Abstraction in Java

The Art of Doing More with Less

Swipe

What is Abstraction ?

Abstraction in Java is all about **showing only the useful details** and **hiding the unnecessary complexity**. It allows you to work with features at a higher level without diving into the background logic.

In Java, Abstraction can be achieved by:

1. **Abstract classes** : can have **abstract methods** (without body) + **concrete methods** (with body).
2. **Interfaces** : defines **contracts** that a class must follow.

Why abstraction?

- Reduces **complexity**.
- Increases **flexibility & maintainability**.
- Focuses on ***"what to do"*** instead of ***"how to do"***.

Abstract Classes

Abstract classes are a powerful way to achieve abstraction in java.
Lets explore how we can use them in action.

1. Full Abstraction (only abstract methods)

Abstract class defines **only abstract methods** → forces child to implement everything.

```
abstract class Shape {
    abstract void draw(); // abstract
    abstract double area(); // methods
}

class Circle extends Shape {
    void draw() {
        System.out.println("Drawing Circle");
    }

    double area() {
        return Math.PI * 5 * 5;
    }
}
```

2. Partial Abstraction (concrete + abstract methods)

Some methods are **abstract (must implement)**, some are **concrete (already implemented)**.

```
abstract class Bank {
    abstract void loan(); // abstract method
    void deposit() { // concrete method
       System.out.println("Deposit Done");
    }
}

class SBI extends Bank {
    void loan() {
       System.out.println("SBI loan process");
    }
}
```

3. No Abstraction (only concrete methods)

Abstract class can also have **only concrete methods** (*weird but possible*).
It cannot be instantiated but works like a normal class.

```
abstract class Utility {
   void log() {
      System.out.println("Logging data...");
    }

   void backup() {
      System.out.println("Taking backup...");
   }
}

class FileUtility extends Utility {}
```

Partial Implementation Of Abstract Class

When a **subclass** extends an abstract class, it doesn't always need to implement **all** abstract methods.

If the subclass doesn't provide implementations for all abstract methods, then **that subclass must also be declared abstract**. Another subclass down the hierarchy can complete the implementation.

This is called **partial implementation**.

```
// Abstract class with 2 abstract methods
abstract class Vehicle {
    abstract void start();
    abstract void stop();

    // Concrete method
    void fuel() {
        System.out.println("Fueling up");
    }
}
```

```java
// Subclass gives partial implementation
abstract class Car extends Vehicle {
    @Override
    void start() {
        System.out.println("Car started");
    }
    // stop() is NOT implemented here
}

// Concrete subclass must finish
   implementation
class Sedan extends Car {
    @Override
    void stop() {
        System.out.println("Car stopped");
    }
}

public class Main {
    public static void main(String[] args) {
        Vehicle myCar = new Sedan();
        myCar.start();  // Car started
        myCar.fuel();   // Fueling up
        myCar.stop();   // Car stopped
    }
}
```

We've seen how we can keep methods inside *abstract classes*, but the question is —
can we also have other classes or interfaces inside them?

So yeah:

Inner classes → allowed (can be *normal*, *static*, or *anonymous*).

Interfaces inside abstract class → also allowed (*nested interface* concept).

→ The main reason to use them: **better encapsulation and grouping of related logic.**

1. Inner class – Normal class

```
abstract class Outer {
    abstract void display();

    // Inner class inside abstract class
    class Inner {
        void show() {
            System.out.println("Inner class
            inside abstract class");
        }
    }
}

class Child extends Outer {
    void display() {
        System.out.println("Child
         implementation of abstract method");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Child child = new Child();

        // Calls abstract method
           implementation
        child.display();
        // Prints: Inner class inside abstract
           class

        // Accessing Inner class
        Outer.Inner inner = child.new Inner();

        inner.show();
        // Prints: Inner class inside abstract
           class
    }
}
```

Key point:

- The **inner class Inner** is a non-static member class of *Outer*.
- To create its object, you first need an object of *Outer* (or *Child*, since it extends *Outer*).

2. Inner class – Static class

When you make an inner class **static**, you don't need an object of the outer class to create it.

```
abstract class Outer {
    abstract void display();

    // Static Inner class
    class StaticInner {
        void show() {
            System.out.println("Static Inner
            class inside abstract class");
        }
    }
}

class Child extends Outer {
    void display() {
        System.out.println("Child
         implementation of abstract method");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Child child = new Child();

        // Calls child implementation
        child.display();
        // Prints: Inner class inside abstract
           class

        // Accessing Static Inner class
        Outer.StaticInner inner =
        new Outer.StaticInner();

        inner.show();
        // Prints: Static Inner class inside
           abstract class
    }
}
```

Difference:

- **Non-static inner class** → Needs an instance of the outer **(so you use** *child.new Inner()***).**
- **Static inner class** → Can be created directly with *new Outer.StaticInner()*.

