Understanding Java Class Types: Abstract, Concrete, Final, Sealed, and Record Classes

Advanced Java Programming

October 12, 2025

Contents

1 Introduction

Java provides multiple ways to define classes, each serving different purposes in object-oriented design. This report explores five fundamental class types: **Abstract**, **Concrete**, **Final**, **Sealed**, and **Record** classes. Understanding these concepts is crucial for designing robust, maintainable, and secure Java applications.

1.1 Overview of Class Types

Class Type	Instantiation	Inheritance
Abstract	Cannot instantiate	Can be extended
Concrete	Can instantiate	Can be extended (unless final)
Final	Can instantiate	Cannot be extended
Sealed (Java 17+)	Depends on abstract	Only permitted classes can extend
Record (Java 16+)	Can instantiate	Cannot be extended (implicitly final)

Table 1: Quick Comparison of Java Class Types

2 Abstract Classes

2.1 Theory

An **abstract class** is a class that cannot be instantiated directly and may contain abstract methods (methods without implementation). Abstract classes serve as templates for subclasses, defining a common interface while allowing subclasses to provide specific implementations.

2.2 Key Characteristics

- Declared with the `abstract` keyword
- Cannot be instantiated using `new` operator
- May contain both abstract and concrete methods
- Can have constructors, fields, and concrete methods
- Subclasses must implement all abstract methods (or be abstract themselves)

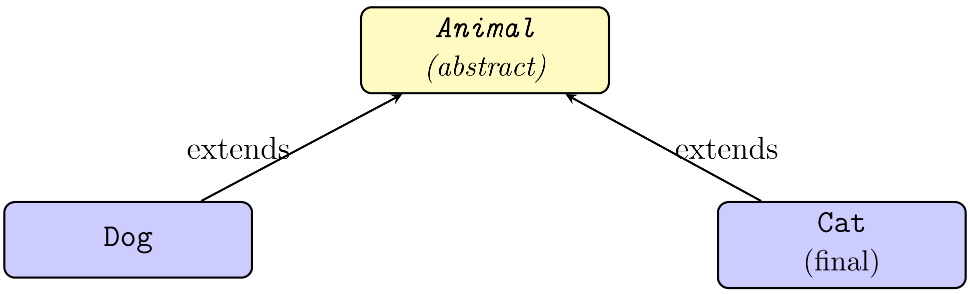
2.3 Code Example

```
abstract class Animal {
    protected String name;

    public Animal(String name) {
        this.name = name;
    }
```

```java

    // Abstract method - no implementation
    public abstract void makeSound();

    // Concrete method - has implementation
    public void sleep() {
        System.out.println(name + " is sleeping...");
    }
}

// Concrete subclass
class Dog extends Animal {
    public Dog(String name) {
        super(name);
    }

    @Override
    public void makeSound() {
        System.out.println(name + " barks: Woof! Woof!");
    }
}
```

Listing 1: Abstract Class Definition

2.4 Class Hierarchy Diagram



2.5 When to Use Abstract Classes

Use Abstract Class When:	**Example Scenarios**
You want to share code among related classes	Shape class with area() method
You need to declare non-static or non-final fields	Employee class with protected fields
You require access modifiers other than public	Template methods with protected helpers
You want to provide default behavior	Vehicle class with default move()

Table 2: Use Cases for Abstract Classes

3 Final Classes

3.1 Theory

A **final class** is a class that cannot be extended by any other class. This prevents inheritance and ensures that the class behavior cannot be modified through subclassing.

