# Inner Classes in Java

Java Programming

Department of Computer Science

December 3, 2025

# Agenda

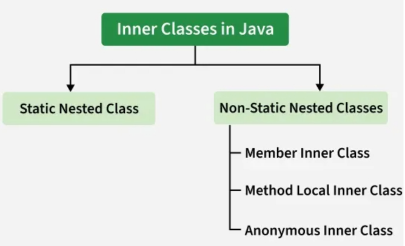
1. Introduction
2. Types of Inner Classes
    - Non-static Nested Class
    - Static Nested Class
    - Local Inner Class
    - Anonymous Inner Class
3. Access Modifiers
4. Real-world Examples
5. Best Practices
6. Common Pitfalls

# What are Inner Classes?

## Definition

Inner classes are classes defined within another class. They help in logical grouping and encapsulation.

- Also called **nested classes**
- Four types of inner classes
- Improve code **readability** and **maintainability**



# Four Types of Inner Classes

1. **Non-static Nested Classes (Inner Classes)**
2. **Static Nested Classes**
3. **Local Inner Classes** (Method-local)
4. **Anonymous Inner Classes**

## Key Difference

**Static vs Non-static**: Static nested classes don't have access to outer class instance variables.

# Non-static Nested Class

## Characteristics

- Has access to outer class members
- Can't exist without outer class instance
- Can be private, protected, or public

```
public class OuterClass {
    private String outerField = ”Outer”;

    class InnerClass {
        void display() {
            // Can access outer class private members
            System.out.println(”Accessing: ” + outerField);
        }
    }
}
```

# Instantiating Non-static Inner Class

```java
public class Main {
    public static void main(String[] args) {
        // First create outer class instance
        OuterClass outer = new OuterClass();
        // Then create inner class instance
        OuterClass.InnerClass inner =
            outer.new InnerClass();
        inner.display(); // Output: Accessing: Outer
    }
}
```

## Important

Inner object is always associated with an outer object instance.

# Static Nested Class

## Characteristics

- Declared with `static` keyword
- Can't access non-static members of outer class
- Can be instantiated without outer class instance

```
public class OuterClass {
    private static String staticField = "Static";
    private String instanceField = "Instance";
    static class StaticNestedClass {
        void display() {
            System.out.println(staticField); // OK
            // System.out.println(instanceField); // ERROR
        }
    }
}
```

# Instantiating Static Nested Class

```
public class Main {
    public static void main(String[] args) {
        // No outer instance needed
        OuterClass.StaticNestedClass nested =
            new OuterClass.StaticNestedClass();
        nested.display(); // Output: Static
        // Alternative syntax (also valid)
        OuterClass.StaticNestedClass nested2 =
            new OuterClass.StaticNestedClass();
    }
}
```

## Use Case

Useful for logical grouping of classes that don't need outer instance access.

# Local Inner Class

## Characteristics

- Defined inside a method or block
- Only accessible within that method/block
- Can access local variables (Java 8+: effectively final)

```
public class OuterClass {
    void outerMethod() {
        final String localVar = "Local";
        class LocalInnerClass {
            void display() {
                System.out.println("Local var: " + localVar);
            }
        }
        LocalInnerClass local = new LocalInnerClass();
        local.display();
    }
}
```

# Local Class with Parameters

```java
public class Calculator {
    void calculate(final int x, final int y) {
        class Adder {
            int add() {
                return x + y; // x and y must be final
            }
        }
        Adder adder = new Adder();
        System.out.println(”Sum: ” + adder.add());
    }
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        calc.calculate(10, 20); // Output: Sum: 30
    }
}
```

# Anonymous Inner Class

## Characteristics

- No name - defined and instantiated at same time
- Used for interface/abstract class implementation
- Common in event handling (Swing/AWT)

```
interface Greeting {
    void greet();
}
public class AnonymousExample {
    public static void main(String[] args) {
        Greeting g = new Greeting() {
            public void greet() {
                System.out.println(”Hello from Anonymous!”);
            }
        };
        g.greet(); // Output: Hello from Anonymous!
    }
}
```

