Design Patterns

Serban Scorteanu

Vlad Totea

Group 1100

Contents

What are design patterns?

Design patterns = typical solutions to frequently occurring software design issues. They are not specific fragments of code, but instead they are concepts for solving specific problems

Usually, patterns are described in this manner:

Intent – this is where both the problem and the solution are described

Motivation – this is where the details of the problem and the solution are presented

Structure – the structure of the classes show how they are related and how they make up the pattern

Example – the actual code in order to prove the new solution and better understand the concept

Advantages and disadvantages

ADVANTAGES

1. Reusability
2. Well-proved, well-tested solutions
3. Clarity and better system architecture
4. Better communication between developers

DISADVANTAGES

1. Complex
2. Seem to be simple but not really
3. Decrease understandability (by increasing code length and by using indirect flow – this indirection might actually reduce performance)
4. Can use more memory

Main Classification

A. **Creational Design Patterns** (most frequently used)
   1) Factory
   2) Abstract Factory
   3) Singleton
   4) Prototype
   5) Builder

B. **Structural Design Patterns**
   1) Adapter
   2) Bridge
   3) Composite
   4) Decorator
   5) Facade
   6) Flyweight
   7) Proxy

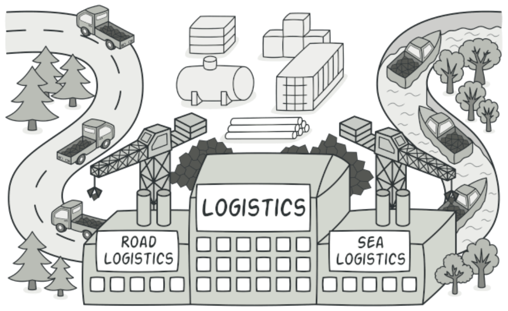
C. **Behavioral Design Patterns**
   1. Chain of Responsibility
   2. Command
   3. Interpreter
   4. Iterator
   5. Mediator
   6. Memento
   7. Observer
   8. State
   9. Strategy
   10. Template
   11. Visitor

Creational Design Patterns

- ### Factory

  The factory design pattern provides an interface for creating objects only in the parent class but gives permission to children classes to alter the type of objects which will be created.
  In Factory – objects are created without exposing the logic to the user and the user will use the common interface to create new types of objects.

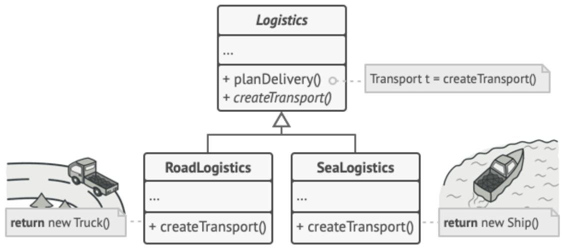
  The implementation consists in creating a static factory method that will create and return instances.

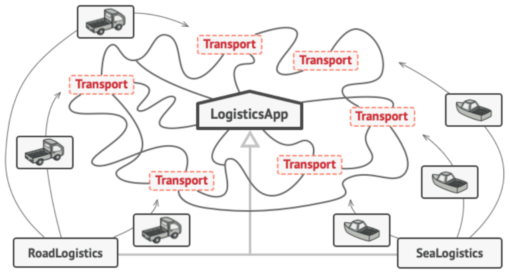


Best example of a problem and why this pattern was created:
We need to create a logistics application. The app was designed to handle truck transportation => most of the code is inside the Truck class. The app becomes so popular that sea transportation companies want to become our clients.

Problem – most of the code is inside the Truck class, so adding Ships would require a lot of refactoring. Sometime after refactoring air transportation companies want to join.

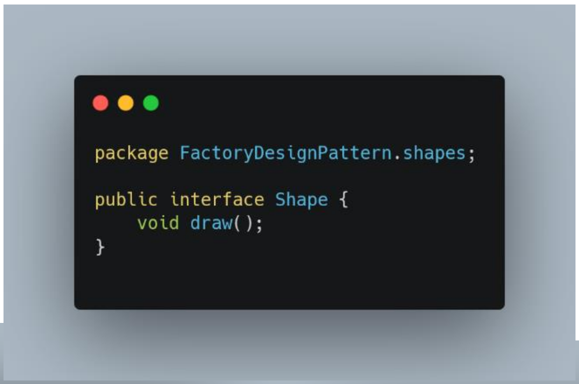
Solution – replace direct object ctor calls with calls to a factory method which will be included in this method.



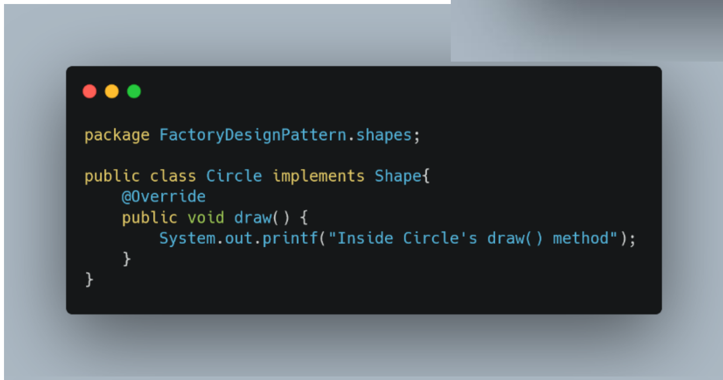


When should we use the Factory Pattern?

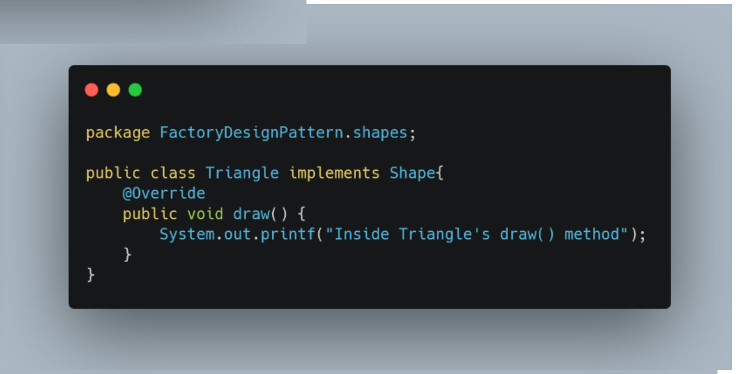
- When an abstract class / interface – expected to change frequently
- When we expect future changes and modify our current implementation would be tedious
- When the initialization is simple and the ctor needs few params



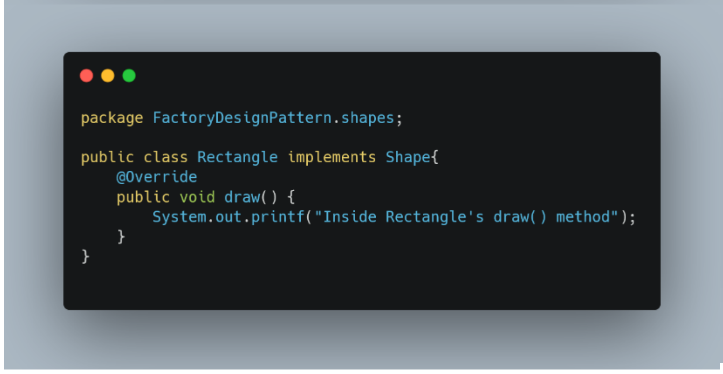
```
package FactoryDesignPattern.shapes;

public interface Shape {
    void draw();
}
```



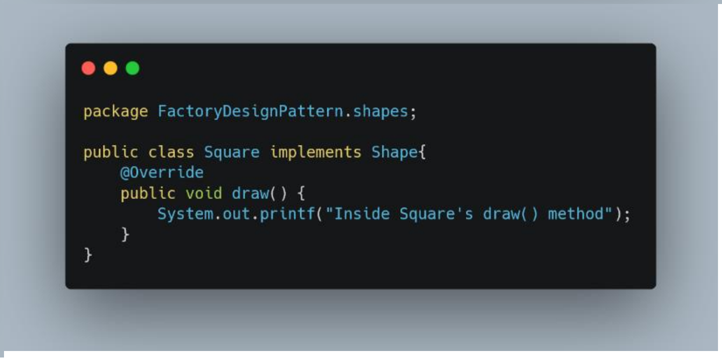
```
package FactoryDesignPattern.shapes;

public class Circle implements Shape{
    @Override
    public void draw() {
        System.out.printf("Inside Circle's draw() method");
    }
}
```



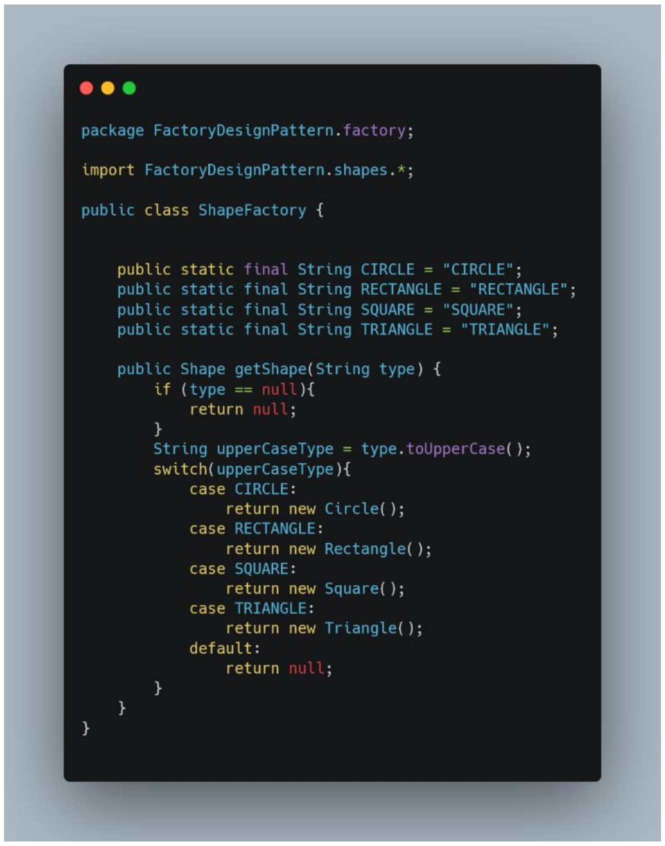
```
package FactoryDesignPattern.shapes;

public class Triangle implements Shape{
    @Override
    public void draw() {
        System.out.printf("Inside Triangle's draw() method");
    }
}
```