3. Inner Class – Anonymous Class

Anonymous inner classes are special because they let you **implement an abstract class (or *interface*) on the fly** without explicitly creating a separate subclass.

```
abstract class Outer {
    abstract void display();
}

public class Main {
    public static void main(String[] args) {
     // Anonymous inner class implementing
        Outer
     Outer obj = new Outer() {
     @Override
     void display() {
        System.out.println("Anonymous inner
        class implementation");
            }
      };
      obj.display(); // Calls anonymous
                        class method
    }
}
```

4. Inner Class – Abstract Class

An abstract class can be declared inside another abstract class (or even normal class). Child classes must then provide implementations for both outer and inner abstract classes if needed.

```
abstract class Outer {
    abstract void display();

    // Abstract inner class
    abstract class Inner {
        abstract void show();
    }
}

// Implementing both abstract classes
class Child extends Outer {
    @Override
    void display() {
        System.out.println("Child
        implementing Outer");
    }
```

```java
// Concrete subclass of Inner
class InnerChild extends Inner {
    @Override
    void show() {
        System.out.println("Inner abstract
        class implemented");
    }
}

public class Main {
    public static void main(String[] args) {
        Child child = new Child();
        child.display();
        // Prints: Child implementing Outer

        Child.InnerChild inner =
        child.new InnerChild();

        inner.show();
        // Prints: Inner abstract
            class implemented
    }
}
```

5. Inner Interface

Yes, abstract classes can also **contain interfaces** (*nested interfaces*). Nested interfaces are implicitly public static — you don't need an outer object to access them.

```
abstract class Outer {
    abstract void display();

    // Nested interface
    interface NestedInterface {
        void greet();
    }
}
```

```
// A class implements both Outer and its
nested interface
class Child extends Outer implements
    Outer.NestedInterface {
    @Override
    void display() {
        System.out.println("Child implementing
        abstract method");
    }
    @Override
    public void greet() {
        System.out.println("Hello from nested
        interface");
    }
}

public class Main {
    public static void main(String[] args) {
        Child child = new Child();
        child.display();
        // Prints: Child implementing abstract
           method
        child.greet();
        // Prints: Hello from nested interface
    }
}
```

Flaws of Abstract Classes in Java

1. No Multiple Inheritance of Classes

Java does **not allow multiple class inheritance** (to avoid *diamond problem*).
So, if you extend an abstract class, you **can't extend another class** — this restricts flexibility.

2. Tight Coupling

When you extend an abstract class, you are tightly coupled to its hierarchy.
Changing the abstract class may force changes in all subclasses.

3. Can't Achieve 100% Abstraction

Abstract classes can have concrete methods.
So unlike interfaces, they don't enforce **full abstraction** unless you only use abstract methods.

4. Inheritance-Only Model

Abstract classes **must be inherited** to be used.
You cannot directly create objects from them, unlike interfaces which can be **implemented by multiple classes** independently.

5. Not Suitable for Common Utilities

If you want to provide **common reusable behavior** (like *static utility methods*), *abstract classes* aren't the best choice.
Interfaces with *static* and *default* methods (since Java 8) are better for that.

6. Limited Design Flexibility

You can only **extend one abstract class**, but you can implement **multiple interfaces**.
This makes abstract classes less flexible in modern Java design.

Since abstract classes can’t provide complete abstraction, we turn to Interfaces for the solution. **Let’s explore how Interfaces make it possible**.

Interfaces

In Java, an **interface** is a reference type that defines a set of abstract methods and constants which a class must implement. Interfaces are used to achieve **abstraction** and **multiple inheritance of type**.

Initially, interfaces could only contain **abstract methods** and **constants**, but with later Java versions, more features were added to make them powerful and flexible.

