Unidade Curricular

“Padrões e Desenho de Software”

#04 – Creational Patterns (2)

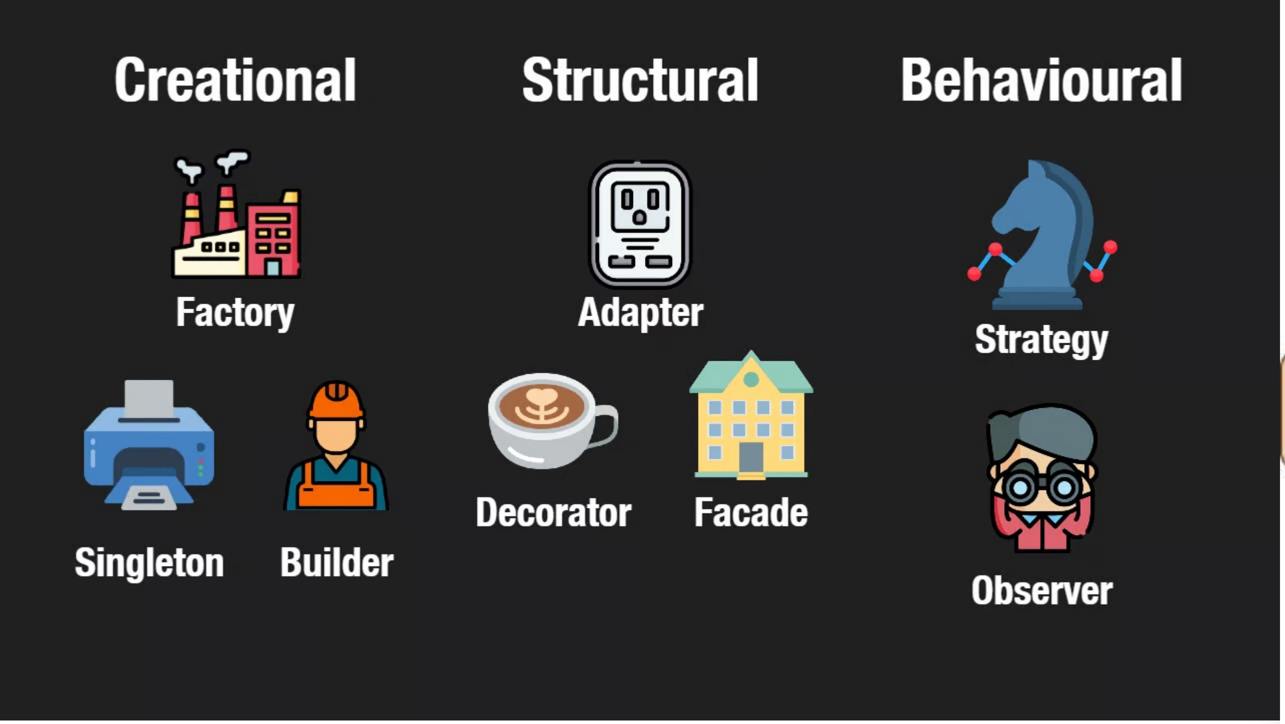
António José Ribeiro Neves

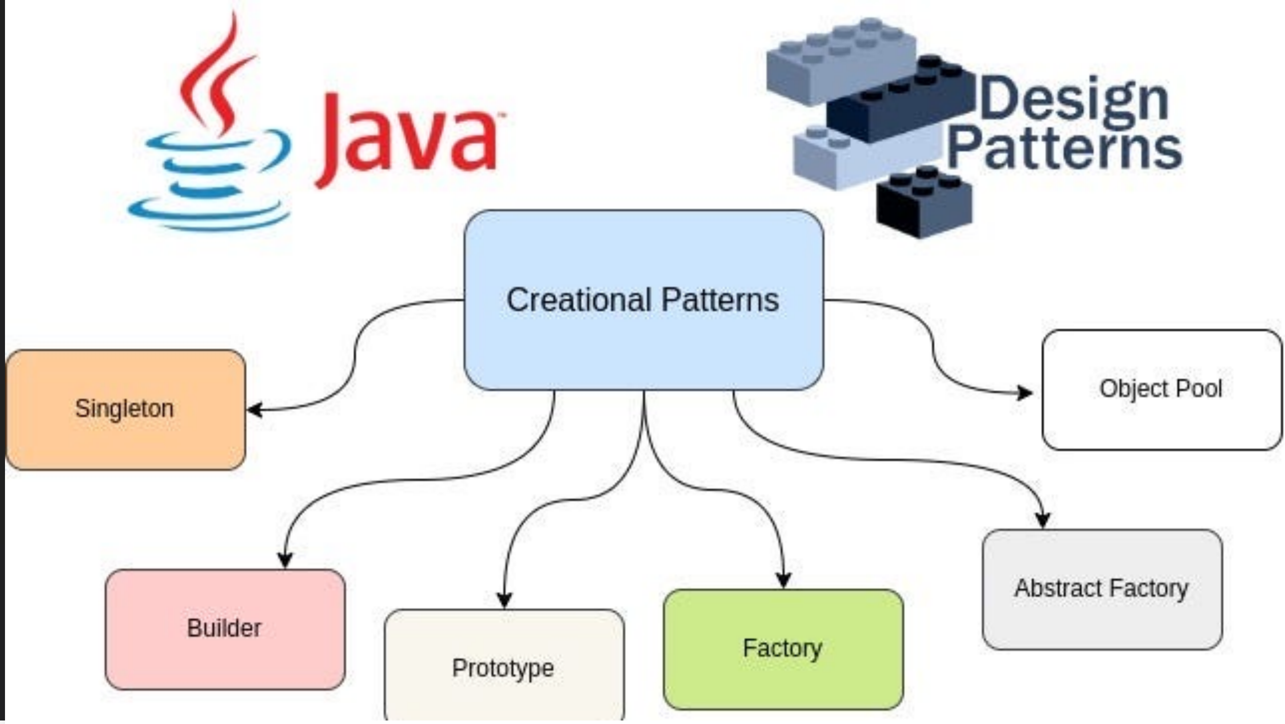
an@ua.pt

https://www.ua.pt/pt/uc/12275

Outline

- Quiz
- Singleton Pattern
- Builder Pattern
- Practical Challenge
- Prototype Pattern