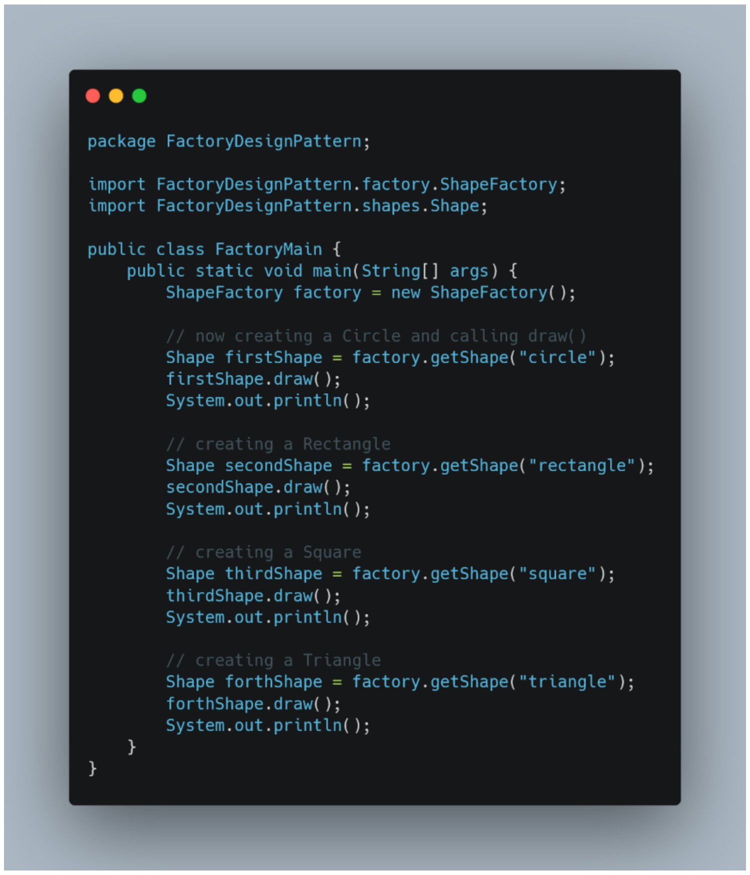
```
package FactoryDesignPattern.shapes;

public class Rectangle implements Shape{
    @Override
    public void draw() {
        System.out.printf("Inside Rectangle's draw() method");
    }
}
```



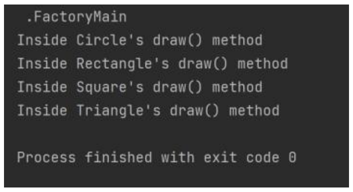
```
package FactoryDesignPattern.shapes;

public class Square implements Shape{
    @Override
    public void draw() {
        System.out.printf("Inside Square's draw() method");
    }
}
```



```java
package FactoryDesignPattern.factory;

import FactoryDesignPattern.shapes.*;

public class ShapeFactory {


    public static final String CIRCLE = "CIRCLE";
    public static final String RECTANGLE = "RECTANGLE";
    public static final String SQUARE = "SQUARE";
    public static final String TRIANGLE = "TRIANGLE";

    public Shape getShape(String type) {
        if (type == null){
            return null;
        }
        String upperCaseType = type.toUpperCase();
        switch(upperCaseType){
            case CIRCLE:
                return new Circle();
            case RECTANGLE:
                return new Rectangle();
            case SQUARE:
                return new Square();
            case TRIANGLE:
                return new Triangle();
            default:
                return null;
        }
    }
}
```



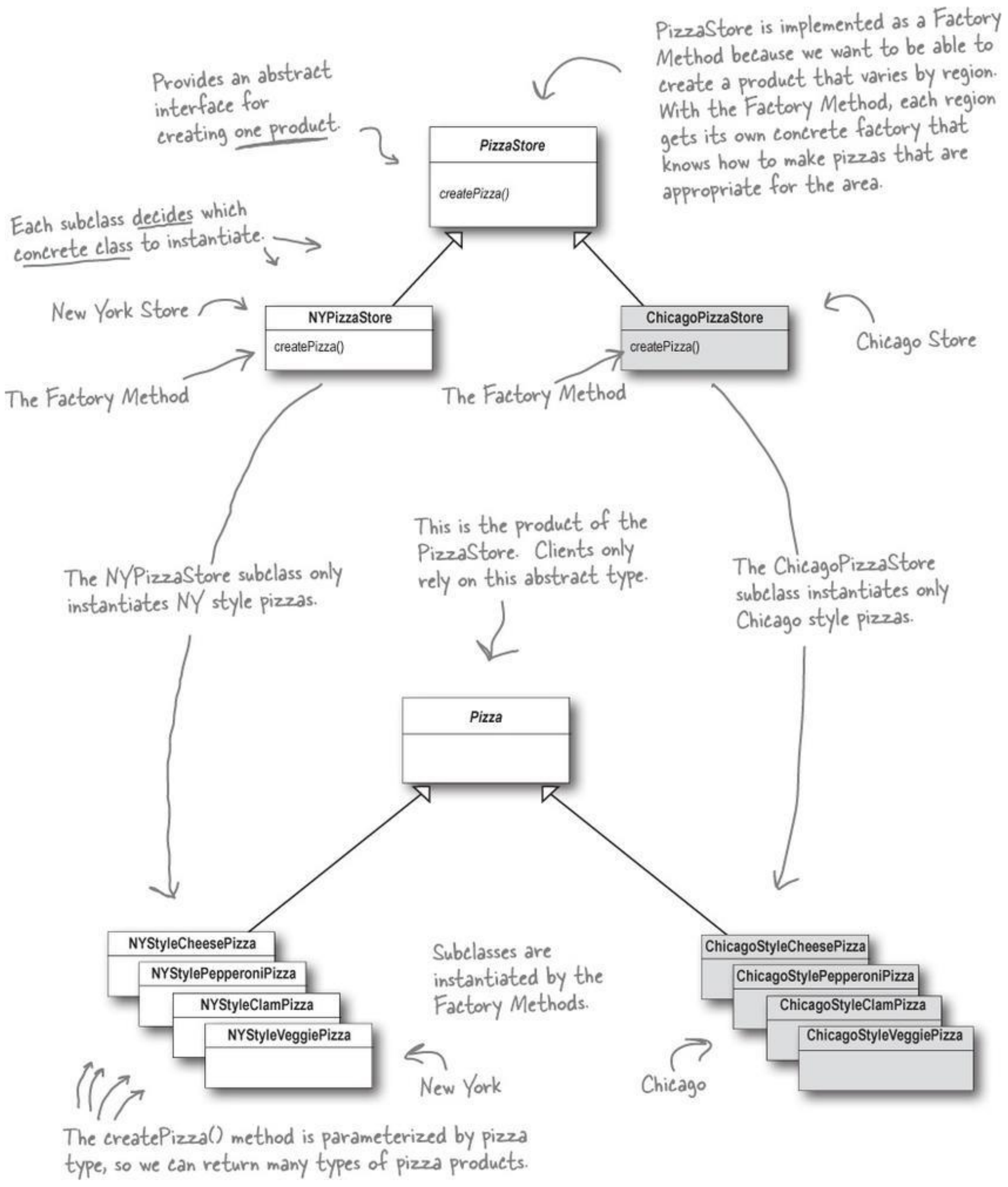
```java
package FactoryDesignPattern;

import FactoryDesignPattern.factory.ShapeFactory;
import FactoryDesignPattern.shapes.Shape;

public class FactoryMain {
    public static void main(String[] args) {
        ShapeFactory factory = new ShapeFactory();

        // now creating a Circle and calling draw()
        Shape firstShape = factory.getShape("circle");
        firstShape.draw();
        System.out.println();

        // creating a Rectangle
        Shape secondShape = factory.getShape("rectangle");
        secondShape.draw();
        System.out.println();

        // creating a Square
        Shape thirdShape = factory.getShape("square");
        thirdShape.draw();
        System.out.println();

        // creating a Triangle
        Shape forthShape = factory.getShape("triangle");
        forthShape.draw();
        System.out.println();
    }
}
```

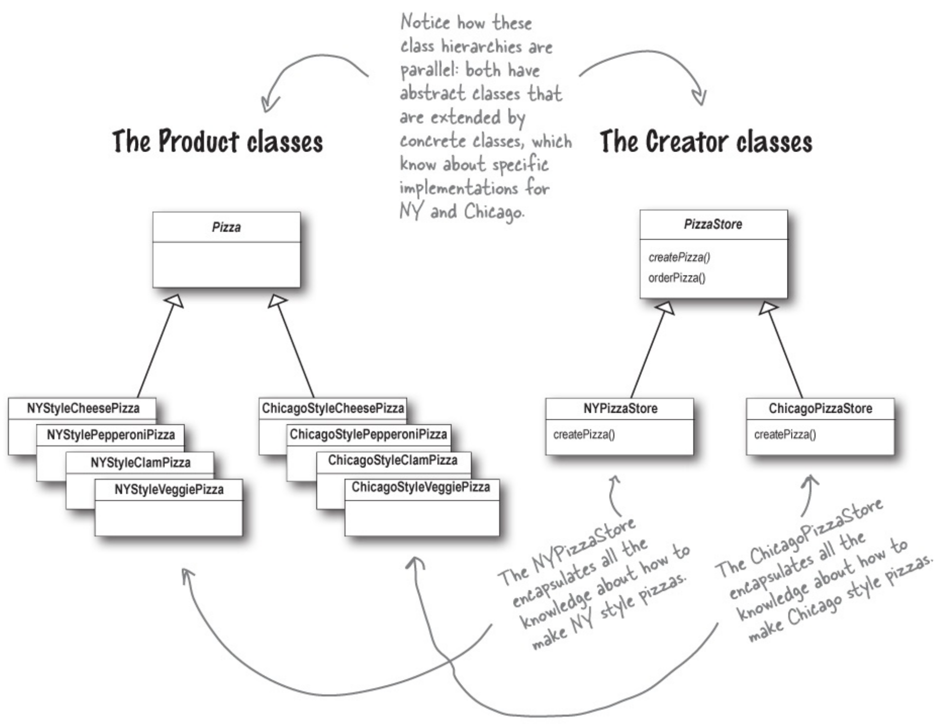


```
 .FactoryMain
Inside Circle's draw() method
Inside Rectangle's draw() method
Inside Square's draw() method
Inside Triangle's draw() method

Process finished with exit code 0
```

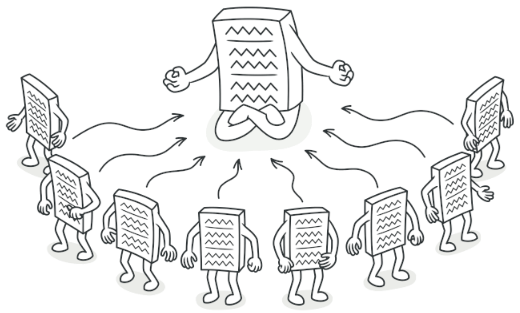
The factory could have also had children and we could have overridden the createShape() method inside these children in order to have even more control.