3.2 Key Characteristics

- Declared with the `final` keyword
- Can be instantiated normally
- Cannot be subclassed
- All methods are implicitly final (cannot be overridden)
- Provides security and immutability guarantees

3.3 Code Example

```java
final class Cat extends Animal {
    public Cat(String name) {
        super(name);
    }

    @Override
    public void makeSound() {
        System.out.println(name + " meows: Meow!");
    }
}

// This would cause a compile-time error:
// class MyCat extends Cat {} // ERROR: Cannot inherit from final Cat
```

Listing 2: Final Class Definition

3.4 Benefits and Trade-offs

Benefits	**Trade-offs**
Security - prevents malicious extensions	Reduced flexibility
Performance - JVM can optimize better	Cannot extend for testing
Immutability - guarantees behavior	Tight coupling
Clear design intent	Must use composition instead

Table 3: Final Classes: Benefits vs Trade-offs

3.5 Famous Final Classes in Java API

- `java.lang.String` - ensures string immutability
- `java.lang.Integer` - wrapper class integrity
- `java.lang.Math` - utility class protection
- `java.lang.System` - system-level security

4 Concrete Classes

4.1 Theory

A **concrete class** is a regular, fully-implemented class that can be instantiated. It contains no abstract methods and provides complete implementations for all its methods.

4.2 Key Characteristics

- Standard Java class without `abstract` keyword
- Can be instantiated using `new` operator
- Can be extended (unless marked `final`)
- All methods must have implementations
- Most commonly used class type

4.3 Code Example

```java
class Car {
    private String model;

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

    public void drive() {
        System.out.println("Driving the car: " + model);
    }
}

// Usage
Car car = new Car("Tesla");
car.drive();
```

Listing 3: Concrete Class Definition

4.4 Comparison Table

Feature	**Abstract**	**Concrete**	**Final**
Can instantiate?	No	Yes	Yes
Can extend?	Yes	Yes	No
Has abstract methods?	Optional	No	No
Keyword required?	Yes	No	Yes

Table 4: Class Type Feature Comparison

5 Sealed Classes (Java 17+)

5.1 Theory

A **sealed class** is a class that explicitly controls which other classes may extend it. Introduced in Java 17, sealed classes provide fine-grained control over the inheritance hierarchy.

5.2 Key Characteristics

- Declared with the `sealed` keyword
- Uses `permits` clause to specify allowed subclasses
- Each permitted subclass must be: `final`, `sealed`, or `non-sealed`
- Provides exhaustive pattern matching possibilities
- All permitted subclasses must be in the same module/package

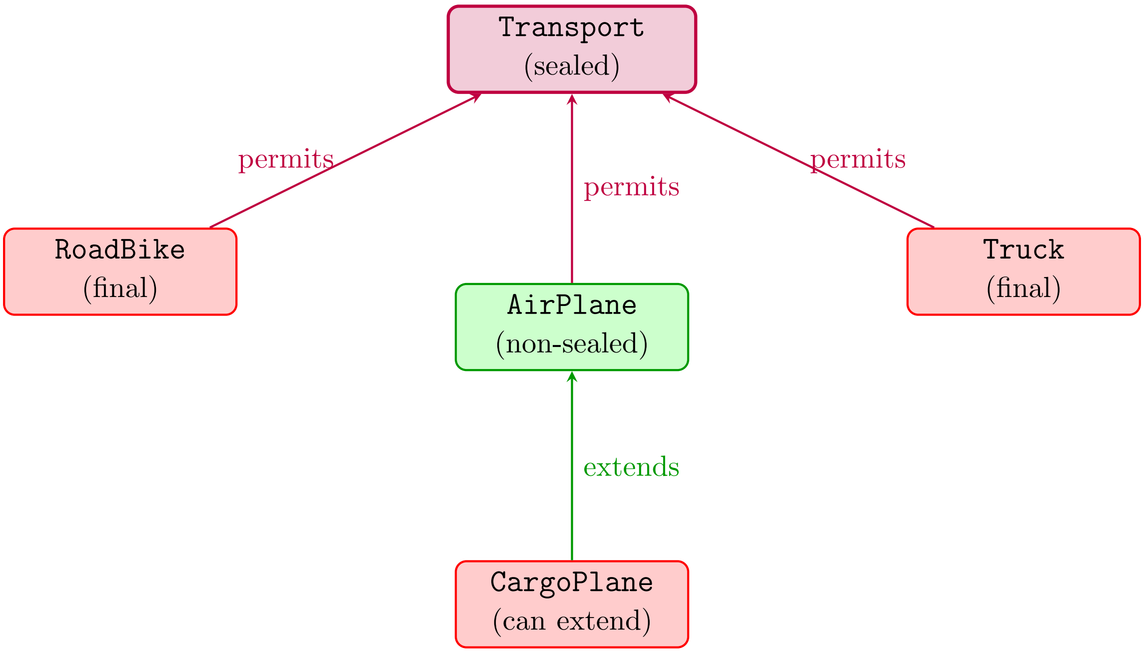
5.3 Code Example

```java
sealed abstract class Transport permits RoadBike, AirPlane, Truck {
    protected String model;

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

    public abstract void move();
}

// Permitted subclass declared as 'final'
final class RoadBike extends Transport {
    public RoadBike(String model) {
        super(model);
    }

    @Override
    public void move() {
        System.out.println("Road bike " + model + " pedals forward.");
    }
}

// Permitted subclass declared as 'non-sealed'
non-sealed class AirPlane extends Transport {
    public AirPlane(String model) {
        super(model);
    }

    @Override
    public void move() {
        System.out.println("Airplane " + model + " flies through the sky.");
    }
```

```java
}

// Permitted subclass declared as 'final'
final class Truck extends Transport {
    public Truck(String model) {
        super(model);
    }

    @Override
    public void move() {
        System.out.println("Truck " + model + " rolls on highways.");
    }
}
```