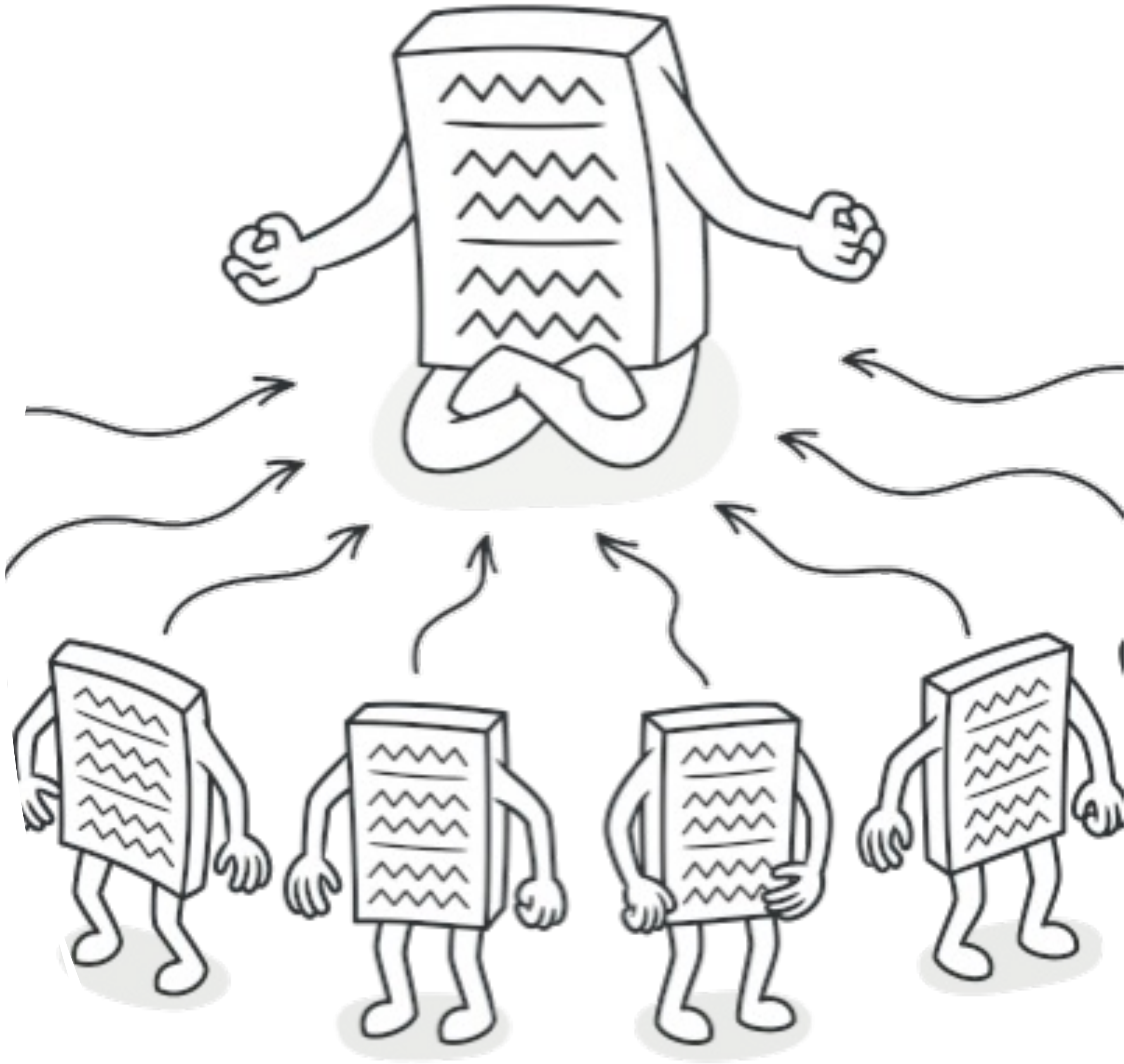
Creational Design Patterns

- **15 minutes** to explore the Singleton Design Pattern and answer the questions in the link:

`https://forms.gle/jtBJ7e1CiH84EvMY9`

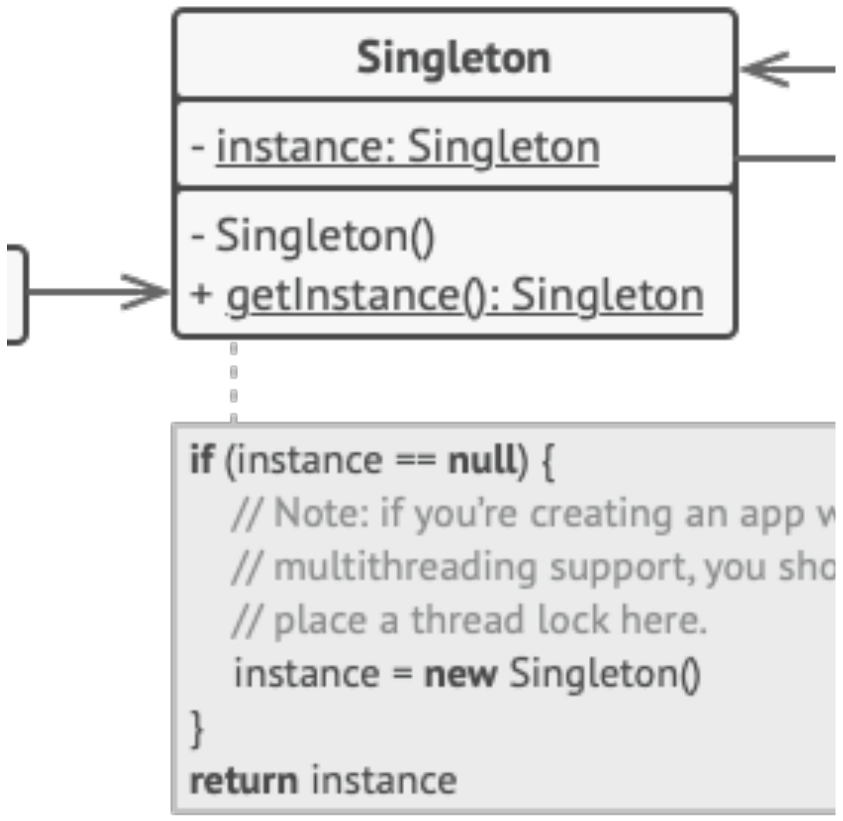
Singleton Pattern Overview

- Definition: The Singleton pattern is a creational design pattern that ensures a class has only one instance and provides a global point of access to that instance.
- Purpose: Ensures that a class has only one instance throughout the application lifecycle and provides a centralized access point to that instance.



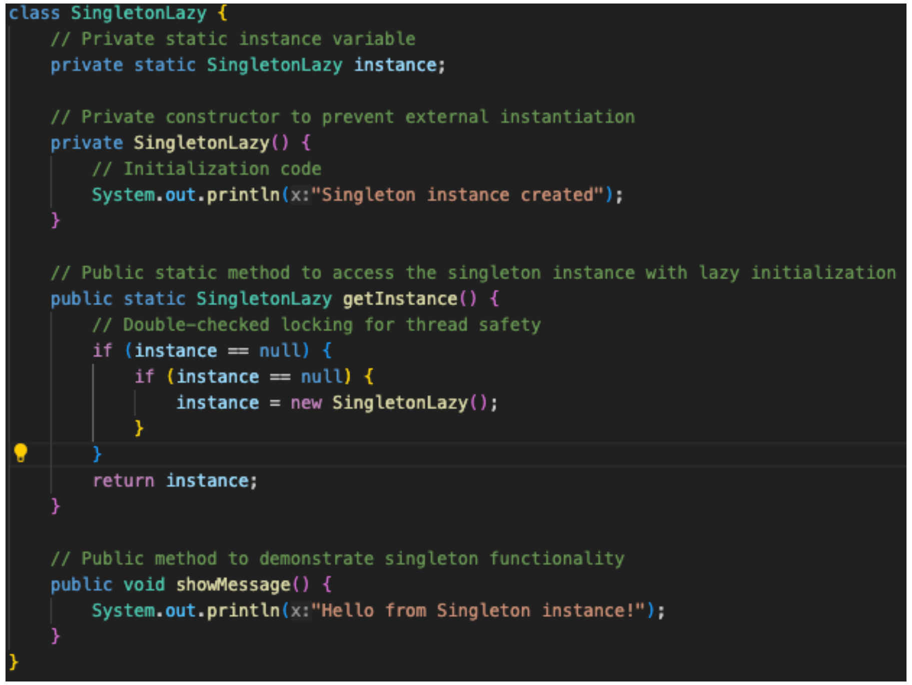
Characteristics of Singleton Pattern

- Single Instance: Ensures that only one instance of the class is created.
- Global Access: Provides a global access point to the single instance of the class.
- Lazy Initialization: Allows the instance to be created only when needed, improving memory usage.
- Thread Safety: Ensures that the singleton instance is thread-safe, preventing concurrent access issues.
- Private Constructor: Restricts external instantiation of the class, ensuring that the singleton instance is created internally.
- Static Method: Provides a static method to access the singleton instance, typically named getInstance().