Another personal favourite example from Head First Design Patterns:


PizzaStore is implemented as a Factory Method because we want to be able to create a product that varies by region. With the Factory Method, each region gets its own concrete factory that knows how to make pizzas that are appropriate for the area.
Provides an abstract interface for creating one product.
PizzaStore
createPizza()
Each subclass decides which concrete class to instantiate.
New York Store
NYPizzaStore
createPizza()
The Factory Method
ChicagoPizzaStore
createPizza()
Chicago Store
The Factory Method
This is the product of the PizzaStore. Clients only rely on this abstract type.
The NYPizzaStore subclass only instantiates NY style pizzas.
The ChicagoPizzaStore subclass instantiates only Chicago style pizzas.
Pizza
NYStyleCheesePizza
NYStylePepperoniPizza
NYStyleClamPizza
NYStyleVeggiePizza
Subclasses are instantiated by the Factory Methods.
New York
Chicago
ChicagoStyleCheesePizza
ChicagoStylePepperoniPizza
ChicagoStyleClamPizza
ChicagoStyleVeggiePizza
The createPizza() method is parameterized by pizza type, so we can return many types of pizza products.

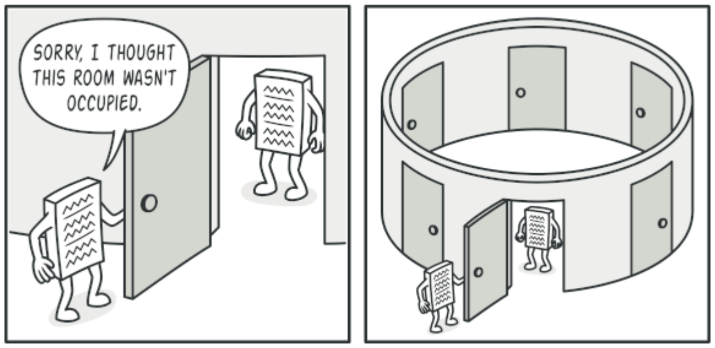

Notice how these class hierarchies are parallel: both have abstract classes that are extended by concrete classes, which know about specific implementations for NY and Chicago.
The Product classes
The Creator classes
Pizza
PizzaStore
createPizza()
orderPizza()
NYStyleCheesePizza
NYStylePepperoniPizza
NYStyleClamPizza
NYStyleVeggiePizza
ChicagoStyleCheesePizza
ChicagoStylePepperoniPizza
ChicagoStyleClamPizza
ChicagoStyleVeggiePizza
NYPizzaStore
createPizza()
ChicagoPizzaStore
createPizza()
The NYPizzaStore encapsulates all the knowledge about how to make NY style pizzas.
The ChicagoPizzaStore encapsulates all the knowledge about how to make Chicago style pizzas.

- ## Singleton



Design pattern which ensures that a class has only one instance and also provides global access to that instance.
Best use cases: when we have a shared resource – databases, files.

Remember SOLID principles? Well, the Singleton pattern violates the Single Responsibility Principle.
The idea is simple. Imagine we have to create an object and after a while we want to create a new one. Even though we think we get a new one, we actually get a reference to the original one. This can't be done with a regular ctor.

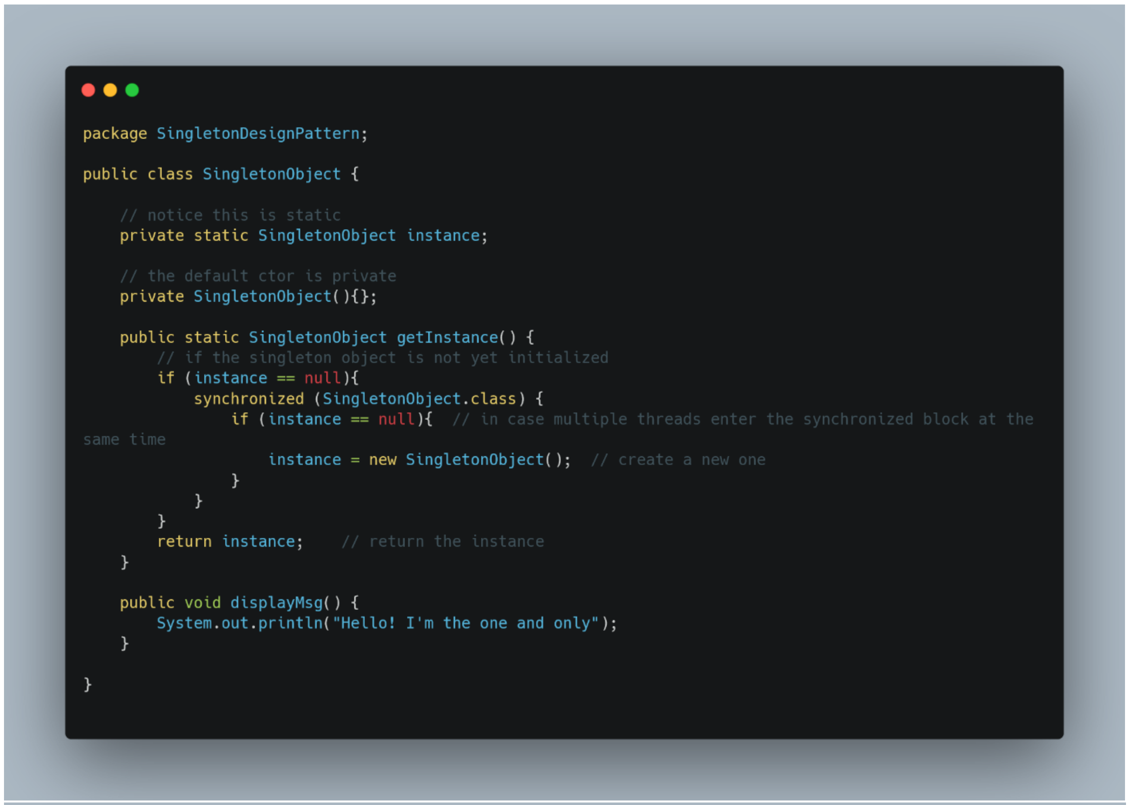


Clients may not even realize that they're working with the same object all the time.

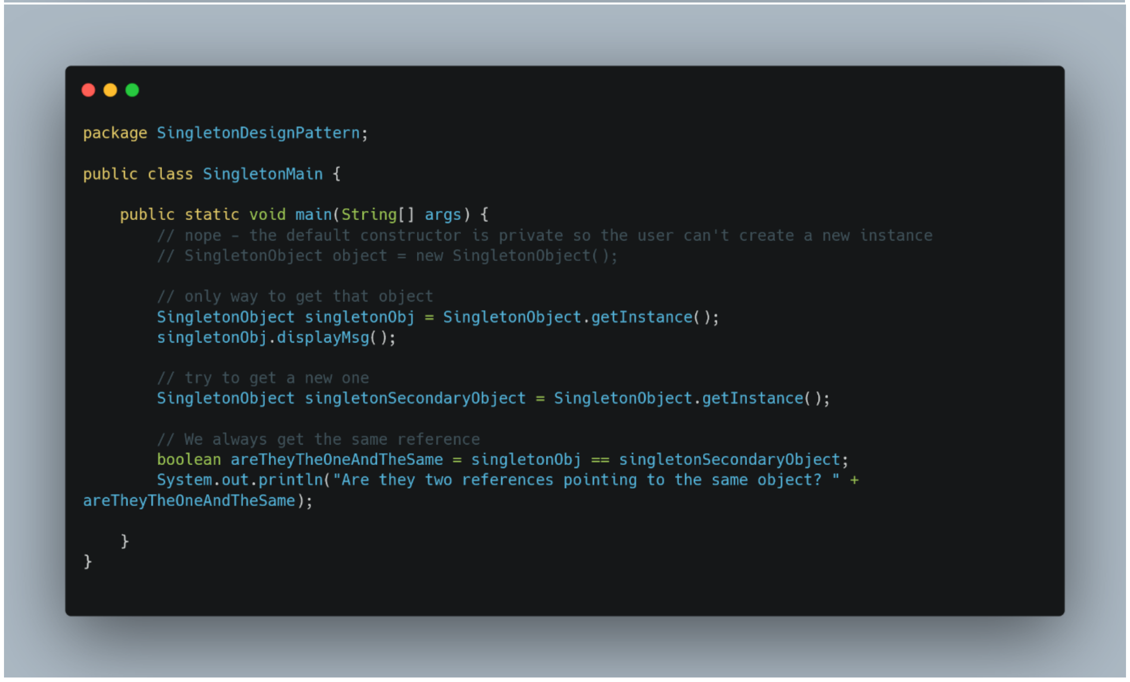
This pattern is so popular some frameworks use it by default. Ex: Spring (Java's most popular framework) uses it by default for all its beans. (there's an extra step needed in order to change the scope of the beans to others such as prototype).

