explanation:

1. **Superclass (`Computer`)**: Defines the `config()` method, which prints `"in Computer config"`.
2. **Subclass (`AdvComputer`)**: Inherits from `Computer` and overrides the `config()` method to print `"in AdvComputer config"`.
3. **Reference Variable (`Computer obj`)**: Holds a reference of type `Computer` but points to an `AdvComputer` object.
4. **Method Call (`obj.config()`)**: Calls the overridden method from `AdvComputer`, demonstrating runtime polymorphism.

### Output:

```
in AdvComputer config
```

### Explanation of Output:

> The `config()` method of `AdvComputer` is invoked, as this is the method that was overridden, even though the reference is of type `Computer`.

### Note:

> Here, `AdvComputer` class is primarily used to override the `config` method from `Computer`. We can also use an inner class in this scenario.

---

## 2.2.3 Using Anonymous Inner Classes: A Practical Example

In real-world scenarios, there are cases where you need a specific implementation or behavior temporarily, and defining a new named class might be overkill. Anonymous inner classes are perfect for such scenarios. They allow you to provide a one-off implementation of a class or interface right at the point of instantiation.

Let's explore this concept with a practical example involving a car with special features:

---

### Practical Example: Customizing a Car

Imagine you are working with a car manufacturing company that produces limited edition cars. A customer requests a special feature, like doors that open upward, and you need to provide a custom implementation. Instead of creating a new class for this customization, you can use an anonymous inner class to define this feature directly.

**Code Example:**

```
package innerClass;

class Student {
    public void st() {
        System.out.println("Student");
    }
}

public class Ex1 {
    public static void main(String[] args) {
        // Creating an instance of Student using an anonymous inner class
        Student s = new Student() {
            @Override
            public void st() {
                System.out.println("Bright Student");
            }
        };
        s.st(); // Output: Bright Student
    }
}
```

**Explanation:**

1. **Class** `Student`: Defines a method `st()` that prints `"Student"`.
2. **Anonymous Inner Class**: Instantiated inline with `new Student() { ... }`. It overrides the `st()` method to provide a new behavior, printing `"Bright Student"`.
3. **Instantiation (**`Student s = new Student() { ... }`): Creates an object with a customized implementation without needing to define a new named class.

**Output:**

```
Bright Student
```

**Explanation of Output:**

> The overridden `st()` method in the anonymous inner class is called, which outputs `"Bright Student"`.

---

## Why Use Anonymous Inner Classes?

1. **Conciseness**: Allows you to provide a specific implementation without creating a separate named class. This is useful when you need a short-lived, specific behavior.
2. **Temporary Customization**: Ideal for situations where customization or special behavior is needed only in certain parts of your code, like creating special design features in a product.

**Scenario Application:**

In the car example, if the customer wants a special feature for a limited edition car, you could use an anonymous inner class to implement this feature directly where it is needed. This approach is efficient and keeps your code clean by avoiding unnecessary class definitions.

Anonymous inner classes provide a streamlined way to implement specific behaviors or functionalities when you don't want to create a full-fledged named class. They are useful for temporary or one-off customizations, like adding unique features to products in manufacturing or defining specific behavior for certain objects.**Anonymous Inner Class**

An **anonymous inner class** is used when you need to override methods or implement interfaces without explicitly declaring a class. It's an inline implementation that typically exists within a method.

## Why Use Inner Classes?

Inner classes in Java provide several advantages, particularly when it comes to code organization and encapsulation. Here's a detailed look at why inner classes are useful and how they fit into the design and maintenance of code.

### 1.Example Scenario:

Suppose you have a class named `Computer`. Within this `Computer` class, there are components like `HardDrive`, `RAM`, and `Chipsets`. These components are integral to the `Computer` class. In this case, it makes sense to include classes like `HardDrive` and `RAM` as inner classes within the `Computer` class because they are inherently part of it. The decision to use inner classes is made during the design phase, not while writing the code.

```
public class Computer {
    private String model;

    public Computer(String model) {
        this.model = model;
    }

    public void displayInfo() {
        System.out.println("Computer model: " + model);
    }

    // Inner class HardDrive
    class HardDrive {
        private int storage;

        public HardDrive(int storage) {
            this.storage = storage;
        }

        public void displayStorage() {
            System.out.println("Hard Drive storage: " + storage + " GB");
        }
    }

    // Inner class RAM
    class RAM {
        private int size;

        public RAM(int size) {
            this.size = size;
        }

        public void displaySize() {
            System.out.println("RAM size: " + size + " GB");
        }
    }
}
```

### Advantages:

- **Encapsulation**: Inner classes can access private members of their outer class, enabling tighter encapsulation.
- **Scope Control**: The scope of inner classes is limited to their enclosing class, preventing unnecessary exposure to other parts of the code.