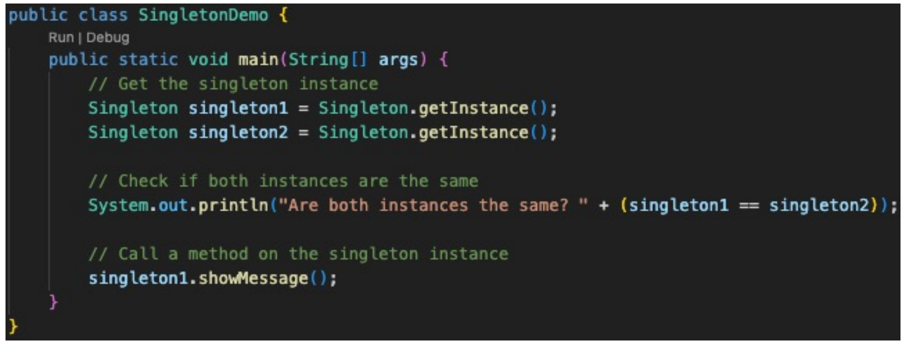


```
if (instance == null) {
  // Note: if you're creating an app w
  // multithreading support, you sho
  // place a thread lock here.
  instance = new Singleton()
}
return instance
```

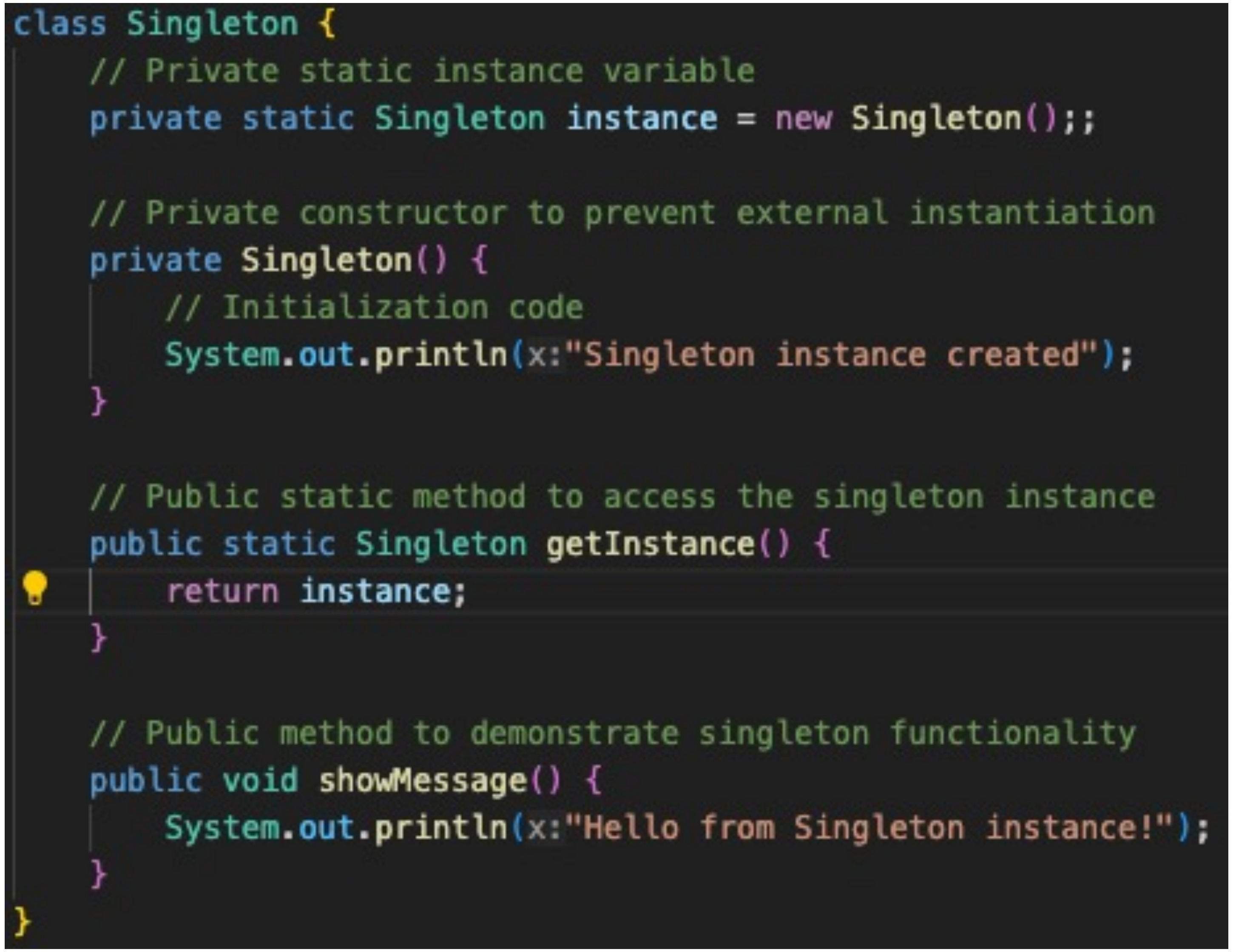
Example



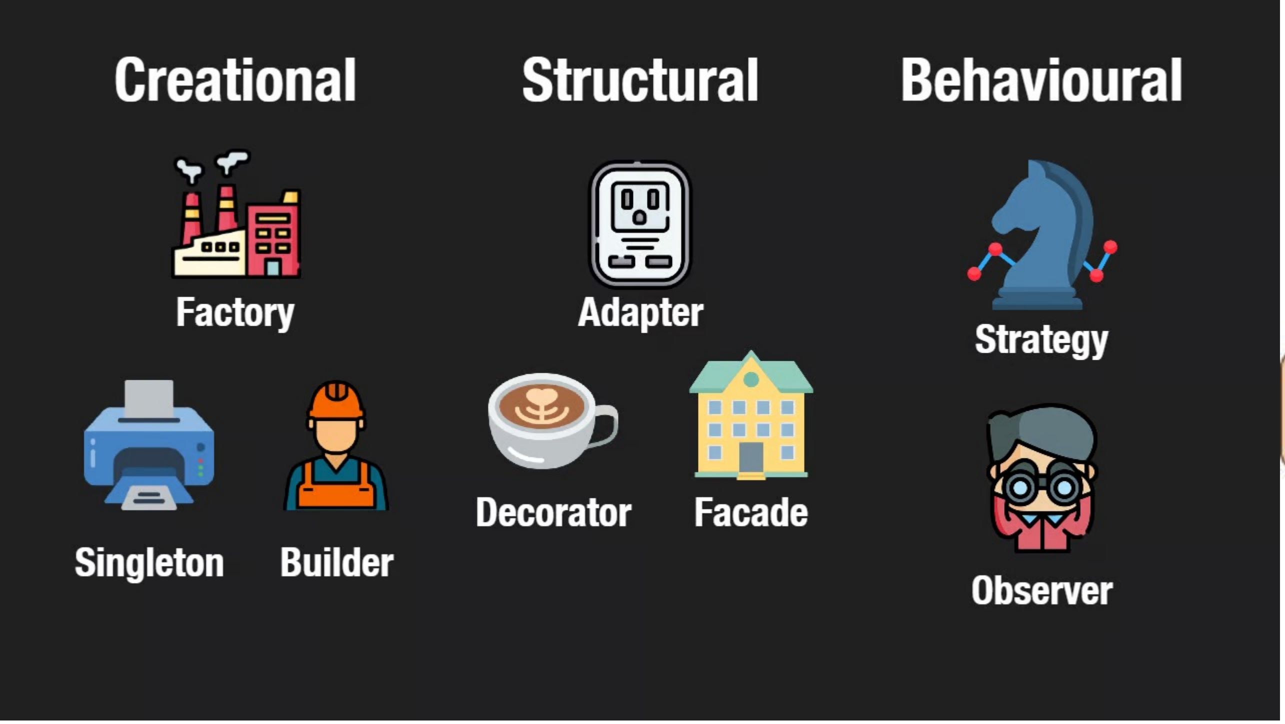
```java
class SingletonLazy {
    // Private static instance variable
    private static SingletonLazy instance;

    // Private constructor to prevent external instantiation
    private SingletonLazy() {
        // Initialization code
        System.out.println(x:"Singleton instance created");
    }

    // Public static method to access the singleton instance with lazy initialization
    public static SingletonLazy getInstance() {
        // Double-checked locking for thread safety
        if (instance == null) {
            if (instance == null) {
                instance = new SingletonLazy();
            }
        }
        return instance;
    }

    // Public method to demonstrate singleton functionality
    public void showMessage() {
        System.out.println(x:"Hello from Singleton instance!");
    }
}
```