How do we implement Singleton?
2 steps:

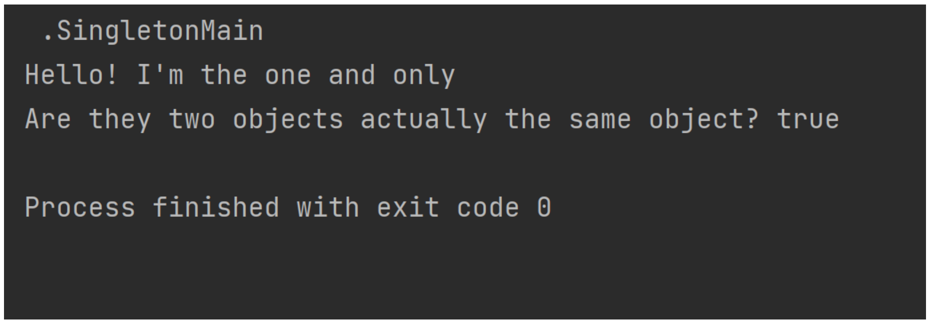
a. The default constructor must be private so that other objects can't use instantiate the singleton class
b. We have to create a static method that acts as a constructor. Behind the scenes (for the user) this method will call that private ctor and it will save it in a static field. From this point on any call to this method will return a reference to that original object.



```
package SingletonDesignPattern;

public class SingletonObject {

    // notice this is static
    private static SingletonObject instance;

    // the default ctor is private
    private SingletonObject(){};

    public static SingletonObject getInstance() {
        // if the singleton object is not yet initialized
        if (instance == null){
            synchronized (SingletonObject.class) {
                if (instance == null){  // in case multiple threads enter the synchronized block at the
same time
                    instance = new SingletonObject();  // create a new one
                }
            }
        }
        return instance;    // return the instance
    }

    public void displayMsg() {
        System.out.println("Hello! I'm the one and only");
    }

}
```

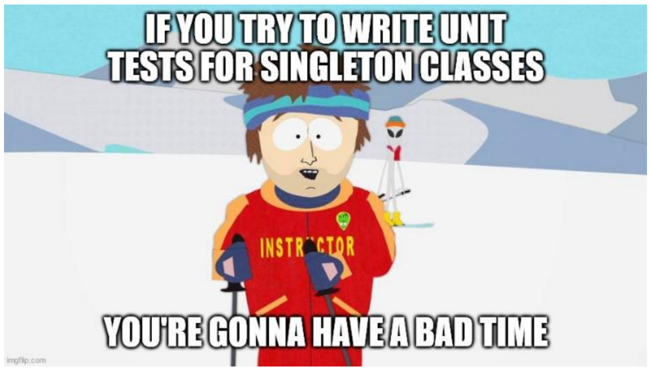


```
package SingletonDesignPattern;

public class SingletonMain {

    public static void main(String[] args) {
        // nope - the default constructor is private so the user can't create a new instance
        // SingletonObject object = new SingletonObject();

        // only way to get that object
        SingletonObject singletonObj = SingletonObject.getInstance();
        singletonObj.displayMsg();

        // try to get a new one
        SingletonObject singletonSecondaryObject = SingletonObject.getInstance();

        // We always get the same reference
        boolean areTheyTheOneAndTheSame = singletonObj == singletonSecondaryObject;
        System.out.println("Are they two references pointing to the same object? " +
areTheyTheOneAndTheSame);

    }
}
```



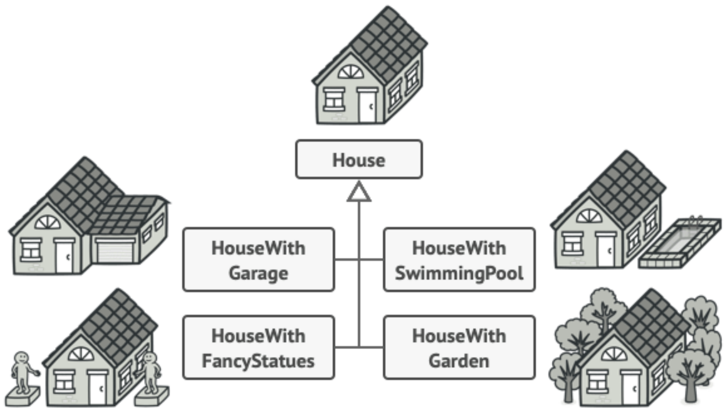
```
 .SingletonMain
Hello! I'm the one and only
Are they two objects actually the same object? true

Process finished with exit code 0
```

Conclusion: why would we use the Singleton Pattern? Well, it lets us access the same object from anywhere and most importantly -> it protects that object/instance from being overwritten somewhere.
CONS: when multithreading we need a thread lock. The biggest issue with this pattern – unit testing. Most test frameworks create mock objects using inheritance. Because the constructor is private and static methods generally can't be overridden.


IF YOU TRY TO WRITE UNIT TESTS FOR SINGLETON CLASSES
INSTRUCTOR
YOU'RE GONNA HAVE A BAD TIME
imgflip.com

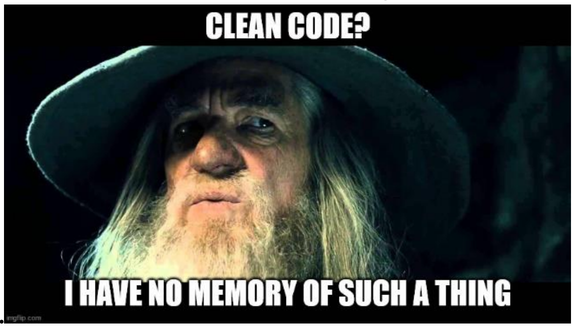
- ## Builder

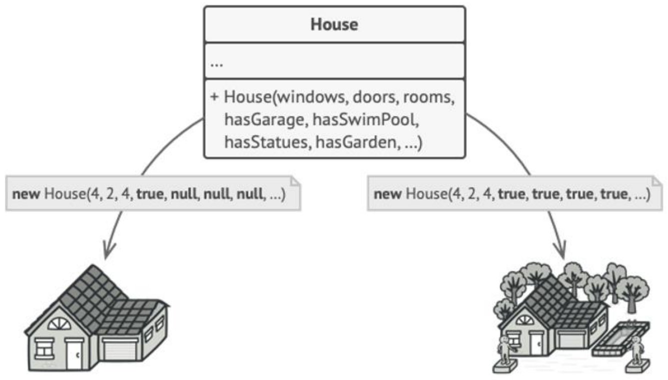
A design pattern that helps constructing large, complex objects step by step. This pattern helps us create different representations, types of an object using the same code.



You might make the program too complex by creating a subclass for every possible configuration of an object.

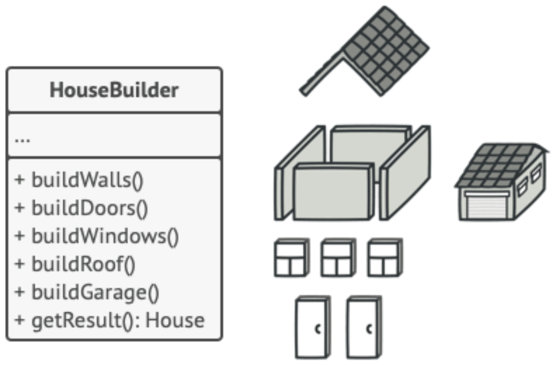
PROBLEM: We have some houses. Some houses have a garage, some don't have a garage but have swimming pools and statues, others have gardens and statues, etc. We have multiple similar objects but using inheritance would still be problematic as they have a lot of fields => a constructor with a lot of parameters which are breaking the rules of Clean Code and also prone to bugs.



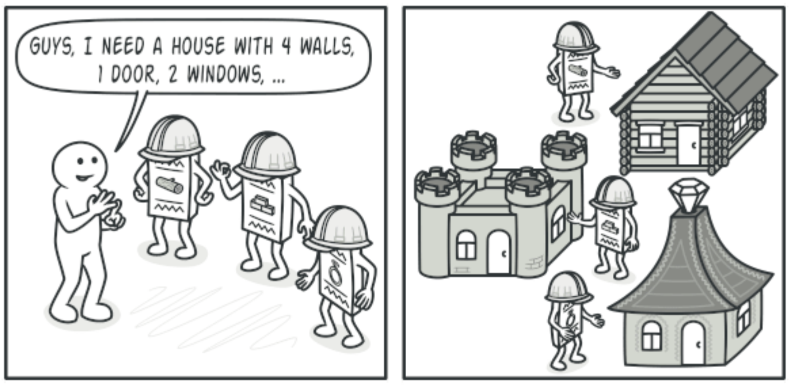

House
...
+ House(windows, doors, rooms, hasGarage, hasSwimPool, hasStatues, hasGarden, ...)
new House(4, 2, 4, true, null, null, null, ...)
new House(4, 2, 4, true, true, true, true, ...)

What does it propose? How do we do it?

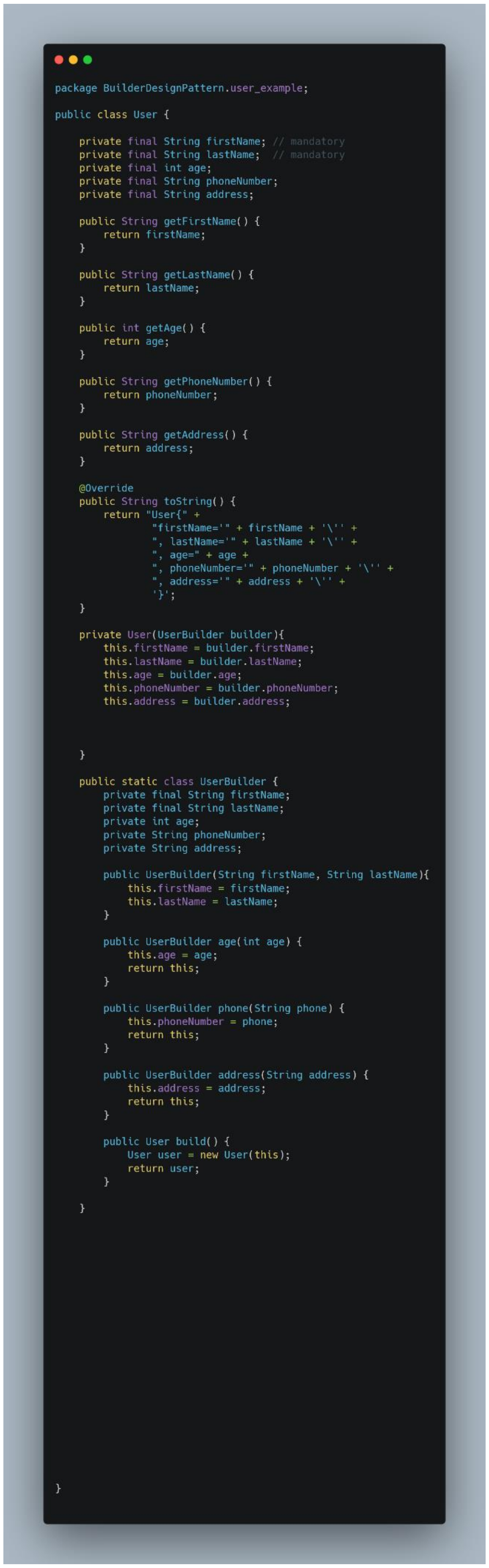
Well, the idea is to extract the object construction part out of the class and move it into so-called builder objects.