## 2. Design Considerations

When designing a system, deciding which classes should be inner classes can improve the clarity and maintenance of your code. This design decision helps in keeping related classes together and ensuring that changes to one class are logically related to changes in the outer class.

**Example**: If you have a `Computer` class and you are designing it, it makes sense to include components like `HardDrive` and `RAM` as inner classes because they are tightly coupled with the `Computer` class. This encapsulation ensures that the components are only used within the context of the `Computer` and not exposed to other parts of the application.

## 3. Documentation in Programming

Documentation is crucial in programming for several reasons:

- **Maintaining Code**: Helps new developers understand the codebase and make necessary updates.
- **Client Communication**: Provides clear information about how the code works, which is essential when clients request updates or changes.
- **Long-Term Maintenance**: Documentation ensures that even if the original developers are not available, others can understand and maintain the code.

**Types of Documentation:**

- **Inline Comments**: Provide explanations directly within the code. Use comments to describe complex logic or decisions.
- **Javadoc**: A documentation tool in Java that generates HTML documentation from comments in the code. It helps create a formal API documentation that is accessible and easy to navigate.

**Creating Javadoc Documentation in Java:**

To create Javadoc documentation, use the `/** ... */` comment style before classes, methods, and fields. Here's an example:

```
/**
 * The Computer class represents a computer with a model and components.
 */
public class Computer {
    private String model;

    /**
     * Constructs a Computer with the specified model.
     *
     * @param model The model of the computer.
     */
    public Computer(String model) {
        this.model = model;
    }

    /**
     * Displays the information about the computer.
     */
    public void displayInfo() {
        System.out.println("Computer model: " + model);
    }

    /**
     * The HardDrive class represents a hard drive in the computer.
     */
    class HardDrive {
        private int storage;

        /**
         * Constructs a HardDrive with the specified storage capacity.
         *
         * @param storage The storage capacity of the hard drive in GB.
         */
        public HardDrive(int storage) {
            this.storage = storage;
        }

        /**
         * Displays the storage capacity of the hard drive.
         */
        public void displayStorage() {
            System.out.println("Hard Drive storage: " + storage + " GB");
        }
    }
}
```

## Generating Javadoc:

1. **Command-Line**: Run `javadoc` command from the terminal:

   ```
   javadoc -d doc -sourcepath src -subpackages your.package.name
   ```

   This command generates HTML documentation in the `doc` directory.

2. **IDE Integration**: Many IDEs (like IntelliJ IDEA or Eclipse) have built-in tools for generating Javadoc.

---

## Section 3:Interface in Java: A Key to Abstraction and Flexibility

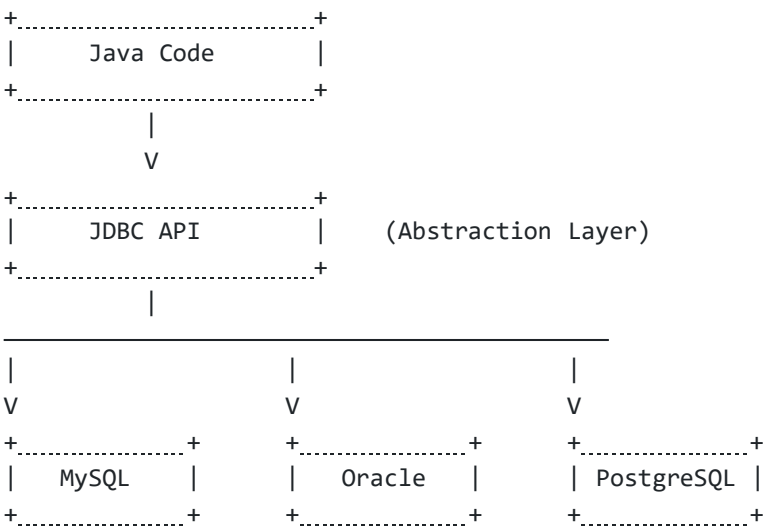
In software development, one of the key challenges is writing reusable and maintainable code. An **interface** in Java serves as a blueprint for classes, allowing multiple implementations without worrying about the specific details of how those implementations are carried out. This concept is particularly useful when interacting with systems that have multiple possible configurations or platforms, such as databases or servers.

