Singleton, Multiton, Factory, & Strategy Design Patterns

Singleton Design Pattern

- **The Singleton Design Pattern**: Ensures that a class has only one instance and provides a global point of access to it.
    - This is useful in scenarios where you need exactly one instance of a class to coordinate actions across the system.
- **Motivation for the Singleton Pattern**
    - **Finite Resource Representation:** Like a camera object, where you may want to restrict its instantiation to avoid conflicts in using the hardware.
    - **Expensive to Create:** When creating instances of a class involves a lot of resources, and you want to reuse it.
    - **Coordination of Components:** For example, when you have a single instance coordinating communication between various components.
- **Implementation Steps for Singleton:**
    1. **Private Constructor:** This prevents any other class from instantiating it.
    2. **Static Method:** Create a static method to control the instantiation.
    3. **Lazy Initialization:** Only create an instance when requested.
- **Criticism of the Singleton Pattern:**
    - **Single Point of Failure:** If the Singleton class has issues, it can affect the entire application.
    - **Potential Global State:** A Singleton is like a global variable, making the system less flexible. For instance, if a phone has two front-facing cameras, the Singleton pattern could be restrictive.
- Example:

```
public class FrontCamera implements Camera {
        // Step 1: Declare a static instance variable
    private static FrontCamera singleton = null;

    private FrontCamera() { // Step 2: Private constructor
        // Initialization code
    }

        // Step 3: Static method for controlled instantiatio
        // Only instantiate when needed, reuse whenever possi
    public static FrontCamera getInstance() {
        if (singleton == null) {
            singleton = new FrontCamera();
        }
        return singleton;
    }
}

public class Main {
    public static void main(String[] args) {
        Camera c1 = FrontCamera.getInstance();
        Camera c2 = FrontCamera.getInstance();
        Camera c3 = FrontCamera.getInstance();
    }
}

// Explanation: This implementation prevents multiple
// instances from being created and returns the same
// instance every time getInstance() is called.
```

Multiton Design Pattern

- The Multiton pattern generalizes the Singleton by managing multiple unique instances, each identified by a key.
- **Implementation Steps for Multiton:**

1. **Private Constructor:** Prevent instantiation from outside.
2. **Static Collection:** Store instances in a `Map` with unique keys.
3. **Factory Method:** Return the same instance for a given key.

- Example:

```
public class Student {
        // Allows for multiple Student instances
    private static Map<Integer, Student> directory = new Hash
    private int pid;
    private String firstName;
    private String lastName;

    private Student(int pid, String first, String last) {
        this.pid = pid;
        this.firstName = first;
        this.lastName = last;
    }

        // Only create a new student with a new pid when need
        // Ensures no duplicate students (same pid) in direct
    public static Student getStudent(int pid, String first, S
        if (!directory.containsKey(pid)) {
            directory.put(pid, new Student(pid, first, last))
        }
        return directory.get(pid);
    }
}

// This pattern maintains a collection of instances,
// ensuring that for the same pid, you always receive
// the same Student object.
```

Factory Method Design Pattern

- **The Factory Method design pattern** provides an interface for creating an object but allows subclasses to alter the type of object that will be created.
    - This encapsulates the instantiation process.
- **Implementation Steps for Factory Method:**
    1. **Define a Parent Class:** Implement a static factory method that decides which subclass to instantiate.
    2. **Subclasses Override:** Subclasses inherit and implement their specific behavior.
- Example:

```
class TextNotification extends Notification { }
class EmailNotification extends Notification { }
class PushNotification extends Notification { }

public class Notification {
    public enum Type { TEXT, EMAIL, PUSH }

        // Factory Method
    public static Notification create(Type t) {
        switch (t) {
            case TEXT: return new TextNotification();
            case EMAIL: return new EmailNotification();
            case PUSH: return new PushNotification();
            default: throw new UnsupportedOperationException(
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Notification n = Notification.create(Notification.Typ
    }
}
```

```
// Explanation: Depending on the parameter passed,
// the Factory Method (create()) instantiates and
// returns the appropriate subclass.
```

Strategy Design Pattern

- **The Strategy pattern** allows defining a family of algorithms, encapsulating each one, and making them interchangeable.
    - This pattern lets the algorithm vary independently from clients using it.
- **Implementation Steps for Strategy Pattern:**
    1. **Define an Interface for Strategy:** The interface declares a method to be implemented by all strategies.
    2. **Implement Strategies:** Create classes that implement this interface.
    3. **Use Strategies Dynamically:** Instantiate the needed strategy based on the situation.
- Example:

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}

public class SmallestPerimeterFirst implements Comparator<Sha
    @Override
    public int compare(Shape o1, Shape o2) {
            // Java prodives built-in method for comparing
            // Double, String, and Integer values
        return Double.compare(o1.getPerimeter(), o2.getPerime
    }
}

public class LargestAreaFirst implements Comparator<Shape> {
    @Override
    public int compare(Shape o1, Shape o2) {
        return Double.compare(o2.getArea(), o1.getArea());
    }
}

public interface Shape {
    double getPerimeter();
    double getArea();
}

public class Main {
    public static void main(String[] args) {
        List<Shape> shapes = new ArrayList<>();
        // Add shapes to the list
        shapes.add(shape1);
                shapes.add(shape2);
                shapes.add(shape3);
        // Pass in a Comparator<> object to tell the
                // sort() method how to order the objects
        shapes.sort(new SmallestPerimeterFirst());
    }
}

// Explanation: The strategy pattern allows you to sort
// shapes differently without changing the core sorting
// logic in the client code.

// Different strategies (SmallestPerimeterFirst or
// LargestAreaFirst) can be used dynamically.
```