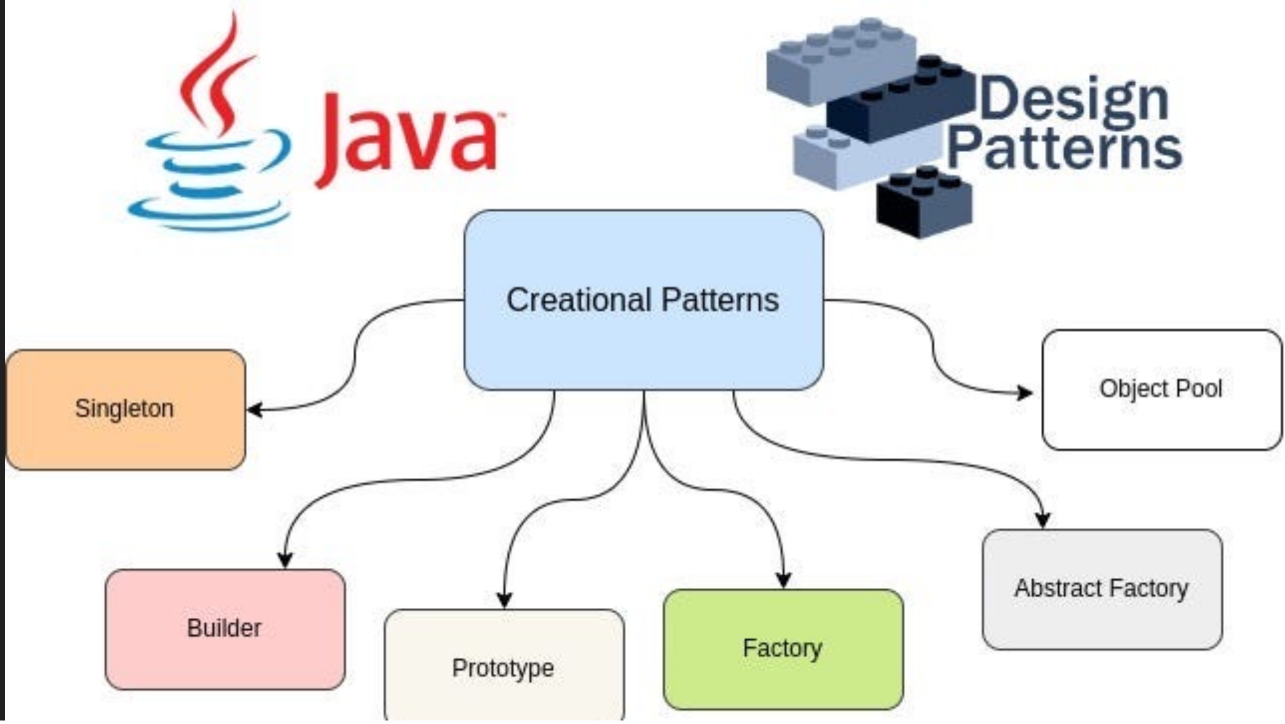


```java
public class SingletonDemo {
    Run | Debug
    public static void main(String[] args) {
        // Get the singleton instance
        Singleton singleton1 = Singleton.getInstance();
        Singleton singleton2 = Singleton.getInstance();

        // Check if both instances are the same
        System.out.println("Are both instances the same? " + (singleton1 == singleton2));

        // Call a method on the singleton instance
        singleton1.showMessage();
    }
}
```



```java
class Singleton {
    // Private static instance variable
    private static Singleton instance = new Singleton();;

    // Private constructor to prevent external instantiation
    private Singleton() {
        // Initialization code
        System.out.println(x:"Singleton instance created");
    }

    // Public static method to access the singleton instance
    public static Singleton getInstance() {
        return instance;
    }

    // Public method to demonstrate singleton functionality
    public void showMessage() {
        System.out.println(x:"Hello from Singleton instance!");
    }
}
```





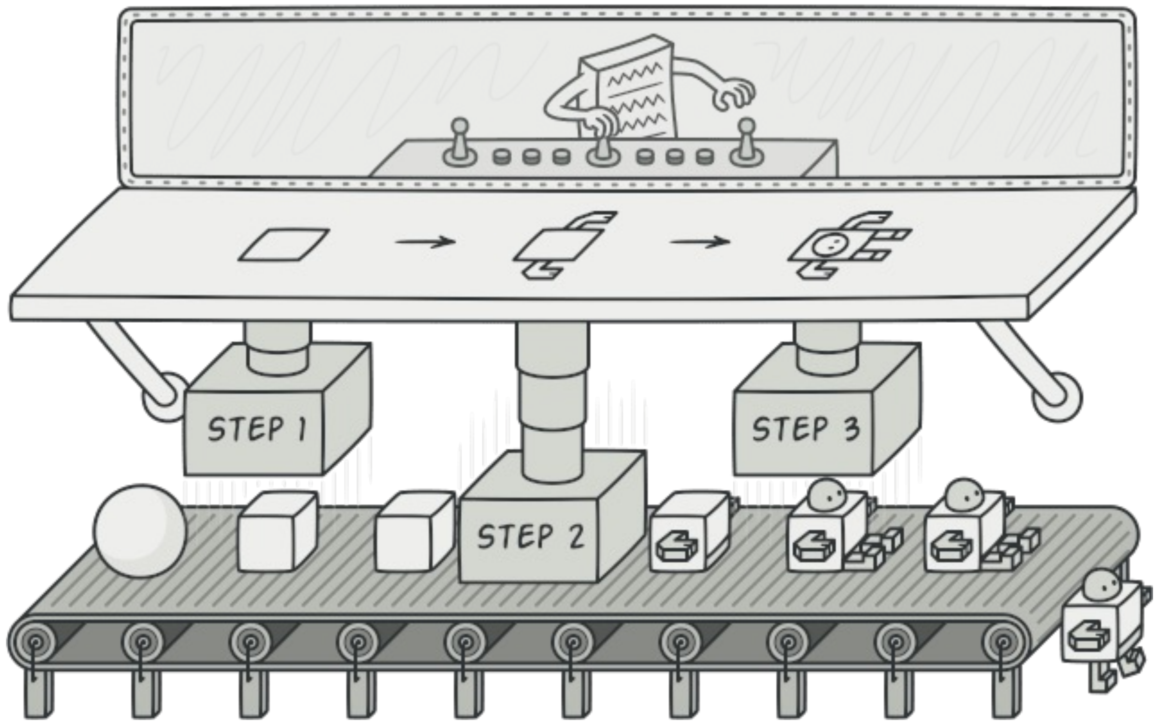
Creational Design Patterns

- **15 minutes** to explore the Builder Design Pattern and answer the questions in the link:

