COMP 3550

7.2 — FACTORY, SINGLETON, ADAPTER PATTERNS

Week 7: Design Patterns & Architecture

FACTORY PATTERN

Let someone else handle the ‘new.’

Definition:

- Creates objects without exposing the instantiation logic to the client.
- Client only knows what it needs, not how it’s created.

FACTORY PATTERN

Let someone else handle the ‘new.’

Definition:

- Creates objects without exposing the instantiation logic to the client.
- Client only knows what it needs, not how it’s created.

Why Use It?

- Encapsulates creation logic in one place
- Makes it easier to add new types without changing client code
- Great for plugin-style design or replacing big switch/if chains

FACTORY PATTERN

Let someone else handle the ‘new.’

```
class NotificationFactory {
    public static Notification create(String type) {
        return switch (type.toLowerCase()) {
            case "email" -> new EmailNotification();
            case "sms"   -> new SmsNotification();
            default      -> throw new IllegalArgumentException("Unknown type");
        };
    }
}
```

```
Notification n = NotificationFactory.create("email");
n.send("Hello!");
```

FACTORY PATTERN

Let someone else handle the ‘new.’

```
class NotificationFactory {
    public static Notification create(String type) {
        return switch (type.toLowerCase()) {
            case "email" -> new EmailNotification();
            case "sms"   -> new SmsNotification();
            default      -> throw new IllegalArgumentException("Unknown type");
        };
    }
}
```

```
Notification n = NotificationFactory.create("email");
n.send("Hello!");
```

Yes this still leaves behind a *mild* OCP violation **but**

FACTORY PATTERN

Let someone else handle the ‘new.’

```
class NotificationFactory {
    public static Notification create(String type) {
        return switch (type.toLowerCase()) {
            case "email" -> new EmailNotification();
            case "sms"   -> new SmsNotification();
            default      -> throw new IllegalArgumentException("Unknown type");
        };
    }
}
```

```
Notification n = NotificationFactory.create("email");
n.send("Hello!");
```

Yes this still leaves behind a *mild* OCP violation **but** it is **centralized** and sets you up for OCP-friendly variations: like Abstract Factory, Factory Method with registration, or using reflection/service loaders to load implementations dynamically. *(Look up Factory with Registration for more interesting information on setting this up)*

SINGLETON PATTERN

Only one...ever!

Definition:

- Ensures a class has only one instance in the application.
- Provides a global access point to that instance.
- Example use: SettingsManager, AppConfig, LoggingService.

Caution:

- Can quickly turn into hidden global state
  - hard to test, tightly coupled code.
- Overuse = anti-pattern.
- Prefer dependency injection to control lifetime and scope when possible.

SINGLETON PATTERN

Only one...ever!

Definition:

- Ensures a class has only one instance in the application.
- Provides a global access point to that instance.
- Example use: SettingsManager, AppConfig, LoggingService.

Caution:

- Can quickly turn into hidden global state
  - hard to test, tightly coupled code.
- Overuse = anti-pattern.
- Prefer dependency injection to control lifetime and scope when possible.

SINGLETON PATTERN

Only one...ever!

```
public class SettingsManager {
    private static final SettingsManager INSTANCE = new SettingsManager();
    private SettingsManager() {} // private constructor

    public static SettingsManager getInstance() {
        return INSTANCE;
    }
}
```

SINGLETON PATTERN

*Aside: Isn't this the same (Hint: **NO**)*

```
public class SettingsManager {
    private static SettingsManager instance;
    private SettingsManager() {}

    public static SettingsManager getInstance() {
        if (instance == null) {
            instance = new SettingsManager();
        }
        return instance;
    }
}
```

SINGLETON PATTERN

*Aside: Isn't this the same (Hint: **NO**)*

```
public class SettingsManager {
    private static SettingsManager instance;
    private SettingsManager() {}

    public static SettingsManager getInstance() {
        if (instance == null) {
            instance = new SettingsManager();
        }
        return instance;
    }
}
```

Eager vs **Lazy** initialization

- Eager is thread safe without extra code
- Eager may involve a possible memory waste
- Lazy requires synchronization

You can do either in this class and be correct but understanding the differences is VERY important (for life)

ADAPTER PATTERN

Make incompatible things work together.

Definition:

- Wraps an incompatible interface so it works with existing code.
- Converts one interface into another expected by the client.

When to Use:

- Integrating legacy code with new systems.
- Using third-party APIs that don’t match your interface.

Analogy:

- Like a power plug adapter which lets your device fit into a different socket.

ADAPTER PATTERN

Make incompatible things work together.

```java
interface Printable {
    void print(String message);
}

// Legacy system
class Document {
    String text;
    Document(String text) { this.text = text; }
    String getText() { return text; }
}

class LegacyPrinter {
    void oldPrint(Document doc) {
        System.out.println("Legacy printing: " + doc.getText());
    }
}

// Adapter
class PrinterAdapter implements Printable {
    private LegacyPrinter legacy;

    PrinterAdapter(LegacyPrinter legacy) {
        this.legacy = legacy;
    }

    @Override
    public void print(String message) {
        Document doc = new Document(message); // convert format
        legacy.oldPrint(doc);
    }
}
```

Real World Java Examples

- java.util.Arrays.asList()
  - takes a regular String[] and adapts it to behave like a List
- InputStreamReader
  - takes InputStream (raw bytes) and adapts it into a Reader (reads characters)

PATTERN COMPARISON TABLE

Factory, Singleton and Adapter by flexibility, testability, complexity

Pattern	Flexibility	Testability	Complexity
Factory	**High** — easy to swap or add implementations	**High** — can inject or create mocks easily	**Low–Medium** – simple to implement, more complex with OCP-friendly registration
Singleton	**Low** — fixed single instance, hard to replace	**Low** — global state makes mocking difficult without hacks	**Low** — easy to implement, but risks long-term maintenance headaches
Adapter	**Medium** — adds compatibility without changing existing code	**Medium–High** — adapters can be mocked, but depend on legacy interface behavior	**Medium** — straightforward, but may require conversion logic

PAUSE & REFLECT

Here’s some messy code that could be refactored into any one of the three patterns we reviewed today. Review the code and refactor it using one of the patterns based on what you believe makes the most sense.

```
public class ReportExporter {
    public void export(String format, String content) {
        if (format.equalsIgnoreCase("pdf")) {
            PdfExporter pdf = new PdfExporter();
            pdf.initialize();
            pdf.write(content);
        } else if (format.equalsIgnoreCase("csv")) {
            CsvExporter csv = new CsvExporter();
            csv.open();
            csv.writeRow(content);
            csv.close();
        } else if (format.equalsIgnoreCase("html")) {
            HtmlExporter html = new HtmlExporter();
            html.render(content);
        } else {
            throw new IllegalArgumentException("Unsupported export format: " + format);
        }
    }
}
```