Listing 4: Sealed Class Definition

5.4 Sealed Class Hierarchy Diagram



5.5 Subclass Modifiers

Modifier	**Description**
`final`	The subclass cannot be extended further. Ends the inheritance chain.
`sealed`	The subclass is also sealed and must declare its own permitted subclasses.
`non-sealed`	Opens up the hierarchy - allows unrestricted inheritance from this point.

Table 5: Sealed Class Subclass Modifiers

5.6 Benefits of Sealed Classes

- **Domain Modeling**: Represent closed sets of types (e.g., payment methods, shapes)
- **Security**: Prevent unauthorized extensions
- **Exhaustiveness**: Enable complete pattern matching checks
- **API Design**: Control public API evolution
- **Maintainability**: Make inheritance hierarchy explicit and manageable

6 Record Classes (Java 16+)

6.1 Theory

A **record class** is a special kind of class introduced in Java 16 designed to be a transparent carrier for immutable data. Records provide a compact syntax for declaring classes whose primary purpose is to hold data, eliminating boilerplate code.

6.2 Key Characteristics

- Declared with the `record` keyword
- Implicitly `final` - cannot be extended
- All fields are implicitly `final` - immutable
- Automatically generates: constructor, getters, `equals()`, `hashCode()`, `toString()`
- Compact constructor syntax available
- Can implement interfaces but cannot extend classes
- Can have static fields and methods

6.3 Code Example

```java
// Simple record - replaces verbose POJO
record Point(int x, int y) {}

// Record with validation using compact constructor
record Person(String name, int age) {
    // Compact constructor - validates without explicit assignment
    public Person {
        if (age < 0) {
            throw new IllegalArgumentException("Age cannot be negative");
        }
        if (name == null || name.isBlank()) {
            throw new IllegalArgumentException("Name cannot be blank");
        }
    }
}

// Record with additional methods
record Rectangle(double width, double height) {
    // Custom method
    public double area() {
        return width * height;
    }

    // Static factory method
    public static Rectangle square(double side) {
        return new Rectangle(side, side);
    }
}
```

```java

// Record implementing interface
interface Drawable {
    void draw();
}

record Circle(double radius) implements Drawable {
    @Override
    public void draw() {
        System.out.println("Drawing circle with radius: " + radius);
    }
}
```

