Private Constructors in Java

Object-Oriented Programming

December 3, 2025

Agenda

1. Introduction to Private Constructors
2. Example 1: Singleton Pattern
3. Example 2: Utility Classes
4. Example 3: Factory Method Pattern
5. Example 4: Builder Pattern
6. Example 5: Constant/Enum-like Classes
7. Best Practices and Considerations
8. Summary

What is a Private Constructor?

Definition

A private constructor is a constructor that can only be accessed within the class itself, preventing instantiation from outside the class.

- **Access Modifier**: private
- **Visibility**: Only within the class
- **Purpose**: Control object creation
- **Common Uses**: Singleton pattern, utility classes, factory methods

Key Point

When all constructors are private, the class cannot be instantiated using the `new` keyword from outside.

Basic Syntax

Example (Private Constructor Declaration)

```
public class MyClass {
    // Private constructor
    private MyClass() {
    }
}
```

Example (Attempting External Instantiation - ERROR!)

```
public class Test {
    public static void main(String[] args) {
        MyClass obj = new MyClass(); // Error!
    }
}
```

Singleton Pattern - Eager Initialization

Purpose

Ensure a class has only one instance and provide global access to it.

```
public class DatabaseConnection {
    private static final DatabaseConnection instance =
        new DatabaseConnection();
    private DatabaseConnection() {
        System.out.println("Database connection created");
    }
    public static DatabaseConnection getInstance() {
        return instance;
    }
    public void connect() {
        System.out.println("Connected to database");
    }
}
```

Singleton Usage

Example (Using the Singleton)

```java
public class Main {
    public static void main(String[] args) {
        // Get the singleton instance
        DatabaseConnection db1 = DatabaseConnection.getInstance();
        DatabaseConnection db2 = DatabaseConnection.getInstance();
        db1.connect();
        // Check if both references point to same object
        System.out.println("Same instance? " + (db1 == db2));
        // Output: Same instance? true
    }
}
```

Important

The constructor is private, so you cannot use `new DatabaseConnection()`. You must use `getInstance()`.

Utility Classes I

Purpose

Create classes that contain only static methods and constants. No instance needed.

```java
public final class MathUtils {
    private MathUtils() {
        throw new AssertionError(
            "Utility class cannot " +
            "be instantiated");
    }
    public static double
    calculateCircleArea(double radius) {
        return Math.PI * radius * radius;
    }
```

```java
    public static int factorial(int n) {
        if (n <= 1) return 1;
        return n * factorial(n - 1);
    }
    public static boolean isPrime(int number
        ) {
        if (number <= 1) return false;
        for (int i = 2;i <= Math.sqrt(number
            );
            i++) {
            if (number % i == 0)
                return false;
        }
```

Using Utility Classes

Example (Correct Usage)

```
public class Calculator {
    public static void main(String[] args) {
        double area = MathUtils.calculateCircleArea(5.0);
        int fact = MathUtils.factorial(5);
        boolean prime = MathUtils.isPrime(17);
        System.out.println(”Area: ” + area);
        System.out.println(”Factorial: ” + fact);
        System.out.println(”Is Prime: ” + prime);
    }
}
```

Example (Incorrect Usage - ERROR!)

```
// This will cause compilation error
MathUtils utils = new MathUtils(); // Error!
```

Factory Method Pattern

Purpose

Control object creation through factory methods instead of constructors.

```java
public class Employee {
    private String name; private String type; private double salary;
    private Employee(String name, String type, double salary) {
        this.name = name; this.type = type;
        this.salary = salary;
    }
    public static Employee createManager(String name) {
        return new Employee(name, "Manager", 75000.0);
    }
    public static Employee createDeveloper(String name) {
        return new Employee(name, "Developer", 60000.0);
    }
    public static Employee createIntern(String name) {
        return new Employee(name, "Intern", 30000.0);
    }
    public String getDetails() {
        return name + " − " + type + " − $" + salary;
    }
```

Using Factory Methods

Example (Creating Employees)

```
public class Company {
    public static void main(String[] args) {
        // Using factory methods
        Employee manager = Employee.createManager(”Alice”);
        Employee developer = Employee.createDeveloper(”Bob”);
        Employee intern = Employee.createIntern(”Charlie”);
        System.out.println(manager.getDetails());
        System.out.println(developer.getDetails());
        System.out.println(intern.getDetails());
        // This would cause compilation error:
        // Employee emp = new Employee(”Name”, ”Type”, 50000); // Error!
    }
}
```