`https://forms.gle/71H4ZKUaH5rm8kbP9`

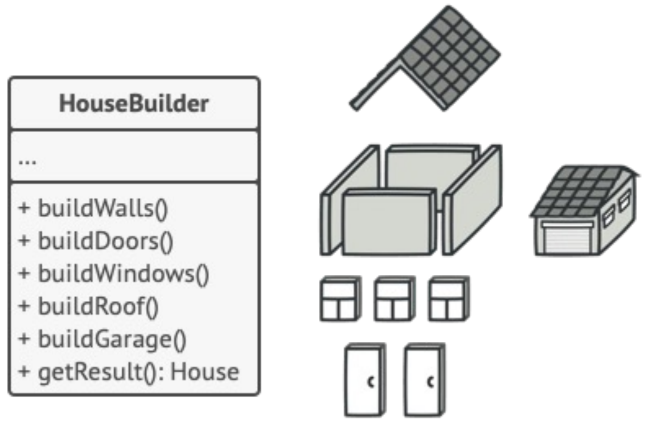
Introduction to Builder Pattern

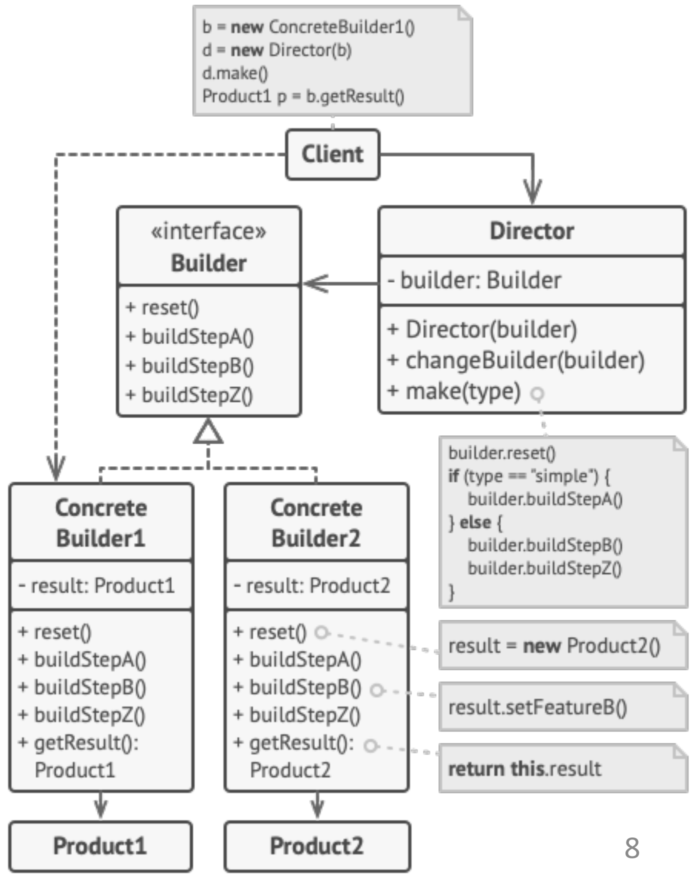
- Definition: The Builder pattern is a creational design pattern that separates the construction of a complex object from its representation, allowing the same construction process to create different representations.
- Purpose: Simplifies object creation by providing a flexible and fluent interface for constructing complex objects with varying configurations.



Key Components of Builder Pattern

- Controls the construction process by invoking builder methods to build the product.
- Abstract interface defining methods for building parts of the product.
- Concrete Builders: Implement the builder interface to construct and assemble parts of the product.
- Product: Represents the complex object being constructed.





```
b = new ConcreteBuilder1()
d = new Director(b)
d.make()
Product1 p = b.getResult()
```

```
builder.reset()
if (type == "simple") {
    builder.buildStepA()
} else {
    builder.buildStepB()
    builder.buildStepZ()
}
```

```
result = new Product2()
```

```
result.setFeatureB()
```

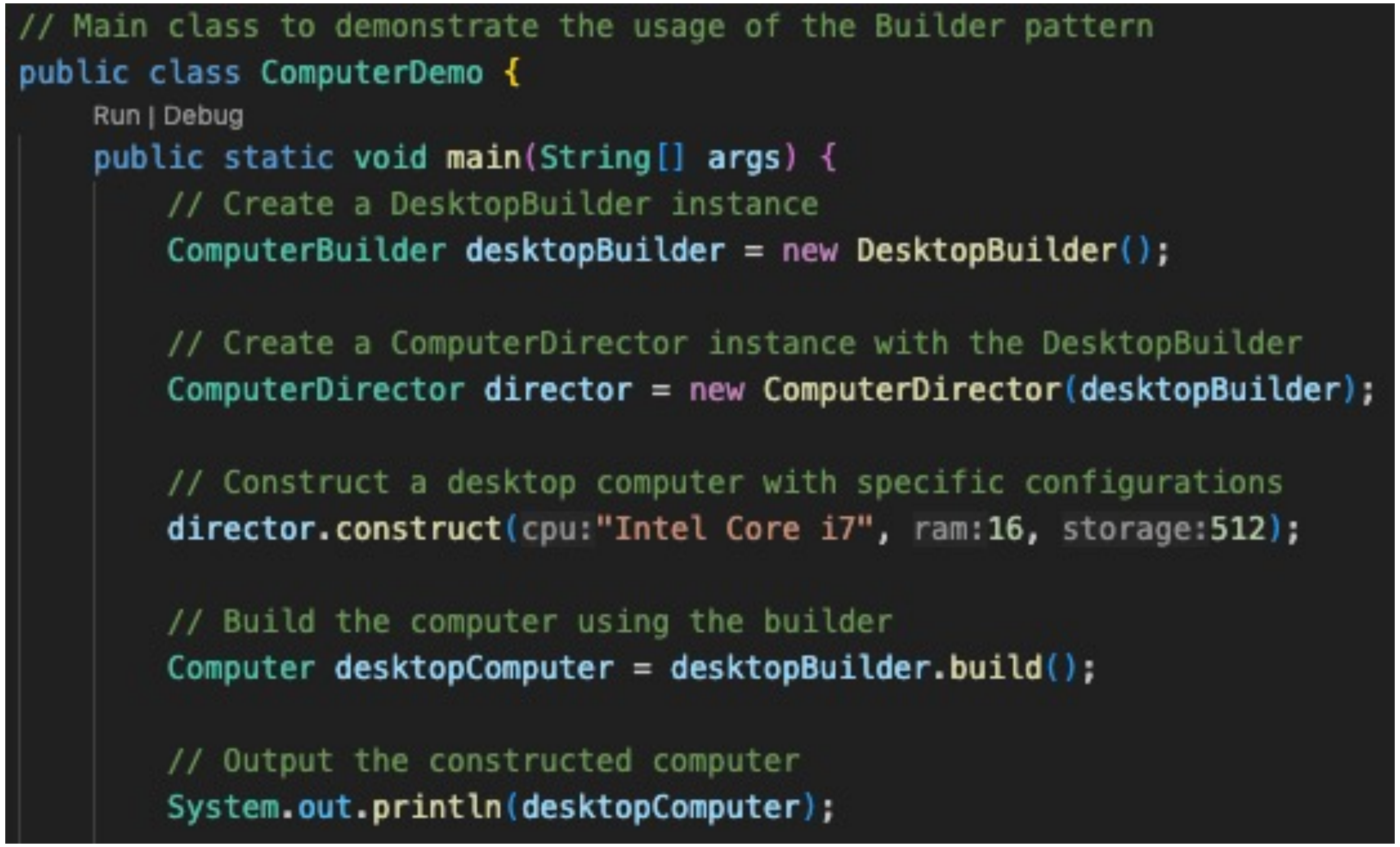
```
return this.result
```

Benefits of Builder Pattern

- Encapsulates construction logic: Separates the construction process from the final product, improving code maintainability and flexibility.
- Provides a fluent interface: Offers a clear and intuitive way to specify object configuration, enhancing code readability.
- Supports variation: Allows for the creation of multiple variations of the same complex object, accommodating diverse requirements.

Example

- Imagine building a computer with different configurations, such as CPU type, RAM size, and storage capacity.
- The Builder pattern allows us to construct the computer step by step, customizing its configuration based on user preferences.
- We can use different concrete builders to create different types of computers, such as gaming PCs, workstations, or servers.



```java
// Main class to demonstrate the usage of the Builder pattern
public class ComputerDemo {
    Run | Debug
    public static void main(String[] args) {
        // Create a DesktopBuilder instance
        ComputerBuilder desktopBuilder = new DesktopBuilder();

        // Create a ComputerDirector instance with the DesktopBuilder
        ComputerDirector director = new ComputerDirector(desktopBuilder);

        // Construct a desktop computer with specific configurations
        director.construct(cpu:"Intel Core i7", ram:16, storage:512);

        // Build the computer using the builder
        Computer desktopComputer = desktopBuilder.build();

        // Output the constructed computer
        System.out.println(desktopComputer);
```

Let’s take a short break

10 Minutes

You are free to go grab a coffee, water, etc.