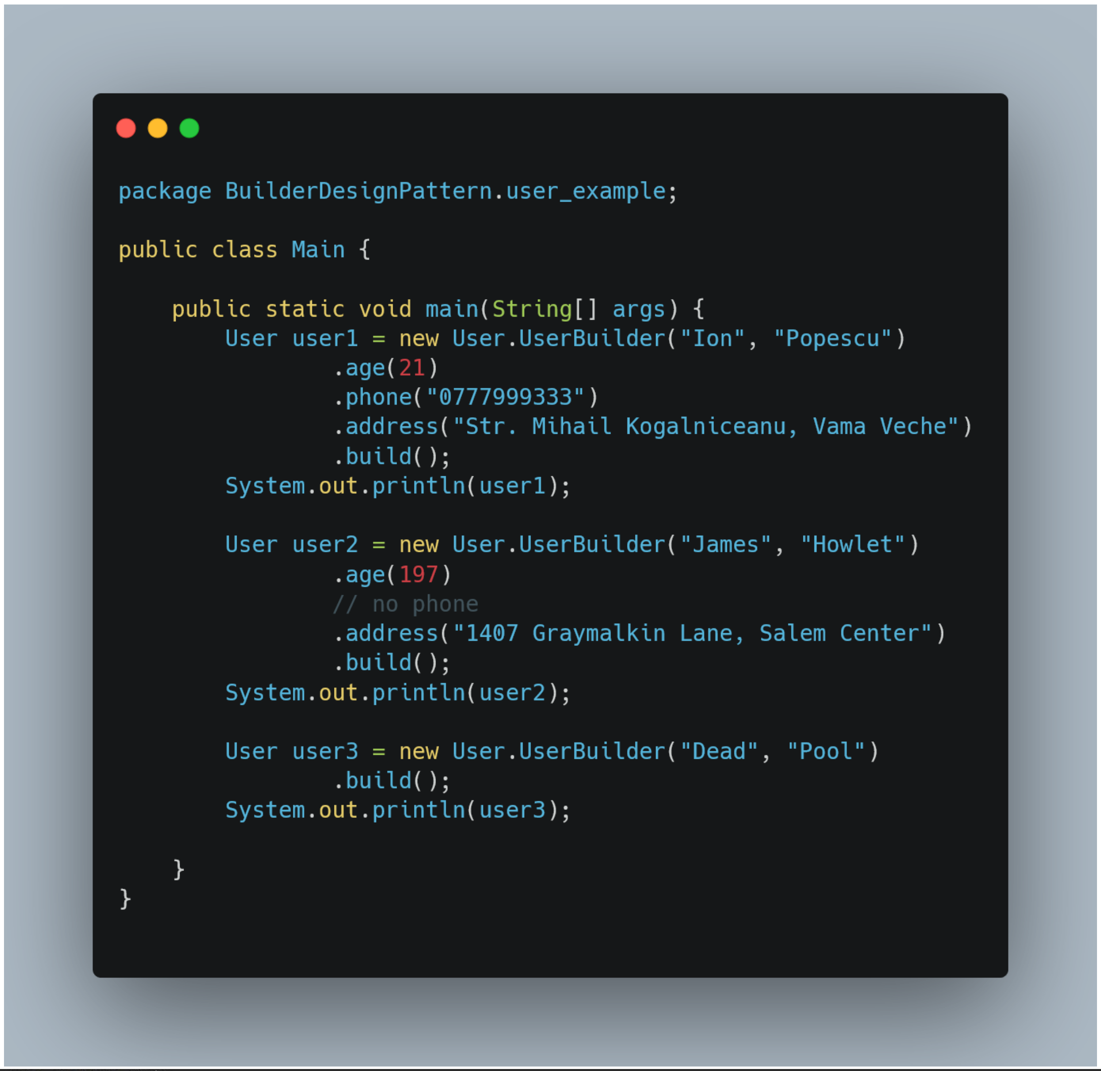
The Builder pattern lets you construct complex objects step by step. The Builder doesn't allow other objects to access the product while it's being built.



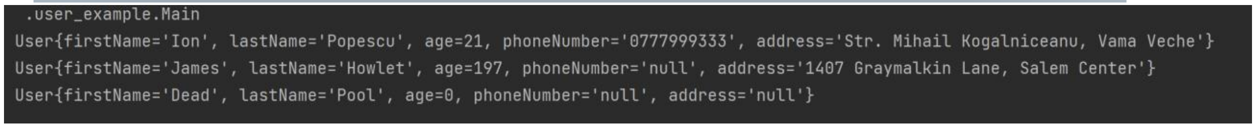
Different builders execute the same task in various ways.



```java
package BuilderDesignPattern.user_example;

public class User {

    private final String firstName; // mandatory
    private final String lastName;  // mandatory
    private final int age;
    private final String phoneNumber;
    private final String address;

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public int getAge() {
        return age;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }

    public String getAddress() {
        return address;
    }

    @Override
    public String toString() {
        return "User{" +
                "firstName='" + firstName + '\'' +
                ", lastName='" + lastName + '\'' +
                ", age=" + age +
                ", phoneNumber='" + phoneNumber + '\'' +
                ", address='" + address + '\'' +
                '}';
    }

    private User(UserBuilder builder){
        this.firstName = builder.firstName;
        this.lastName = builder.lastName;
        this.age = builder.age;
        this.phoneNumber = builder.phoneNumber;
        this.address = builder.address;



    }

    public static class UserBuilder {
        private final String firstName;
        private final String lastName;
        private int age;
        private String phoneNumber;
        private String address;

        public UserBuilder(String firstName, String lastName){
            this.firstName = firstName;
            this.lastName = lastName;
        }

        public UserBuilder age(int age) {
            this.age = age;
            return this;
        }

        public UserBuilder phone(String phone) {
            this.phoneNumber = phone;
            return this;
        }

        public UserBuilder address(String address) {
            this.address = address;
            return this;
        }

        public User build() {
            User user = new User(this);
            return user;
        }

    }
















}
```



```java
package BuilderDesignPattern.user_example;

public class Main {

    public static void main(String[] args) {
        User user1 = new User.UserBuilder("Ion", "Popescu")
                .age(21)
                .phone("0777999333")
                .address("Str. Mihail Kogalniceanu, Vama Veche")
                .build();
        System.out.println(user1);

        User user2 = new User.UserBuilder("James", "Howlet")
                .age(197)
                // no phone
                .address("1407 Graymalkin Lane, Salem Center")
                .build();
        System.out.println(user2);

        User user3 = new User.UserBuilder("Dead", "Pool")
                .build();
        System.out.println(user3);

    }
}
```



```
 .user_example.Main
User{firstName='Ion', lastName='Popescu', age=21, phoneNumber='0777999333', address='Str. Mihail Kogalniceanu, Vama Veche'}
User{firstName='James', lastName='Howlet', age=197, phoneNumber='null', address='1407 Graymalkin Lane, Salem Center'}
User{firstName='Dead', lastName='Pool', age=0, phoneNumber='null', address='null'}
```

Structural Design Patterns

Structural design patterns – focus on the relationships between classes – inheritance, as well as composition, tries to simplify the overall structure.

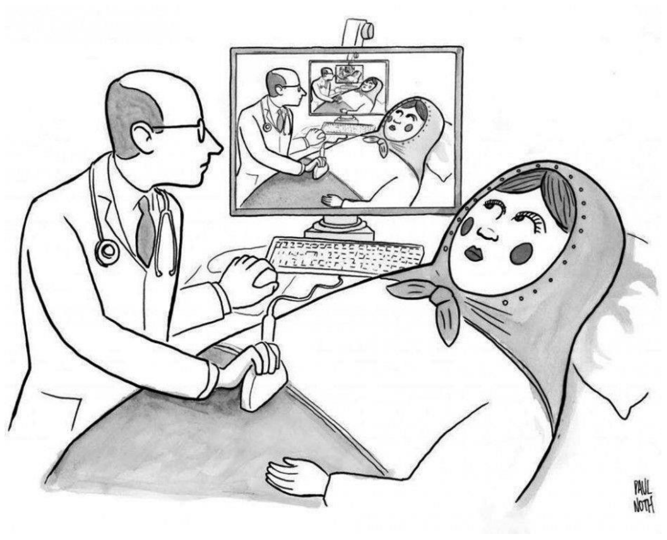
Their purpose – to simplify the design of large object structures, describing ways of composing so that classes and objects can become repeatable without refactoring each time.

The type of problem this category of patterns tries to solve – Example:

- ### Decorator

= 'Wrapper' pattern

= pattern that lets us attach new behaviours to objects by wrapping them using object which contain these behaviours



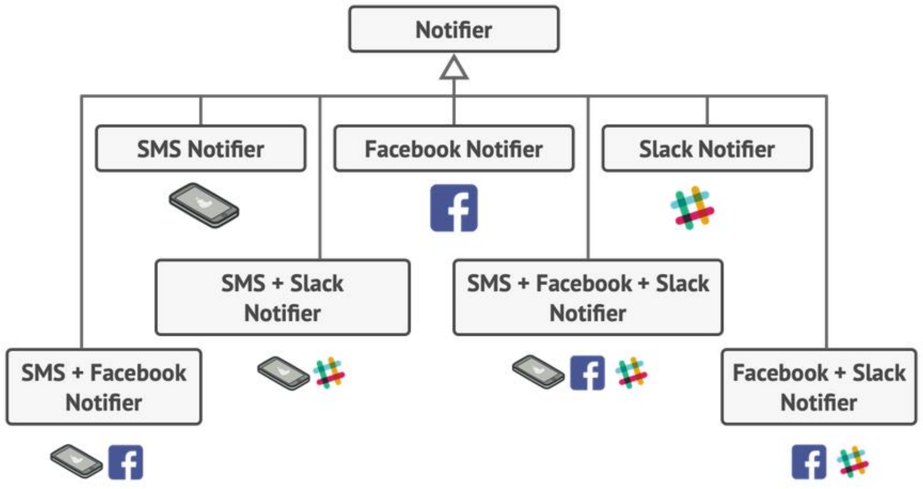
Example of problems which this pattern is meant to solve:

There's an app needed – a notification library app – which will let other apps notify users about important events.

So, this app was created, and it will take care of sending emails about these events to the users.

After a while, the 'other apps' want to also send notification via SMS/Facebook/Slack. Ok, the developers of the notifier app will create three separate classes and inherit the notifier class (code is beginning to get large). OK, done. But someone comes out and says it – What if your house is on fire? I think you would want to get notified via everything – and this is where it gets messy with inheritance.