Output

```
Alice - Manager - $75000.0
Bob - Developer - $60000.0
```

Builder Pattern with Private Constructor

Purpose

Create complex objects step by step with a fluent interface.

```
public class Computer {
    private final String cpu;
    private final int ram;
    private final int storage;
    private final boolean hasGraphicsCard;
    // Private constructor
    private Computer(Builder builder) {
        this.cpu = builder.cpu;
        this.ram = builder.ram;
        this.storage = builder.storage;
        this.hasGraphicsCard = builder.hasGraphicsCard;
    }
    // Static Builder class
    public static class Builder {
        // Required parameters
        private final String cpu;
        private final int ram;

        // Optional parameters
```

Builder Pattern (Continued)

```java
            public Builder(String cpu, int ram) {
                this.cpu = cpu;
                this.ram = ram;
            }
            public Builder storage(int storage) {
                this.storage = storage;
                return this;
            }
            public Builder graphicsCard(boolean hasGraphicsCard) {
                this.hasGraphicsCard = hasGraphicsCard;
                return this;
            }
            public Computer build() {
                return new Computer(this);
            }
        }
        @Override
        public String toString() {
            return "Computer[CPU=" + cpu + ", RAM=" + ram +
                   "GB, Storage=" + storage +
                   "GB, Graphics=" + hasGraphicsCard + "]";
        }
    }
```

Using the Builder Pattern

Example (Building Computers)

```
public class ComputerShop {
    public static void main(String[] args) {
        // Using builder pattern
        Computer gamingPC = new Computer.Builder(”Intel i9”, 32)
            .storage(1000)
            .graphicsCard(true)
            .build();

        Computer officePC = new Computer.Builder(”Intel i5”, 16)
            .storage(512)
            .build();

        System.out.println(”Gaming PC: ” + gamingPC);
        System.out.println(”Office PC: ” + officePC);

        // This would cause compilation error:
        // Computer pc = new Computer(); // Error!
    }
}
```

Constant Classes with Private Constructor

Purpose

Create classes that only contain constants, similar to enums but more flexible.

```java
public final class ApplicationConstants {
    // Private constructor to prevent instantiation
    private ApplicationConstants() {
        // Throw exception if someone tries to instantiate via reflection
        throw new IllegalStateException("Constants class");
    }
    // Database constants
    public static final String DB_URL = "jdbc:mysql://localhost:3306/mydb";
    public static final String DB_USERNAME = "admin";
    public static final String DB_PASSWORD = "password";
    // Application settings
    public static final int MAX_USERS = 1000;
    public static final int SESSION_TIMEOUT = 1800; // seconds
    public static final String APP_VERSION = "2.1.0";
    // Error messages
    public static final String ERROR_INVALID_INPUT = "Invalid input provided";
    public static final String ERROR_DB_CONNECTION = "Database connection failed";
}
```

Using Constant Classes

Example (Accessing Constants)

```java
public class DatabaseManager {
    public void connectToDatabase() {
        try {
            String url = ApplicationConstants.DB_URL;
            String user = ApplicationConstants.DB_USERNAME;
            String password = ApplicationConstants.DB_PASSWORD;

            System.out.println("Connecting to: " + url);
            System.out.println("Max users allowed: " +
                ApplicationConstants.MAX_USERS);
            // Database connection logic here
        } catch (Exception e) {
            System.out.println(ApplicationConstants.ERROR_DB_CONNECTION);
        }
    }
    public void validateInput(String input) {
        if (input == null || input.trim().isEmpty()) {
            throw new IllegalArgumentException(
                ApplicationConstants.ERROR_INVALID_INPUT);
        }
```

Best Practices for Private Constructors

1. **Singleton Pattern**: Use for classes that need exactly one instance
2. **Utility Classes**: Mark class as `final` and throw exception in constructor
3. **Factory Methods**: Use when object creation logic is complex
4. **Builder Pattern**: Use for creating complex objects with many parameters
5. **Constant Classes**: Prevent instantiation of classes containing only constants

Important Considerations

- **Testing**: Private constructors can make testing more challenging
- **Reflection**: Can bypass private constructors (use SecurityManager)
- **Serialization**: Special care needed for singleton serialization
- **Inheritance**: Classes with only private constructors cannot be subclassed

When to Use Private Constructors

Appropriate Uses

- Singleton classes
- Utility classes
- Factory classes
- Builder classes
- Constant containers
- Enum-like classes

Avoid When

- Normal instantiation needed
- Class needs inheritance
- Simple value objects
- Framework components
- Test-heavy codebases

Use private constructors judiciously to enforce design constraints

Summary

- **Private constructors** restrict object creation to within the class
- **Five common patterns**:
  1. Singleton Pattern - Single instance
  2. Utility Classes - No instance needed
  3. Factory Methods - Controlled creation
  4. Builder Pattern - Complex object construction
  5. Constant Classes - Prevent instantiation
- **Benefits**: Better control, encapsulation, design enforcement
- **Considerations**: Testing difficulty, reflection issues

Key Takeaway

Private constructors are a powerful tool for enforcing specific object creation patterns and maintaining control over how instances of your classes are created and used.