# Anonymous Class with Abstract Class

```
abstract class Animal {
    abstract void makeSound();
}
public class TestAnimal {
    public static void main(String[] args) {
        Animal dog = new Animal() {
            void makeSound() {
                System.out.println(”Woof! Woof!”);
            }
        };
        dog.makeSound(); // Output: Woof! Woof!
        // Another anonymous class
        Animal cat = new Animal() {
            void makeSound() {
                System.out.println(”Meow!”);
            }
        };
        cat.makeSound(); // Output: Meow!
    }
}
```

# Access Modifiers for Inner Classes

| Modifier | Outer Class | Inner Class |
|---|---|---|
| public | | |
| protected | | |
| default (package) | | |
| private | | |

Table: Access Modifiers Comparison

## Note

Inner classes can be more restricted than outer classes. Outer class can only be public or default (package-private).

# Example 1: Iterator Pattern

```java
public class ShoppingCart {
    private String[] items = {"Apple", "Banana", "Orange"};

    // Inner class implementing iterator
    class CartIterator {
        private int index = 0;

        boolean hasNext() {
            return index < items.length;
        }

        String next() {
            return items[index++];
        }
    }

    public CartIterator getIterator() {
        return new CartIterator();
    }
}
```

# Using the Iterator

```java
public class Main {
    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart();
        ShoppingCart.CartIterator iterator =
            cart.getIterator();

        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
        // Output:
        // Apple
        // Banana
        // Orange
    }
}
```

## Advantage

Encapsulates iteration logic within the collection class.

# Example 2: Builder Pattern

```java
public class Computer {
    private String CPU;
    private String RAM;
    private String storage;

    // Private constructor
    private Computer(Builder builder) {
        this.CPU = builder.CPU;
        this.RAM = builder.RAM;
        this.storage = builder.storage;
    }

    // Static nested Builder class
    public static class Builder {
        private String CPU;
        private String RAM;
        private String storage;

        public Builder setCPU(String CPU) {
            this.CPU = CPU;
            return this;
        }
```

# Builder Pattern (Continued)

```java
        public Builder setRAM(String RAM) {
            this.RAM = RAM;
            return this;
        }

        public Builder setStorage(String storage) {
            this.storage = storage;
            return this;
        }

        public Computer build() {
            return new Computer(this);
        }
    }

    @Override
    public String toString() {
        return "Computer[CPU=" + CPU +
               ", RAM=" + RAM +
               ", Storage=" + storage + "]";
    }
}
```

# Using the Builder

```
public class Main {
    public static void main(String[] args) {
        Computer myPC = new Computer.Builder()
            .setCPU("Intel i7")
            .setRAM("16GB")
            .setStorage("1TB SSD")
            .build();

        System.out.println(myPC);
        // Output: Computer[CPU=Intel i7, RAM=16GB,
        // Storage=1TB SSD]
    }
}
```

## Benefits

- Fluent API for object creation
- Immutable objects
- Clear separation of construction and representation

# Best Practices

1. **Use static nested classes** when:
   - Don't need access to outer instance
   - Just logical grouping needed
2. **Use non-static inner classes** when:
   - Need access to outer instance members
   - Implementing helper/iterator classes
3. **Prefer lambda expressions** over anonymous classes for single-method interfaces (Java 8+)
4. **Limit anonymous class size** - keep them small and focused
5. **Avoid deep nesting** - more than 2 levels reduces readability

# Common Pitfalls and Solutions

## 1. Memory Leaks

- Inner class holds reference to outer class
- Prevents garbage collection
- **Solution**: Use static nested class when possible

## 2. Serialization Issues

- Inner classes have synthetic fields
- Can cause serialization problems
- **Solution**: Implement `Serializable` carefully

## 3. Testing Difficulties

- Private inner classes hard to test
- **Solution**: Use package-private access for testability

# Summary and Q&A

## Key Takeaways

- Four types: Non-static, Static, Local, Anonymous
- Choose based on access requirements
- Improve encapsulation and organization
- Enable callback mechanisms and event handling
- Useful for design patterns (Iterator, Builder)