Now, let’s explore all the aspects of interfaces with a complete code example. →

```java
interface MyInterface {
    // Variable (always public static final
       by default)
    int MAX = 100;

    // Abstract method (public and abstract
       by default)
    void abstractMethod();

    // Default method (Java 8+)
    default void defaultMethod() {
        System.out.println("Default Method
        (can be inherited/overridden)");
        helperMethod(); // calling private
                           method
    }

    // Static method (Java 8+)
    static void staticMethod() {
        System.out.println("Static Method
        (can’t be overridden, accessed via
        interface name)");
        privateStaticHelper(); // calling
        private static helper
    }
```

```
    // Private method (Java 9+)
    private void helperMethod() {
        System.out.println("Private Method
        (used inside default methods only)");
    }

    // Private static method (Java 9+)
    private static void privateStaticHelper()
    {
        System.out.println("Private Static
        Method (used inside static methods
        only)");
    }
}

// Implementing class
class MyClass implements MyInterface {
    @Override
    public void abstractMethod() {
        System.out.println("Abstract Method
        implementation");
    }
```

```
        // Overriding default method (optional)
        @Override
        public void defaultMethod() {
            System.out.println("Overridden
            Default Method");
        }
}

// Main class
public class InterfaceDemo {
    public static void main(String[] args) {
        MyClass obj = new MyClass();

        // Accessing abstract method
            (implemented in child)
        obj.abstractMethod();
        // Accessing default method
        obj.defaultMethod();
        // Accessing static method via
            Interface name
        MyInterface.staticMethod();
        // Accessing variable
        System.out.println("MAX Value = " +
        MyInterface.MAX);
    }
}
```

```
//Output
Abstract Method implementation
Overridden Default Method
Static Method (can’t be overridden, accessed
via interface name)
Private Static Method (used inside static
methods only)
MAX Value = 100
```

1. Variables in Interfaces

- All variables in an interface are **public**, **static**, **and final** by default.
- This means they are treated as constants and must be initialized at the time of declaration.
- They cannot be reassigned later.
- It defines constants that are shared by all implementations.

2. Abstract Methods

- Methods in an interface are **public and abstract** by default.
- They do not contain a body and must be implemented by the class that implements the interface.
- Abstract methods define the **contract** that the implementing class is obligated to follow.

3. Default Methods (Java 8 onwards)

- Introduced in **Java 8**.
- These methods have a body and can provide a **default implementation** within the interface.
- Implementing classes can use them directly or override them if required.
- This allows interfaces to **evolve** without breaking existing code, which ensures **backward compatibility** and **reduces code duplication**.

4. Static Methods (Java 8 onwards)

- Introduced in **Java 8**.
- A static method belongs to the interface itself, not to the objects of the implementing classes.
- They **cannot be overridden**.
- They are called using the interface name.
- It provides **utility methods** that are grouped within the interface.

5. Private Methods (Java 9 onwards)

- Introduced in **Java 9**.
- Private methods in an interface are not accessible outside the interface.
- They can be **instance private methods** or **static private methods**.
- Used only as helper methods inside **default** or **static** methods.
- Their main purpose is to **remove code duplication** within the interface.
- It promotes code reusability inside interface methods.

So, after exploring all the possible methods we can have inside interfaces, we will now explore what else can be placed inside them apart from methods.

Inside an interface, you can declare:

- **Nested Classes (static by default)**
- **Nested Interfaces**
- **Nested Enums**
- **Nested Records (Java 16+)**

Nested Classes in Interfaces

Any class you put inside an **interface** is **implicitly public static**.
That means you can use it **without creating an object of the interface**.
Let’s go through an example. →

```java
interface Outer {
    class Inner { // implicitly public static
        void show() {
            System.out.println("Hello from
            Inner Class inside Interface!");
        }
    }
}

public class Test {
    public static void main(String[] args) {
        // No need to create Outer object
        Outer.Inner obj = new Outer.Inner();
        obj.show();
    }
}
```

```
//Output
Hello from Inner Class inside Interface!
```

Why are they static by default?

- Interfaces don’t have constructors (they can’t be instantiated).
- So, if inner classes were **non-static**, you’d need an object of the outer type to create them (which isn’t possible).
- Hence, Java designers made them **automatically static**.

Where are Nested Classes Useful?

1. Utility Classes

- If your interface defines a contract, sometimes you want helper/utility logic tied to it.

Let’s take a look at an example. →

```java
interface MathOps {
    int operate(int a, int b);

    class Utils { // utility helper class
        static int square(int x) {
            return x * x;
        }
    }
}

class Add implements MathOps {
    public int operate(int a, int b) {
        return a + b;
    }
}

public class Demo {
    public static void main(String[] args) {
        Add add = new Add();
        System.out.println(add.operate(5,3));
                                    // 8
        System.out.println(MathOps.Utils.
        square(4));                 // 16
    }
}
```

2. Data Holder Classes

- Sometimes you define small *POJO* inside an interface.