Listing 5: Record Class Definition

6.4 Code Comparison: Traditional Class vs Record

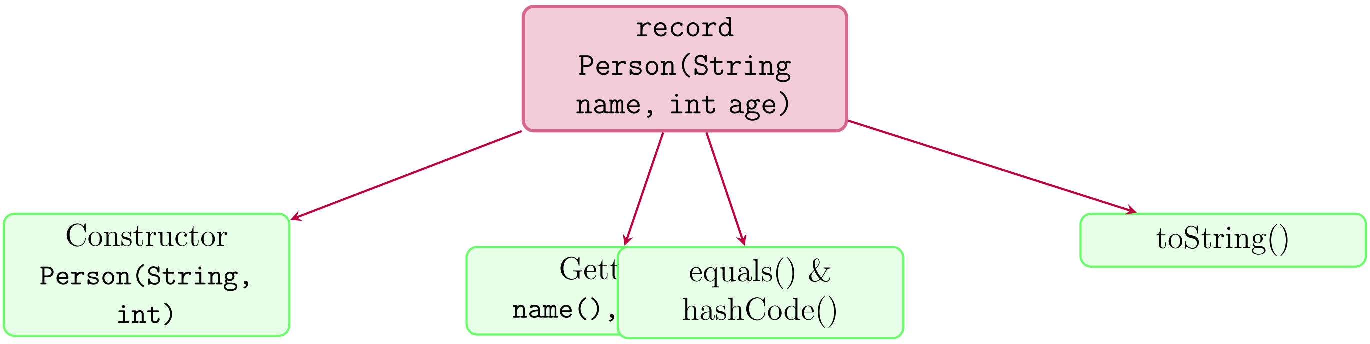
Traditional Class (50+ lines)

```
public final class Point {
    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; }
    public int getY() { return y; }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Point)) return false;
        Point point = (Point) o;
        return x == point.x && y == point.y;
    }

    @Override
    public int hashCode() {
        return Objects.hash(x, y);
    }

    @Override
    public String toString() {
        return "Point[x=" + x + ", y=" + y + "]";
    }
}
```

Record Class (1 line)

```
record Point(int x, int y) {}

// All methods automatically generated:
// - Constructor: Point(int x, int y)
// - Getters: x(), y()
// - equals(Object o)
// - hashCode()
// - toString()
```

6.5 Record Components Diagram



All components automatically generated by the compiler

6.6 When to Use Records

Use Records For:	Don't Use Records For:
Data Transfer Objects (DTOs)	Mutable state
Value objects	JavaBeans (need setters)
Query results from databases	Classes requiring inheritance
API responses/requests	Complex business logic
Configuration data	Classes with changing fields
Composite keys	JPA entities (limitations exist)

Table 6: Record Usage Guidelines

6.7 Advanced Record Features

```java
// Record with validation and derived data
record Temperature(double celsius) {
    // Compact constructor with validation
    public Temperature {
        if (celsius < -273.15) {
            throw new IllegalArgumentException(
                "Temperature below absolute zero");
        }
    }

    // Derived methods
    public double fahrenheit() {
        return celsius * 9/5 + 32;
    }

    public double kelvin() {
        return celsius + 273.15;
    }
}

```

```java
// Record with static fields
record MathConstants(double value, String name) {
    public static final MathConstants PI =
        new MathConstants(3.14159, "Pi");
    public static final MathConstants E =
        new MathConstants(2.71828, "Euler's number");
}

// Nested records
record Address(String street, String city, String zip) {}
record Employee(String name, Address address, int id) {}
```

Listing 6: Advanced Record Usage

6.8 Benefits of Records

- **Conciseness**: Reduces boilerplate code dramatically (50+ lines to 1 line)
- **Immutability**: All fields are final by default, promoting thread safety
- **Clarity**: Intent is clear - this is a data carrier
- **Correctness**: Compiler-generated methods are bug-free
- **Performance**: Can be optimized by JVM more effectively
- **Pattern Matching**: Works seamlessly with modern Java pattern matching

6.9 Record Restrictions

Restriction	**Reason**
Cannot extend other classes	Records implicitly extend `java.lang.Record`
Cannot be extended	Records are implicitly final
Cannot declare instance fields	All data must be in record header
Cannot have non-final instance fields	Ensures immutability
Native methods not allowed	Violates transparency principle