But... 10 minutes **is 10 minutes** (600 seconds, **not 601 seconds!)**

10 minutes

Challenge: Implementing a Builder Class for Pizza Orders

Create a PizzaBuilder class that constructs Pizza objects with various configurations, taking inspiration from the pizza construction process, and an entity called Waiter to facilitate the ordering process.

1. Define a Pizza class with attributes such as dough, sauce and topping.
2. Implement a PizzaBuilder class with methods to set different attributes of a Pizza object.
3. Create a Waiter class responsible for taking orders and using the PizzaBuilder to construct Pizza objects based on customer preferences.
4. Demonstrate the usage of the Waiter class to take orders and construct Pizza objects (ex. `HawaiianPizza, SpicyPizza, ...`)

- **30 minutes** to solve this problem and submit the code in the elearning:

`https://elearning.ua.pt/mod/assign/view.php?id=1408428`

Building Objects with Multiple Constructors

Some objects may have multiple constructors to accommodate different combinations of parameters.

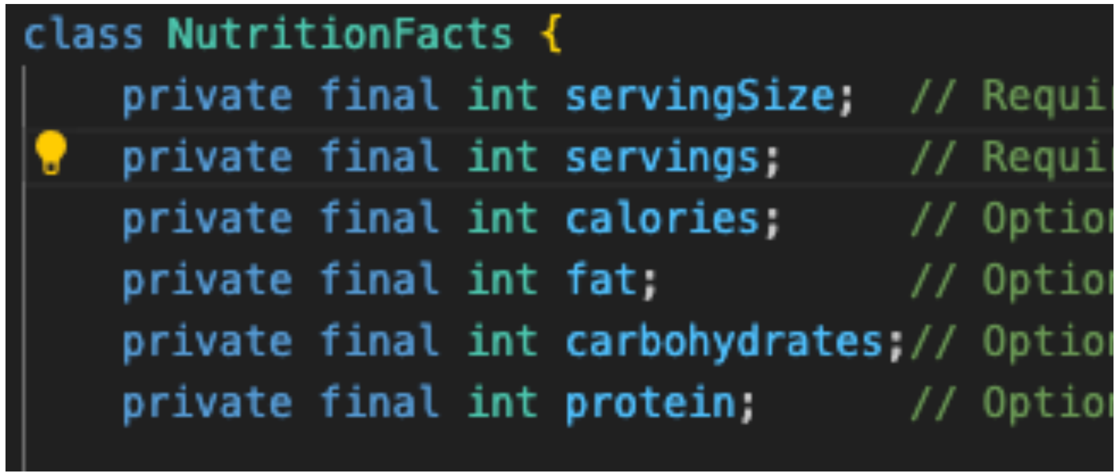
Having multiple constructors can lead to constructor overloading and complexity, especially with a large number of parameters or optional parameters.

With the Builder pattern, a separate builder class is responsible for constructing the object step by step, allowing for the selective configuration of object properties.

Clients can use the builder to specify only the desired properties and leave the rest with default values, simplifying object creation and enhancing code readability.

1. Constructor with only required parameters:
   - `public NutritionFacts(int servingSize, int servings)`
2. Constructor with required and one optional parameter:
   - `public NutritionFacts(int servingSize, int servings, int calories)`
   - `public NutritionFacts(int servingSize, int servings, int fat)`
   - `public NutritionFacts(int servingSize, int servings, int carbohydrates)`
   - `public NutritionFacts(int servingSize, int servings, int protein)`
3. Constructors with required and two optional parameters:
   - `public NutritionFacts(int servingSize, int servings, int calories, int fat)`
   - `public NutritionFacts(int servingSize, int servings, int calories, int carbohydrates)`
   - `public NutritionFacts(int servingSize, int servings, int calories, int protein)`
   - `public NutritionFacts(int servingSize, int servings, int fat, int carbohydrates)`
   - `public NutritionFacts(int servingSize, int servings, int fat, int protein)`
   - `public NutritionFacts(int servingSize, int servings, int carbohydrates, int protein)`
4. Constructors with required and three optional parameters:
   - `public NutritionFacts(int servingSize, int servings, int calories, int fat, int carbohydrates)`
   - `public NutritionFacts(int servingSize, int servings, int calories, int fat, int protein)`
   - `public NutritionFacts(int servingSize, int servings, int calories, int carbohydrates, int protein)`
   - `public NutritionFacts(int servingSize, int servings, int fat, int carbohydrates, int protein)`