Instead of creating hundreds of subclasses, the decorator patterns reduce that subclass count by inserting "add-on classes"



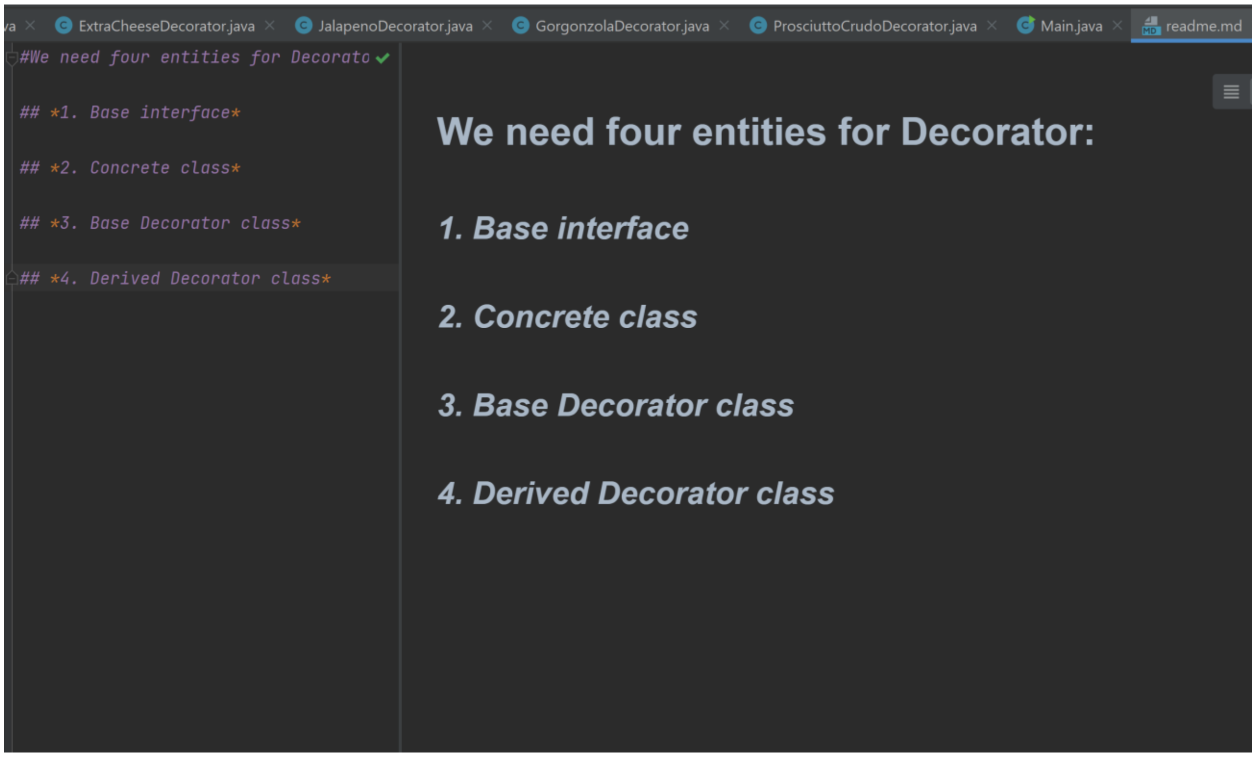
Combinatorial explosion of subclasses.

Why is inheritance bad in this case? Well – inheritance is static and can't be changed at runtime – you can only replace a whole object with another one. Also, child classes can only have one parent (at least in most programming languages)

Composition/Aggregation is the key here (and in most of the design patterns)

How do we implement it?

Implementation:



```
#We need four entities for Decorato

## *1. Base interface*

## *2. Concrete class*

## *3. Base Decorator class*

## *4. Derived Decorator class*
```

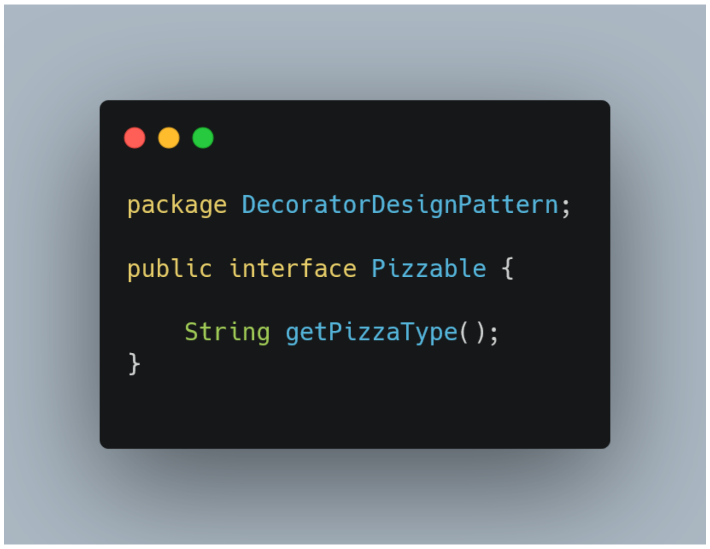
We need four entities for Decorator:

1. Base interface

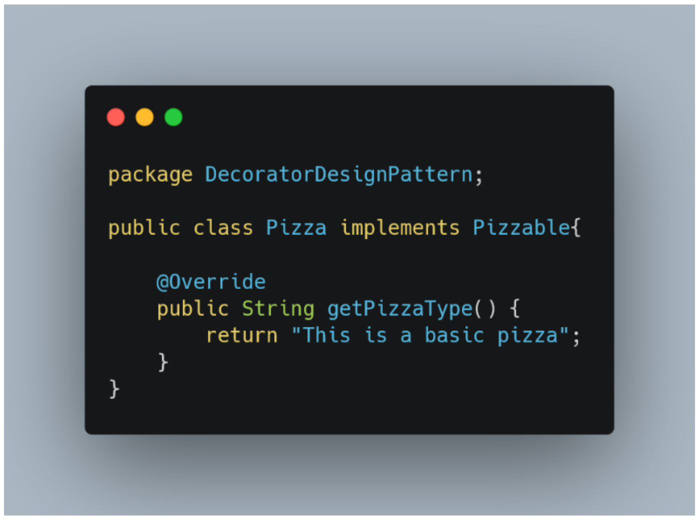
2. Concrete class

3. Base Decorator class

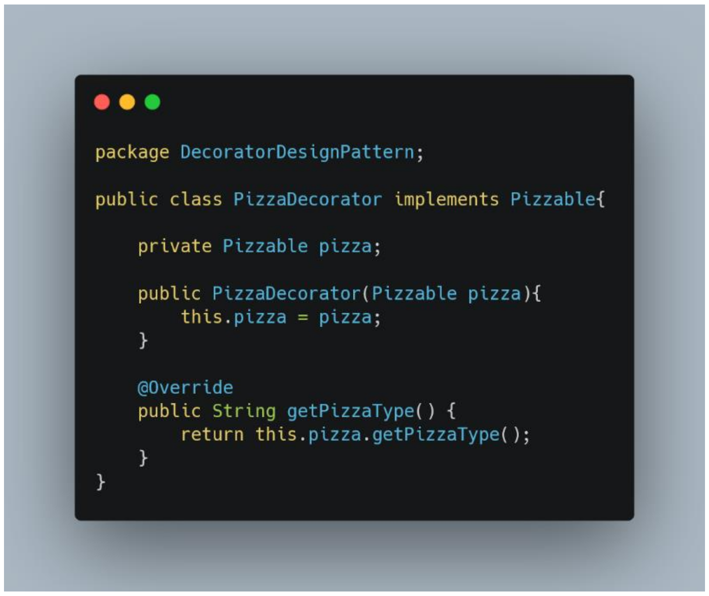
4. Derived Decorator class



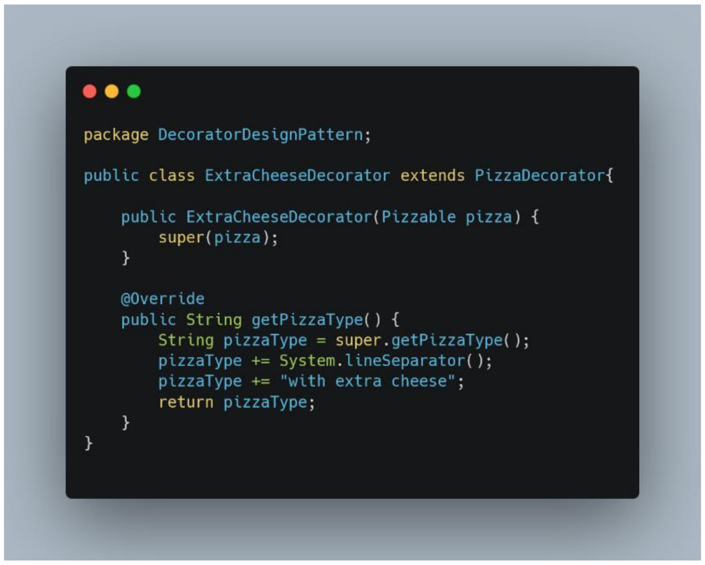
```java
package DecoratorDesignPattern;

public interface Pizzable {

    String getPizzaType();
}
```



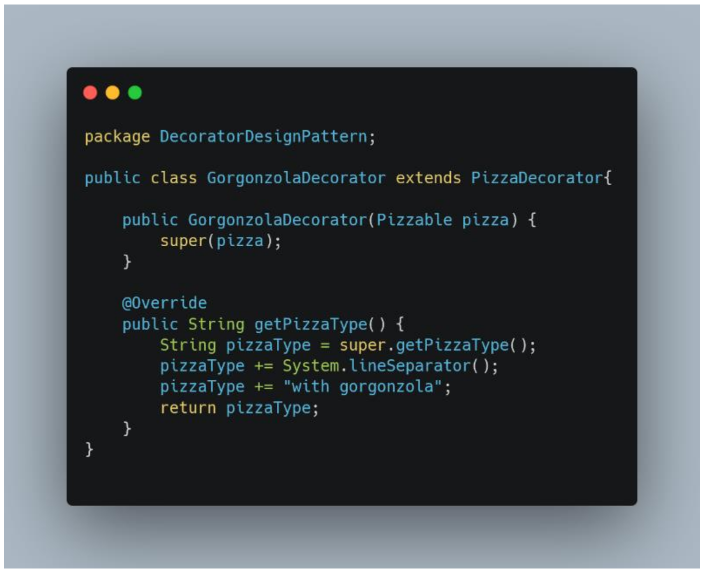
```java
package DecoratorDesignPattern;

public class Pizza implements Pizzable{

    @Override
    public String getPizzaType() {
        return "This is a basic pizza";
    }
}
```



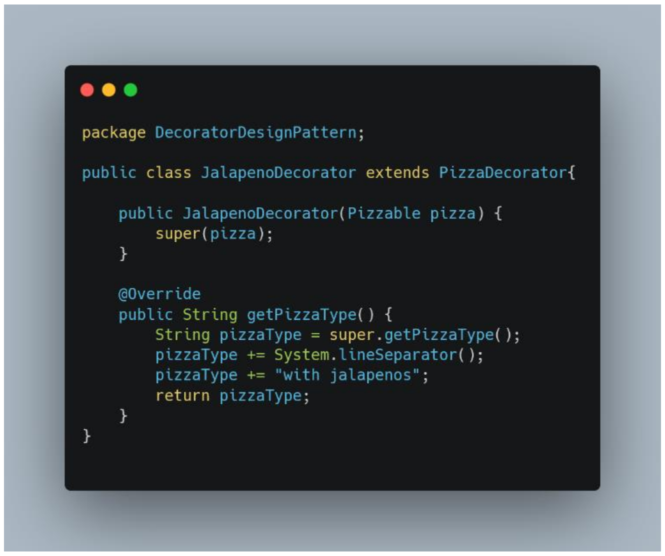
```java
package DecoratorDesignPattern;

public class PizzaDecorator implements Pizzable{

    private Pizzable pizza;

    public PizzaDecorator(Pizzable pizza){
        this.pizza = pizza;
    }

    @Override
    public String getPizzaType() {
        return this.pizza.getPizzaType();
    }
}
```