As we continue our journey through **Java programming**, mastering these concepts will give you a strong foundation in both **modularity** and **abstraction**, essential for building robust software systems.

### Example: Database Interactions

Let’s consider an example involving databases. When we write Java code that interacts with databases, there are various vendors to choose from, such as **MySQL**, **Oracle**, and **PostgreSQL**. Although each database system has its own way of storing and managing data, as a Java developer, we don’t want to write separate code for each database.

This is where **interfaces** come in handy. Java has a **JDBC API (Java Database Connectivity API)** that provides a way to connect to different databases without changing the underlying code. The **JDBC API** acts as an abstraction layer between your Java code and the database. The beauty of this system is that you write your Java code once, and it can work with any database that has a JDBC-compliant driver.




In this scenario, the **JDBC API** serves as an interface that provides methods to execute queries, update data, and manage transactions. Regardless of which database is being used, the Java code communicates with the database through this interface, ensuring **Write Once, Run Anywhere (WORA)**, one of Java's core principles.

In Java, communication through interfaces ensures smooth operation, allowing different classes to interact seamlessly. The `.class` file is run by the JVM (part of the JDK software). For web applications, the `.class` file is executed by the JVM within specialized server software like Tomcat, GlassFish, or JBoss.

## The Role of GUI (Graphical User Interface)

However, customers cannot interact directly with Java code or backend logic. To make the application usable, we need an **interface** that allows customers to access the services provided by the application. In a banking app, this interface is typically a **Graphical User Interface (GUI)**. This GUI is essentially a bridge that stands between the customer and the application's core functionality.

The GUI could be a mobile app screen, a web page, or a desktop application. It presents the services — deposit, check balance, and withdraw — to the customer as buttons or menu options.

For instance, when you open a banking app, you may see the following screen:

## Visual Representation of the GUI:

```
sqlCopy code+-----------------------------------------+
|    Welcome to ABC Bank       |
+-----------------------------------------+
| 1. Deposit                   |
| 2. Check Balance             |
| 3. Withdraw                  |
+-----------------------------------------+
```

Here, the customer clicks on a button (e.g., "Deposit"), which sends a request to the application, and the corresponding method in the Java code is executed. The GUI acts as a **mediator**, an **interface** through which the interaction happens between the client (the customer) and the backend (the bank's services).

## Example: Web Applications and Servers

Consider another example where we are developing a **web application**. Web applications typically run on servers, and there are various server software options available such as **Tomcat**, **JBoss**, and **Wildfly**. Instead of writing server-specific code, we can write our Java code using the **Servlet API**, which is an interface for creating web applications. This allows our web app to run on any server without modification, thanks to the abstraction provided by the API.

## Real-World Scenario: Bank Application Interface

In a banking application, we might have multiple services such as **Deposit**, **Check Balance**, and **Withdraw**. These services interact with a user through a **GUI (Graphical User Interface)**, which serves as an interface between the user and the backend banking system. The user doesn't interact directly with the Java code; instead, they interact with the GUI, which sends the appropriate commands to the backend to handle these services. The GUI, in this sense, acts as an interface, facilitating communication between the customer and the application.

Here's a simple class structure for such a banking application:

```
abstract class Account {
    public abstract void deposit();
    public abstract void checkBalance();
    public abstract void withdraw();

    // Concrete method
    public void printPassbook() {
        // Business logic here
        System.out.println("Passbook updated");
    }
}
```

In this structure:

- The `Account` class is marked as **abstract** because we are defining some **abstract methods** (e.g., `deposit()`, `checkBalance()`, `withdraw()`) that will be implemented differently by each account type (Savings, Current, Salary).
- The method `printPassbook()` is a **concrete method** because it contains the same logic for all account types.