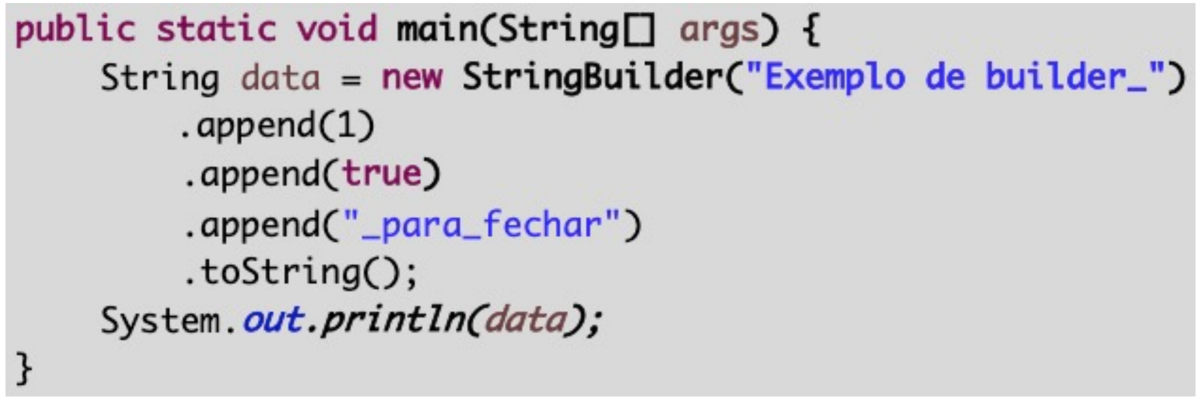
Example



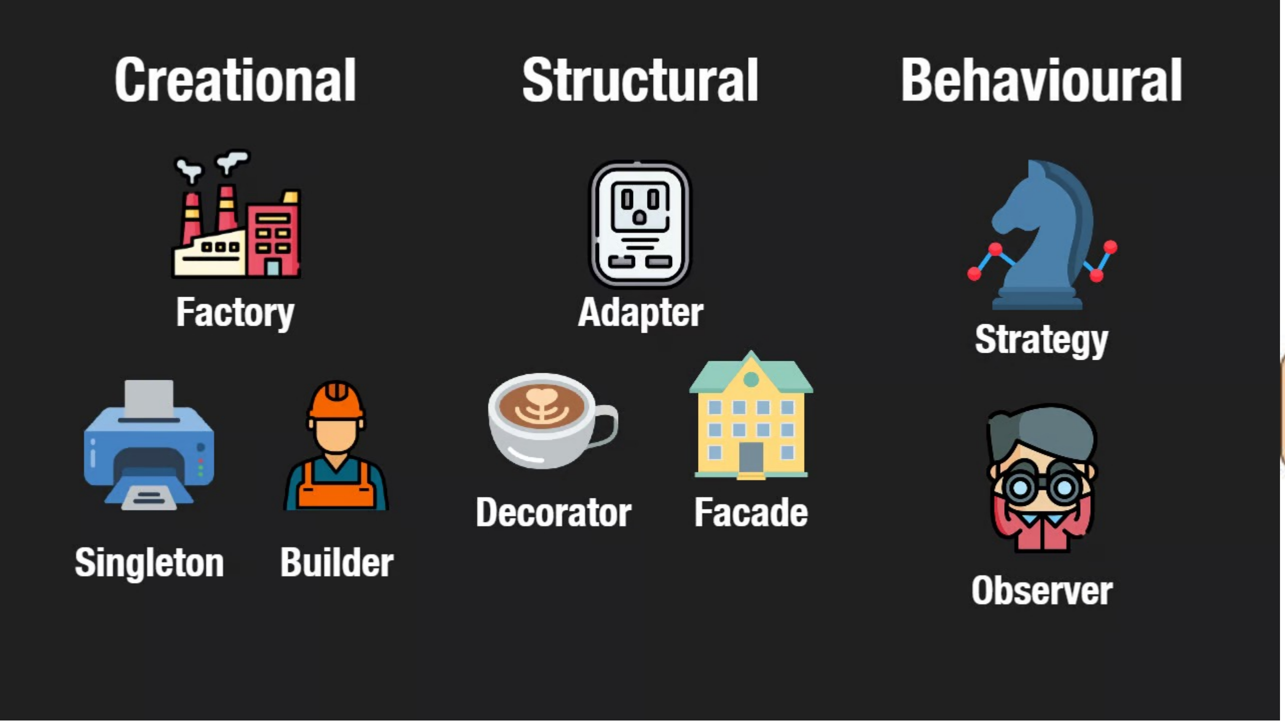
```
class NutritionFacts {
    private final int servingSize;   // Requi
    private final int servings;      // Requi
    private final int calories;      // Optio
    private final int fat;           // Optio
    private final int carbohydrates;// Optio
    private final int protein;       // Optio
```

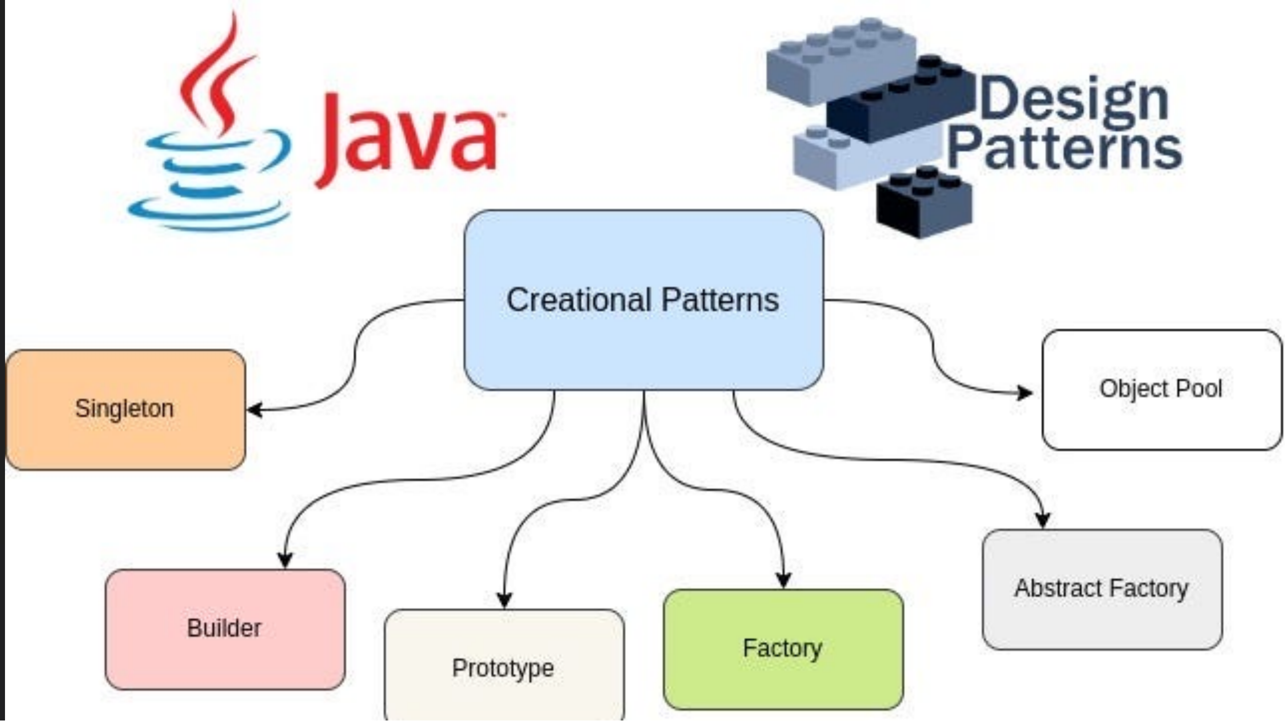
Builder Pattern in Java Development Kit (JDK)

- Several JDK classes and libraries utilize the Builder pattern:
  - StringBuilder: Building strings with fluent interface.
  - java.util.stream.Stream.Builder: Constructing streams with ease.
  - javax.swing.JOptionPane: Creating dialogs for user interaction.
  - java.nio.file.Path.Builder: Building file paths in a structured manner.



```
public static void main(String[] args) {
    String data = new StringBuilder("Exemplo de builder_")
        .append(1)
        .append(true)
        .append("_para_fechar")
        .toString();
    System.out.println(data);
}
```





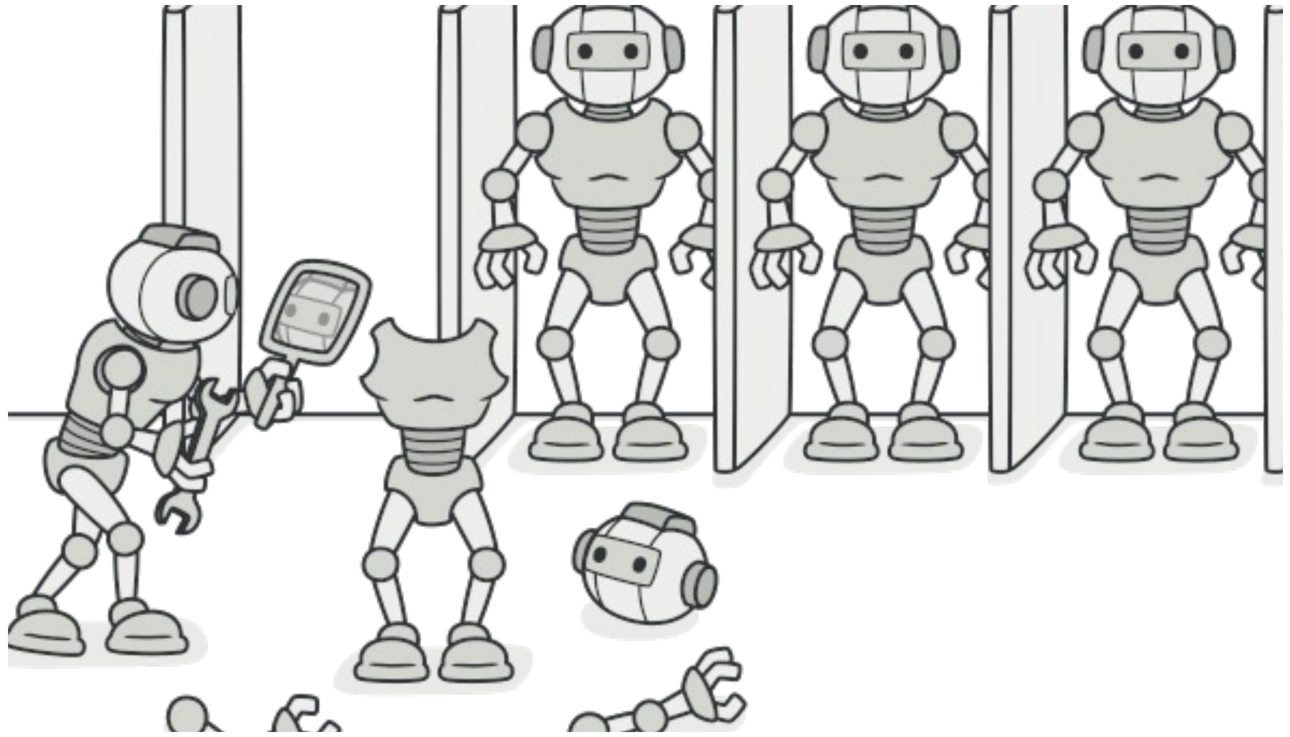
Creational Design Patterns

- **15 minutes** to explore the Prototype Design Pattern and answer the questions in the link:

`https://forms.gle/Lid61HArYXKUXj4K6`

Introduction to Prototype Pattern

- Definition: It allows creating new objects based on an existing object by copying its state.
- Purpose:
    - Facilitates object creation without specifying their exact class.
    - Reduces subclassing by using cloning to create new instances.
    - Improves performance by avoiding costly object creation operations.