Let's take a look at an example. →

```
interface UserRepo {
    class User {
      String name;
      User(String name) { this.name = name; }
    }
    User findByName(String name);
}
class InMemoryRepo implements UserRepo {
    public User findByName(String name) {
      return new User(name);
    }
}
public class Run {
    public static void main(String[] args) {
      UserRepo repo = new InMemoryRepo();
      UserRepo.User user =
       repo.findByName("Shekhar");
      System.out.println(user.name);
    }
}
```

Is it possible to have an abstract class inside an interface in Java?

Since **any class inside an interface is implicitly public static**, this also applies if that class is **abstract**.
So yes, you can declare a **nested abstract class inside an interface**.

Why is this allowed?

- Because the nested class is **always static** inside an interface, so it behaves like a **top-level class**, but just scoped inside the interface.
- Making it abstract is perfectly legal — subclasses outside the interface can extend it.

When is it Useful?

- When your interface defines a contract, and you want to provide **partial implementation or common logic** inside an abstract class.
- It acts like a **base class** tied closely to the interface.

```java
interface Shape {
    // Nested abstract class
    abstract class Drawer {
        abstract void draw();
        void info() {
            System.out.println("Drawer helper
            inside Shape interface");
        }
    }
}
class CircleDrawer extends Shape.Drawer {
    @Override
    void draw() {
        System.out.println("Drawing a
        Circle...");
    }
}

public class Test {
    public static void main(String[] args) {
        Shape.Drawer d = new CircleDrawer();
        d.info(); // method from abstract
                  class
        d.draw(); // overridden method
    }
}
```

Important Notes:

- A nested abstract class inside an interface **is not abstract itself by default** — you have to explicitly use abstract.
- It can have both **abstract and concrete methods**.
- And since it's static, you can extend it without needing any object of the interface.

Can we have nested private classes inside an interface?

- **No, we cannot.**

Why?

- Any class declared inside an interface is **implicitly public static**.
- Interfaces are designed to **expose contracts, not hide implementations**.
- You **cannot use private, protected, or final modifiers** for a nested class inside an interface.

Nested Interfaces Inside Interfaces

- It’s simply an **interface declared inside another interface**.
- In Java, a nested interface declared inside another interface is **implicitly public and static** (even if you don’t write those keywords).

Example:

```
interface Outer {
    interface Inner {
        void innerMethod();
    }
}
```

How to Implement a Nested Interfaces?

```
interface Outer {
    void outerMethod();

    interface Inner {
        void innerMethod();
    }
}

class Demo implements Outer.Inner {
    @Override
    public void innerMethod() {
        System.out.println("Inner interface
        method implemented");
    }
}

class Test {
    public static void main(String[] args) {
        Outer.Inner obj = new Demo();
        obj.innerMethod(); // Output: Inner
        interface method implemented
    }
}
```

Example in Real-World APIs:

Java's **Map interface** has a nested interface **Map.Entry<K,V>** to represent *key-value pairs*.

```
Map<String, Integer> map = new HashMap<>();

for (Map.Entry<String, Integer> entry :
     map.entrySet()) {
          System.out.println(entry.getKey() +
          " : " + entry.getValue());
}
```

Why Use Them?

- **Logical Grouping** – Sometimes an interface has a related sub-contract. **Example:** A Database interface might have a nested Connection interface.
- **Readability** – Keeps related contracts together instead of spreading them around.
- **Modularity** – Allows inner interfaces to be implemented independently.

Important Points:

- All nested interfaces inside an interface are **automatically public static**.
- You cannot mark them **private** or **protected**.
- To refer to them, you use the **Outer.Inner** syntax.
- They're often used when the outer interface defines a broader contract, and the nested interface defines a **particular behavior**.

The following are the examples of inner interfaces:

1. Marker Interface (inside an interface)

- A nested interface with **no methods**, only used as a *"tag"*.

Example:

```
interface SerializableManager {
    interface Serializable {}  // marker interface
}
```

2. Generic Interface (inside an interface)

- A nested interface that takes **type parameters**.

Example:

```
interface Container {
    interface Holder<T> {
        T get();
    }
}
```

3. Functional Interface (inside an interface, since Java 8)

- A nested interface with **just one abstract method**.
- Can be used with **lambda expressions**.