Table 7: Record Class Restrictions

7 Complete Demo Program

7.1 Program Output

When the demo program is executed, it produces the following output:

Console Output

```
=== Abstract / Concrete / Final classes demo ===
Buddy barks:  Woof!  Woof!
Buddy is sleeping...
Whiskers meows:  Meow!
Driving the car:  Tesla

=== Sealed classes demo (Java 17+) ===
Road bike Giant pedals forward on the road.
Airplane Boeing flies through the sky.
```

7.2 Main Method Analysis

```java
public class Demo {
    public static void main(String[] args) {
        // Abstract class usage - polymorphism
        Animal dog = new Dog("Buddy");
        dog.makeSound();
        dog.sleep();

        // Final class usage
        Cat cat = new Cat("Whiskers");
        cat.makeSound();

        // Concrete class usage
        Car car = new Car("Tesla");
        car.drive();

        // Sealed class usage - polymorphism with controlled hierarchy
        Transport bike = new RoadBike("Giant");
        Transport plane = new AirPlane("Boeing");
        bike.move();
        plane.move();

        // Record class usage
        Point p = new Point(10, 20);
        System.out.println("Point: " + p);
    }
}
```

Listing 7: Demo Main Method

8 Design Patterns and Best Practices

8.1 Choosing the Right Class Type

Scenario	Recommended Class Type
Shared behavior with variation	Abstract class
Complete, reusable component	Concrete class
Prevent inheritance for security	Final class
Controlled domain modeling	Sealed class
Immutable data carrier	Record class
Template method pattern	Abstract class
Utility class	Final class with private constructor
API with fixed subtypes	Sealed class
DTOs and value objects	Record class

Table 8: Class Type Selection Guide

8.2 Design Principles

1. **Favor composition over inheritance**: Use final classes and composition
2. **Design for extension or prohibit it**: Use abstract/sealed or final
3. **Program to interfaces**: Abstract classes define contracts
4. **Closed for modification, open for extension**: Sealed classes balance both
5. **Liskov Substitution Principle**: All class types must maintain this
6. **Immutability when possible**: Prefer records for data objects

9 Summary

9.1 Key Takeaways

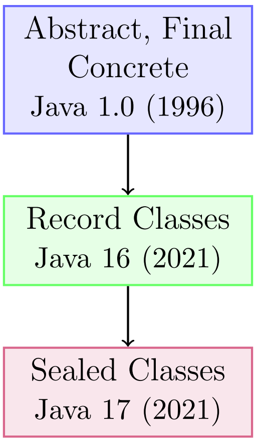
- **Abstract classes** provide partial implementation and enforce contracts
- **Final classes** prevent extension for security and design clarity
- **Concrete classes** are the standard building blocks of Java programs

Class Type	**Keyword**	**Can Create?**	**Can Extend?**	**Primary Use**
Abstract	`abstract`	No	Yes	Template/Contract
Concrete	None	Yes	Yes	Standard class
Final	`final`	Yes	No	Immutable design
Sealed	`sealed`	Depends	Restricted	Domain modeling
Record	`record`	Yes	No	Data carrier

Table 9: Complete Class Types Summary

- **Sealed classes** offer controlled inheritance for modern Java design
- **Record classes** eliminate boilerplate for immutable data objects
- Each class type serves specific design goals and architectural needs
- Choose based on your specific requirements for extensibility and security
- Modern Java (16+) encourages records for data-centric classes

9.2 Evolution Timeline



10 References

- Oracle Java Documentation: `https://docs.oracle.com/javase/tutorial/`
- JEP 395: Records (Final) - `https://openjdk.org/jeps/395`
- JEP 409: Sealed Classes (Final) - `https://openjdk.org/jeps/409`
- Effective Java by Joshua Bloch (3rd Edition)
- Java Language Specification - Chapter 8: Classes