We could create subclasses for different types of accounts like **Savings**, **Current**, and **Salary**, and provide the specific implementation for each service:

```
class SavingsAccount extends Account {
    public void deposit() {
        // Implementation for Savings Account deposit
    }

    public void checkBalance() {
        // Implementation for checking balance
    }

    public void withdraw() {
        // Implementation for withdrawing from Savings Account
    }
}

class CurrentAccount extends Account {
    public void deposit() {
        // Implementation for Current Account deposit
    }

    public void checkBalance() {
        // Implementation for checking balance
    }

    public void withdraw() {
        // Implementation for withdrawing from Current Account
    }
}
```

However, there's still some level of **concreteness** in this setup because we are not achieving 100% abstraction. To achieve **full abstraction**, we can use an **interface**:

```
interface Account {
    void deposit();
    void checkBalance();
    void withdraw();
}
```

In an interface:

- All methods are **public** and **abstract** by default. We do not need to use the `abstract` keyword explicitly.
- Any class that implements this interface must provide implementations for all the methods.

Example implementation:

```
class SavingsAccount implements Account {
    public void deposit() {
        System.out.println("Depositing into Savings Account");
    }

    public void checkBalance() {
        System.out.println("Checking balance of Savings Account");
    }

    public void withdraw() {
        System.out.println("Withdrawing from Savings Account");
    }
}
```

Now, using this structure, we can achieve **100% abstraction**, as the `Account` interface provides no concrete methods, and the actual behavior is defined by the implementing classes. This approach is useful when we want to define a contract that multiple classes can adhere to, while ensuring that each class provides its own unique implementation.

## Naming Conventions and Implementation

When naming interfaces in Java, it's a common practice to use a capital `I` at the beginning of the interface name to differentiate it from regular classes. For example, an interface could be named `IAccount` or `IShape`.

Here's a quick example:

```
interface ISample {
    void m1();
    void m2();
}

class SampleImpl implements ISample {
    public void m1() {
        System.out.println("Implementation of m1");
    }

    public void m2() {
        System.out.println("Implementation of m2");
    }
}
```

In this case, the `SampleImpl` class **implements** the `ISample` interface, providing the required implementations for the methods `m1()` and `m2()`. The `@Override` annotation is used to inform the compiler that these methods are being overridden from the interface.

### Interface vs. Abstract Class: Key Differences

| Feature | Abstract Class | Interface |
|---|---|---|
| **Abstraction Level** | Provides partial abstraction | Provides full abstraction (before Java 8) |
| **Multiple Inheritance** | A class can extend only one abstract class | A class can implement multiple interfaces |
| **Concrete Methods** | Can have both abstract and concrete methods | Can only have abstract methods (before Java 8). Can have default and static methods from Java 8 onward |
| **Access Modifiers** | Can have any access modifier (public, protected, etc.) | Methods are always public and abstract unless default or static |
| **Usage** | Used when classes share common functionality | Used when a contract needs to be defined without any implementation |

## Section 7: Conclusion

In this comprehensive post, we have revised the core **Object-Oriented Programming** principles that serve as the foundation for understanding **inner classes** and **interfaces** in Java. We've explored the different types of inner classes, including **non-static**, **static**, **local**, and **anonymous** inner classes, and demonstrated how they work with code examples and memory mappings. Additionally, we've covered the importance of **interfaces**, their syntax, and how they can be implemented in Java. We also looked at combining inner classes with interfaces to build modular and maintainable code.

## Tables Recap:

| Concept | Explanation |
| --- | --- |
| **Non-Static Inner Class** | Requires an instance of the outer class. |
| **Static Inner Class** | Doesn’t require an instance of the outer class. |
| **Local Inner Class** | Defined within a method, scoped to that method. |
| **Anonymous Inner Class** | No class name, used for quick implementation or overriding. |
| **Interface** | A blueprint for a class, with abstract methods. |

## Further Reading and Resources

If you found this post insightful, you might be interested in exploring more detailed topics in my ongoing series:

1. **Full Stack Java Development**: A comprehensive guide to becoming a full stack Java developer.
2. **DSA in Java**: Dive into data structures and algorithms in Java with detailed explanations and examples.
3. **HashMap Implementation Explained**: Understand the underlying mechanics of Java’s HashMap.
4. **Inner Classes and Interfaces in Java**: Explore the intricacies of inner classes and interfaces.

### Connect with Me

Stay updated with my latest posts and projects by following me on social media:

- **LinkedIn**: Connect with me for professional updates and insights.
- **GitHub**: Explore my repositories and contributions to various projects.
- **LeetCode**: Check out my coding practice and challenges.

Your feedback and engagement are invaluable. Feel free to reach out with questions, comments, or suggestions. Happy coding!

---

**Rohit Gawande**
*Full Stack Java Developer | Blogger | Coding Enthusiast*