Example:

```
interface TaskManager {
    @FunctionalInterface
    interface Task {
        void execute();
    }
}

class Demo {
    public static void main(String[] args) {
        TaskManager.Task task = () ->
        System.out.println("Running
        Task...");
        task.execute();
    }
}
```

Nested Enums Inside Interfaces

- Just like nested classes or interfaces, an **enum can also be declared inside an interface**.
- By default, it is **public static** (*same rule as other nested types inside interfaces*).
- Enums inside interfaces are useful when you want to define **a set of constants closely tied to the contract** of that interface.

Example:

```
interface PaymentGateway {

    enum Status { // Transaction status
          SUCCESS, FAILED, PENDING, CANCELLED
    }

    enum ErrorCode { // Possible error codes
          INSUFFICIENT_FUNDS, NETWORK_ERROR,
          INVALID_ACCOUNT, TIMEOUT
    }

     Status processPayment(double amount);
     void handleError(ErrorCode code);
}
```

```java
// Implementation for UPI payments
class UpiPayment implements PaymentGateway {

    @Override
    public Status processPayment(double
     amount) {
        if (amount <= 0) {
            return Status.FAILED;
        } else if (amount > 5000) {
            return Status.PENDING;
            // requires manual approval
        }
        return Status.SUCCESS;
    }

    @Override
    public void handleError(ErrorCode code) {
        switch (code) {
            case INSUFFICIENT_FUNDS ->
                System.out.println("Balance
                is too low!");
            case NETWORK_ERROR ->
                System.out.println("Please
                check your internet.");
            case INVALID_ACCOUNT ->
                System.out.println("Invalid
                UPI ID.");
```

```
                case TIMEOUT -> System.out.println
                    (" Transaction timed out.");
            }
        }
    }

class TestPayment {
    public static void main(String[] args) {
        PaymentGateway pg = new UpiPayment();

        PaymentGateway.Status status =
        pg.processPayment(7000);

        if (status ==
            PaymentGateway.Status.SUCCESS) {
                System.out.println("Payment
                successful");
        } else if (status ==
            PaymentGateway.Status.PENDING) {
                System.out.println("Payment
                pending, wait for approval");
        } else {
            pg.handleError(PaymentGateway.
            ErrorCode.NETWORK_ERROR);
        }
    }
}
```

Nested Dictionary Inside Interfaces

- Just like nested *classes*, *enums*, and *interfaces*, you can also put **records inside an interface**.
- Any nested type (*class*, *interface*, *enum*, *record*) inside an interface is **implicitly public static**, so records inside interfaces are automatically **public static**.

Why Put a Record Inside an Interface?

- To define **data carriers (immutable data structures)** that are tightly related to the contract.
- Makes the interface **self-contained**: it defines both the **behavior (methods)** and the **data structure (records)**.
- Helps in **domain modeling** (like APIs that need *DTOs — Data Transfer Objects*).

Example:

```
interface OrderService {

    // Nested Record to represent order
       details
    record Order(int id, String product,
    double price) {}

    // Nested Record to represent order
       response
    record Response(Order order, Status
    status) {}

    // Nested Enum for status
    enum Status {
        SUCCESS, FAILED, PENDING
    }

    Response placeOrder(Order order);
}
```

```java
// Implementation of OrderService
class OnlineOrderService implements
OrderService {

    @Override
    public Response placeOrder(Order order) {
        if (order.price() <= 0) {
            return new Response(order,
                                Status.FAILED);
        }
        return new Response(order,
                            Status.SUCCESS);
    }
}
```

```
// Test
class TestOrder {
    public static void main(String[] args) {
        OrderService service = new
        OnlineOrderService();

        OrderService.Order order1 = new
        OrderService.Order(101, "Laptop",
        55000);

        OrderService.Response response =
        service.placeOrder(order1);

        System.out.println("Order: " +
        response.order());

        System.out.println("Status: " +
        response.status());
    }
}
```

Can We Put Annotations Inside an Interface?

- Yes

Why Would Anyone Do This?

- **Grouping:** Keep annotations that are only relevant to that interface.
- **Domain-Specific Contracts:** Some APIs define annotations as part of their contract.
- **Cleaner Namespacing:** Instead of scattering *annotations*, you keep them close to the interface they belong to.