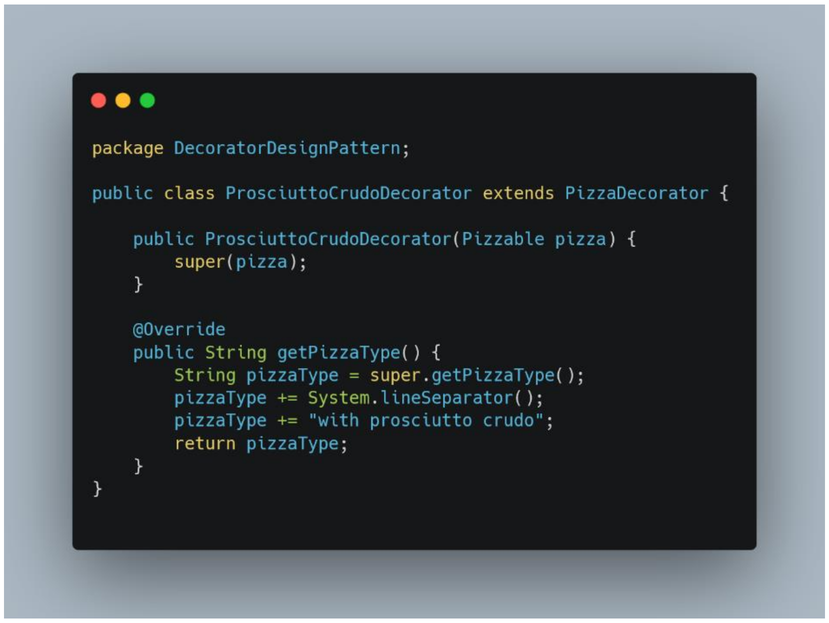
```java
package DecoratorDesignPattern;

public class ExtraCheeseDecorator extends PizzaDecorator{

    public ExtraCheeseDecorator(Pizzable pizza) {
        super(pizza);
    }

    @Override
    public String getPizzaType() {
        String pizzaType = super.getPizzaType();
        pizzaType += System.lineSeparator();
        pizzaType += "with extra cheese";
        return pizzaType;
    }
}
```



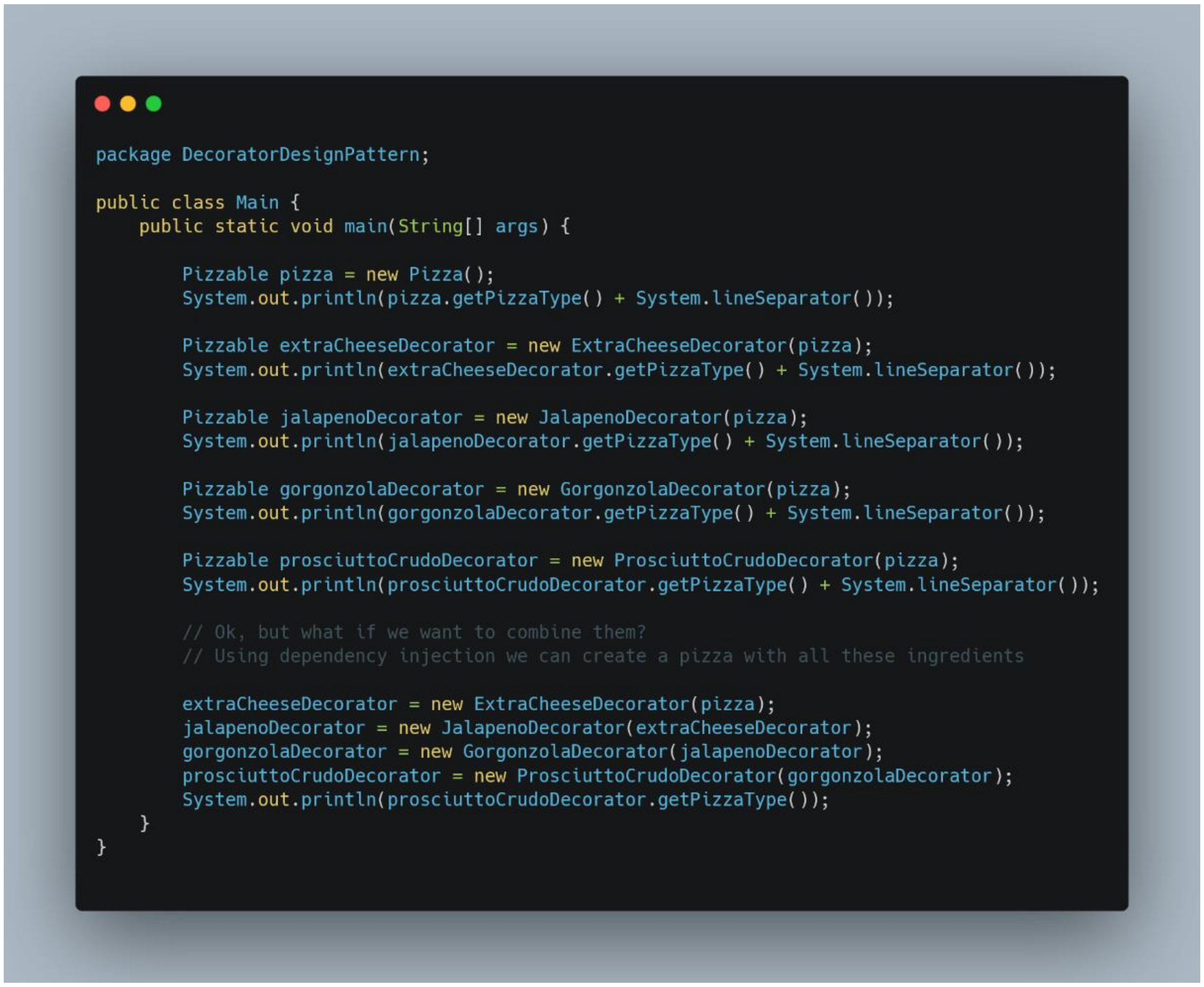
```java
package DecoratorDesignPattern;

public class GorgonzolaDecorator extends PizzaDecorator{

    public GorgonzolaDecorator(Pizzable pizza) {
        super(pizza);
    }

    @Override
    public String getPizzaType() {
        String pizzaType = super.getPizzaType();
        pizzaType += System.lineSeparator();
        pizzaType += "with gorgonzola";
        return pizzaType;
    }
}
```



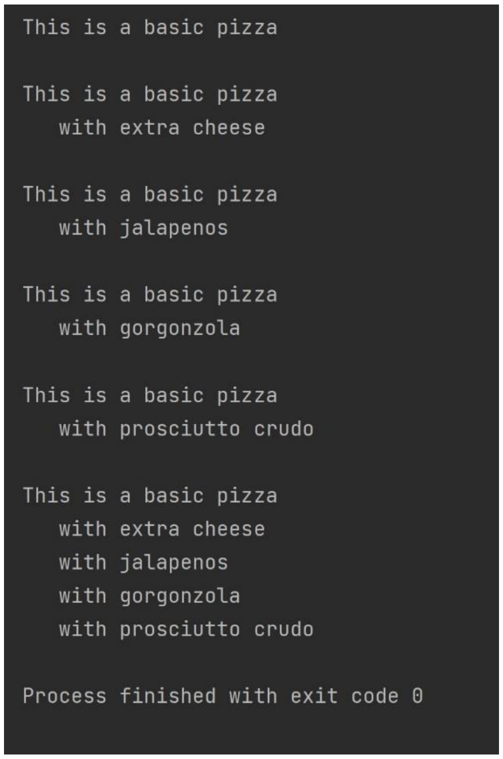
```java
package DecoratorDesignPattern;

public class JalapenoDecorator extends PizzaDecorator{

    public JalapenoDecorator(Pizzable pizza) {
        super(pizza);
    }

    @Override
    public String getPizzaType() {
        String pizzaType = super.getPizzaType();
        pizzaType += System.lineSeparator();
        pizzaType += "with jalapenos";
        return pizzaType;
    }
}
```



```java
package DecoratorDesignPattern;

public class ProsciuttoCrudoDecorator extends PizzaDecorator {

    public ProsciuttoCrudoDecorator(Pizzable pizza) {
        super(pizza);
    }

    @Override
    public String getPizzaType() {
        String pizzaType = super.getPizzaType();
        pizzaType += System.lineSeparator();
        pizzaType += "with prosciutto crudo";
        return pizzaType;
    }
}
```