Implementation of Prototype Pattern

Prototype Interface:

- Defines methods for cloning objects.
- Typically includes a clone method to create a copy of the object.

Concrete Prototypes:

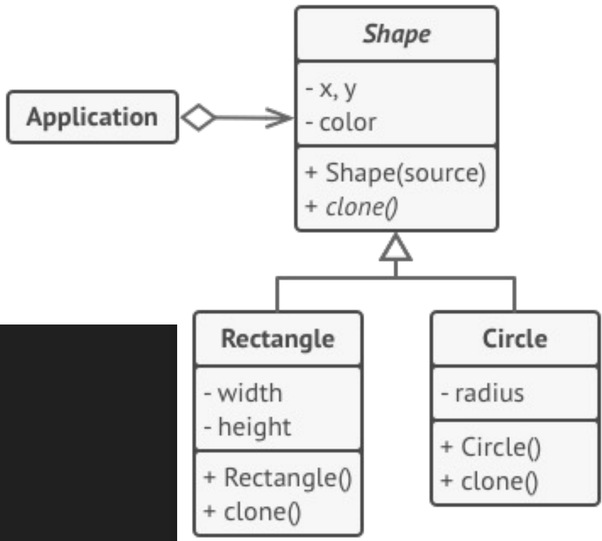
- Concrete classes implementing the prototype interface.
- Each concrete prototype represents a specific type of object.

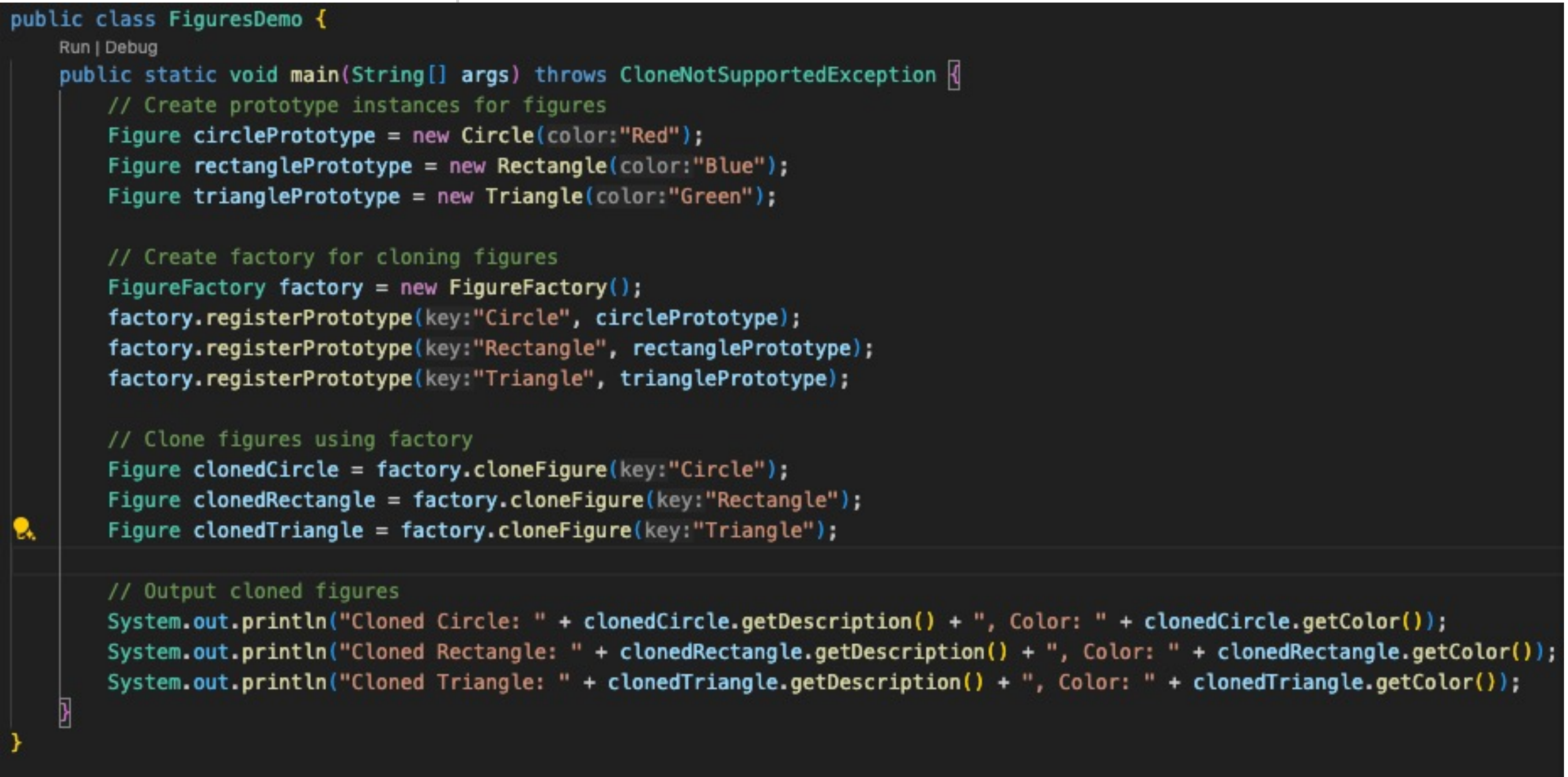
Object Cloning:

- Objects are cloned using either shallow or deep cloning techniques.
- Shallow cloning copies the object's references, while deep cloning creates new instances of referenced objects.

An object that supports cloning is called a prototype. When your objects have dozens of fields and hundreds of possible configurations, cloning them might serve as an alternative to subclassing.

Example





```java
public class FiguresDemo {
    Run | Debug
    public static void main(String[] args) throws CloneNotSupportedException {
        // Create prototype instances for figures
        Figure circlePrototype = new Circle(color:"Red");
        Figure rectanglePrototype = new Rectangle(color:"Blue");
        Figure trianglePrototype = new Triangle(color:"Green");

        // Create factory for cloning figures
        FigureFactory factory = new FigureFactory();
        factory.registerPrototype(key:"Circle", circlePrototype);
        factory.registerPrototype(key:"Rectangle", rectanglePrototype);
        factory.registerPrototype(key:"Triangle", trianglePrototype);

        // Clone figures using factory
        Figure clonedCircle = factory.cloneFigure(key:"Circle");
        Figure clonedRectangle = factory.cloneFigure(key:"Rectangle");
        Figure clonedTriangle = factory.cloneFigure(key:"Triangle");

        // Output cloned figures
        System.out.println("Cloned Circle: " + clonedCircle.getDescription() + ", Color: " + clonedCircle.getColor());
        System.out.println("Cloned Rectangle: " + clonedRectangle.getDescription() + ", Color: " + clonedRectangle.getColor());
        System.out.println("Cloned Triangle: " + clonedTriangle.getDescription() + ", Color: " + clonedTriangle.getColor());
    }
}
```