Example:

```java
import java.lang.annotation.*;

interface Service {

    // Nested annotation
    @Retention(RetentionPolicy.RUNTIME)
    @Target(ElementType.METHOD)
    @interface Transactional {
        boolean readOnly() default false;
    }

    void process();

    @Transactional(readOnly = true)
    void fetchData();
}
```

```
class MyService implements Service {

    @Override
    public void process() {
        System.out.println("Processing...");
    }

    @Override
    public void fetchData() {
        System.out.println("Fetching data
        safely...");
    }
}
```

```
class TestAnnotation {
    public static void main(String[] args)
        throws Exception {

            for (var method :
                MyService.class.getMethods()) {
                if(method.isAnnotationPresent
                (Service.Transactional.class)) {

                    Service.Transactional ann =
                    method.getAnnotation
                    (Service.Transactional
                    .class);

                    System.out.println(
                    method.getName() +"readOnly="
                    + ann.readOnly());
                }
            }
        }
    }
```

Sealed Interfaces (Java 17)

Problem before Sealed Type:

Traditionally in Java, whenever you define an **interface**, *any class* in the codebase could implement it without restrictions.

For Example:

```
interface Shape {
    void draw();
}
```

Here, any class can do:

```
class RandomShape implements Shape {
    public void draw() {
        System.out.println("Random shape");
    }
}
```

This leads to **uncontrolled hierarchies**, where developers may implement an interface in unintended ways, making the code harder to maintain, extend, and reason about.

Solution: Sealed Interfaces

Java 17 introduced the **sealed** keyword, which allows you to **control and restrict** which classes or interfaces are permitted to implement/extend a given interface.

Syntax:

```
public sealed interface Shape permits Circle,
Rectangle, Triangle {
    void draw();
}
```

Here:

- Only *Circle*, *Rectangle*, and *Triangle* are allowed to implement the *Shape* interface.
- Any other class attempting to implement Shape will cause a **compile-time error**.

Rules for Implementing Classes

All permitted classes must explicitly declare **one of the following modifiers**:

1. **Final** → cannot be extended further

```
public final class Circle implements Shape {
    public void draw() {
        System.out.println("Drawing Circle");
    }
}
```

→ The hierarchy ends here.

2. **sealed** → can be extended, but only by a restricted set of subclasses

```
public sealed class Rectangle implements
Shape permits Square {
    public void draw() {
        System.out.println("Drawing
        Rectangle");
    }
}
```

→ Only Square can extend Rectangle.

3. **non-sealed** → removes restrictions and allows open extension

```
public non-sealed class Triangle implements
Shape {
    public void draw() {
        System.out.println("Drawing Triangle");
    }
}
```

→ The hierarchy ends here.

Benefits of Sealed Interfaces

Controlled inheritance → prevents unwanted or accidental implementations.

Stronger encapsulation → hierarchy is explicit and predictable.

Better maintainability → developers instantly know all valid subtypes.

Improved readability → sealed hierarchies act like a contract for your design.

Pattern Matching with Switch (using Sealed Interfaces) Java 21

Problem before:

When working with **sealed interfaces** and their subtypes, developers often relied on *if-else* chains or *switch* statements.

For Example:

```
Shape shape = new Circle(5);

if (shape instanceof Circle) {
    Circle c = (Circle) shape;
    // explicit cast required
    System.out.println("Area: " + (Math.PI *
    c.radius() * c.radius()));
} else if (shape instanceof Rectangle) {
    Rectangle r = (Rectangle) shape;
    // explicit cast required
    System.out.println("Area: " + (r.length()
    * r.width()));
}
```

Issue:

- Repeated **type checks** (*instanceof*)
- Manual **casting** after every check
- Boilerplate and error-prone code

Solution:

Java 21 introduces **pattern matching with switch**, which simplifies type-checking and casting.

- The compiler automatically handles casting.
- All **permitted subtypes** of a sealed interface must be covered (or explicitly handled with a default case).
- Ensures **type-safety** and **exhaustiveness checking**.

For Example:

```
sealed interface Shape permits Circle,
Rectangle {}

record Circle(double radius) implements Shape
{}

record Rectangle(double length, double width)
implements Shape {}
```

Using Pattern Matching in *Switch*:

```
static double area(Shape shape) {
    return switch (shape) {
        case Circle c ->
            Math.PI * c.radius() * c.radius();
        case Rectangle r ->
            r.length() * r.width();
    };
}
```

Key Advantages

- **No manual casting** → pattern variables (*c, r*) are automatically typed.
- **Exhaustive handling** → the compiler ensures every permitted subtype of Shape is covered.
- If a subtype is missing, compilation fails (unless a *default* is added).
- **Concise and readable** → reduces boilerplate and focuses on the logic.

Thanks For Reading

Let’s connect, share ideas, and keep leveling up our skills!