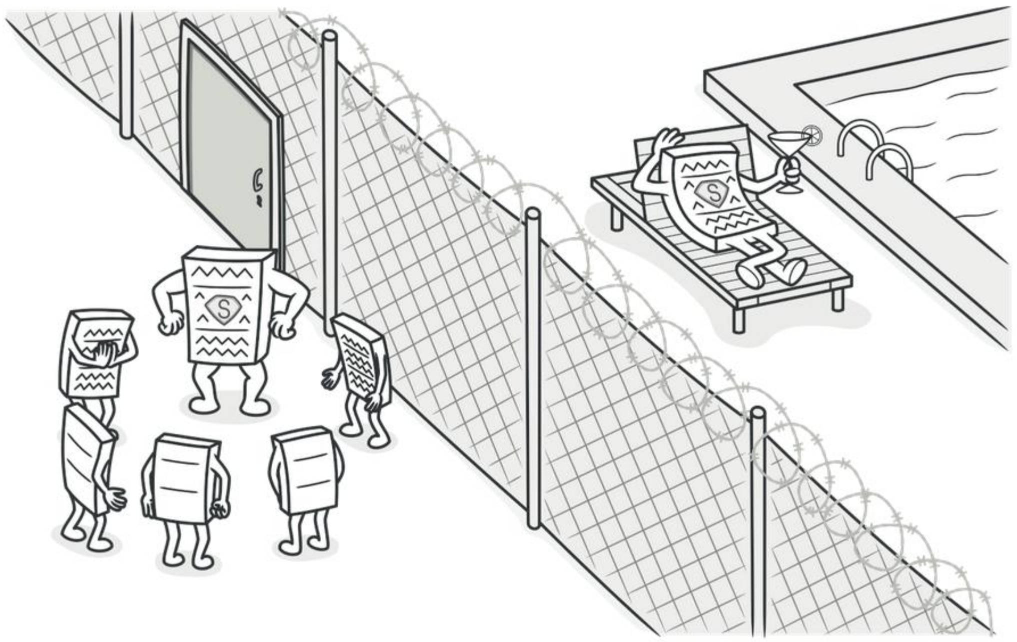
```java
package DecoratorDesignPattern;

public class Main {
    public static void main(String[] args) {

        Pizzable pizza = new Pizza();
        System.out.println(pizza.getPizzaType() + System.lineSeparator());

        Pizzable extraCheeseDecorator = new ExtraCheeseDecorator(pizza);
        System.out.println(extraCheeseDecorator.getPizzaType() + System.lineSeparator());

        Pizzable jalapenoDecorator = new JalapenoDecorator(pizza);
        System.out.println(jalapenoDecorator.getPizzaType() + System.lineSeparator());

        Pizzable gorgonzolaDecorator = new GorgonzolaDecorator(pizza);
        System.out.println(gorgonzolaDecorator.getPizzaType() + System.lineSeparator());

        Pizzable prosciuttoCrudoDecorator = new ProsciuttoCrudoDecorator(pizza);
        System.out.println(prosciuttoCrudoDecorator.getPizzaType() + System.lineSeparator());

        // Ok, but what if we want to combine them?
        // Using dependency injection we can create a pizza with all these ingredients

        extraCheeseDecorator = new ExtraCheeseDecorator(pizza);
        jalapenoDecorator = new JalapenoDecorator(extraCheeseDecorator);
        gorgonzolaDecorator = new GorgonzolaDecorator(jalapenoDecorator);
        prosciuttoCrudoDecorator = new ProsciuttoCrudoDecorator(gorgonzolaDecorator);
        System.out.println(prosciuttoCrudoDecorator.getPizzaType());
    }
}
```



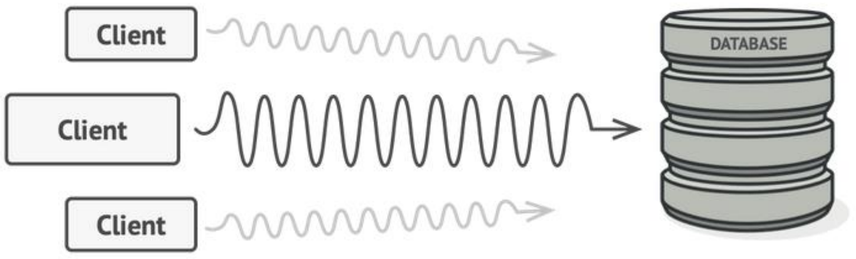
```
This is a basic pizza

This is a basic pizza
   with extra cheese

This is a basic pizza
   with jalapenos

This is a basic pizza
   with gorgonzola

This is a basic pizza
   with prosciutto crudo

This is a basic pizza
   with extra cheese
   with jalapenos
   with gorgonzola
   with prosciutto crudo

Process finished with exit code 0
```

• Proxy Design Pattern

The proxy is a pattern which lets developers provide substitutes/placeholders for objects. It has control over the original object and allows actions before/after getting to the original object.
It basically allows the creation of an intermediary that will hide the complexity and act as an interface for the object.

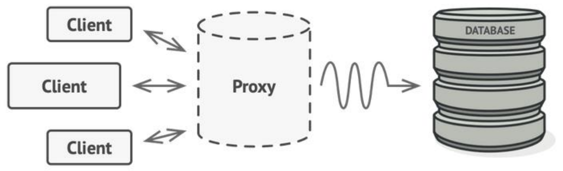


Example of problem solved by proxy:
We have a database in our application. As we know a database connection need a large number of resources, but we still need it so it's not like we have a choice.
What if we need to execute the application sometimes without needing to query anything in the database? What if we need to access the database just before we close the app to save some data? Why would we slow the app by connecting to the db every time or connecting when we start the app instead of when we need it.



Database queries can be really slow.

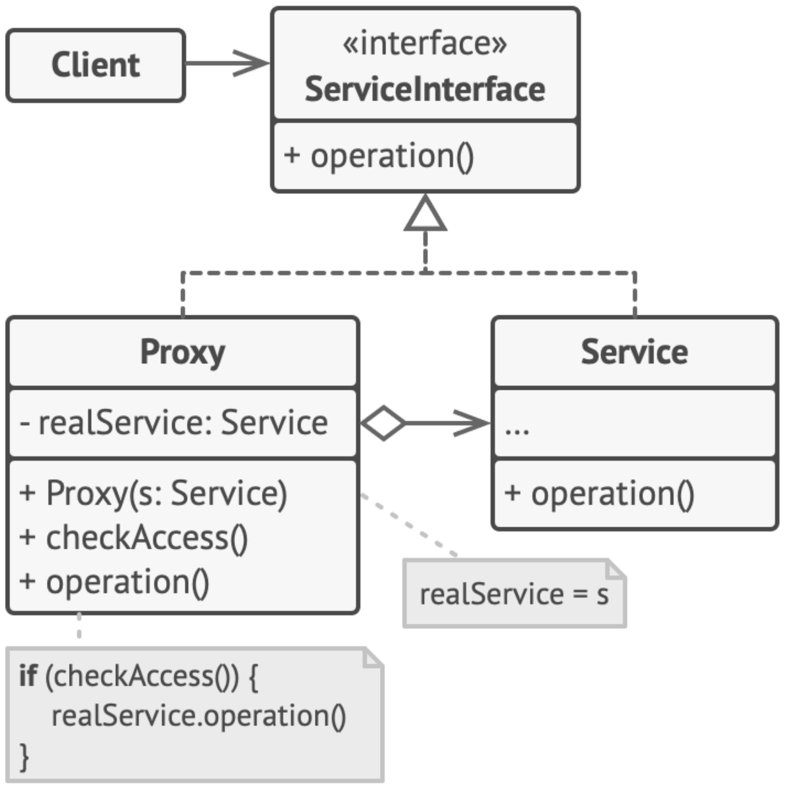
We only want this kind of object to be initialized when we want, and once they are initialized, reuse them for all our calls.



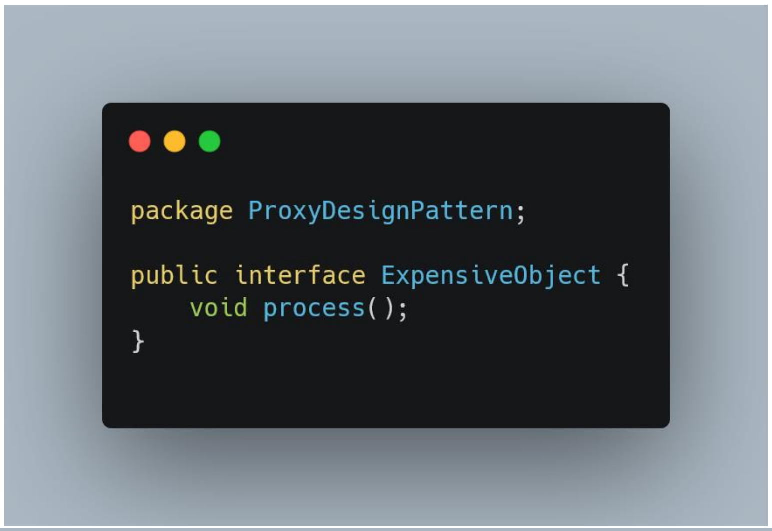
The main reasons for using proxy:

1. Need for a simplified version of a “heavy” resource/object – will load the og object on demand (lazy initialization instead of early) = **Virtual Proxy**
2. Need to represent locally an object at a different address space = **Remote Proxy**
3. Need to add security – to provide controlled access depending on the client’s permissions = **Protection Proxy**

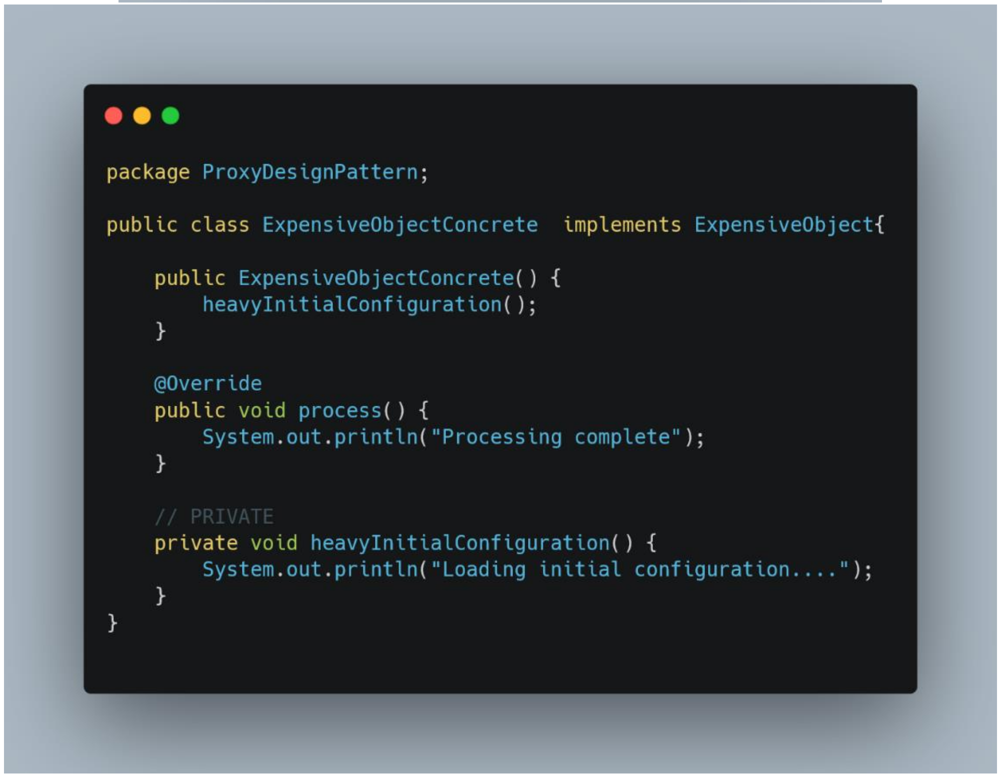
Implementation:
Basic Structure – Interface inherited by both the original object and the proxy object. The client calls the interface, and he doesn’t know that this will actually call the proxy instead of the original object.



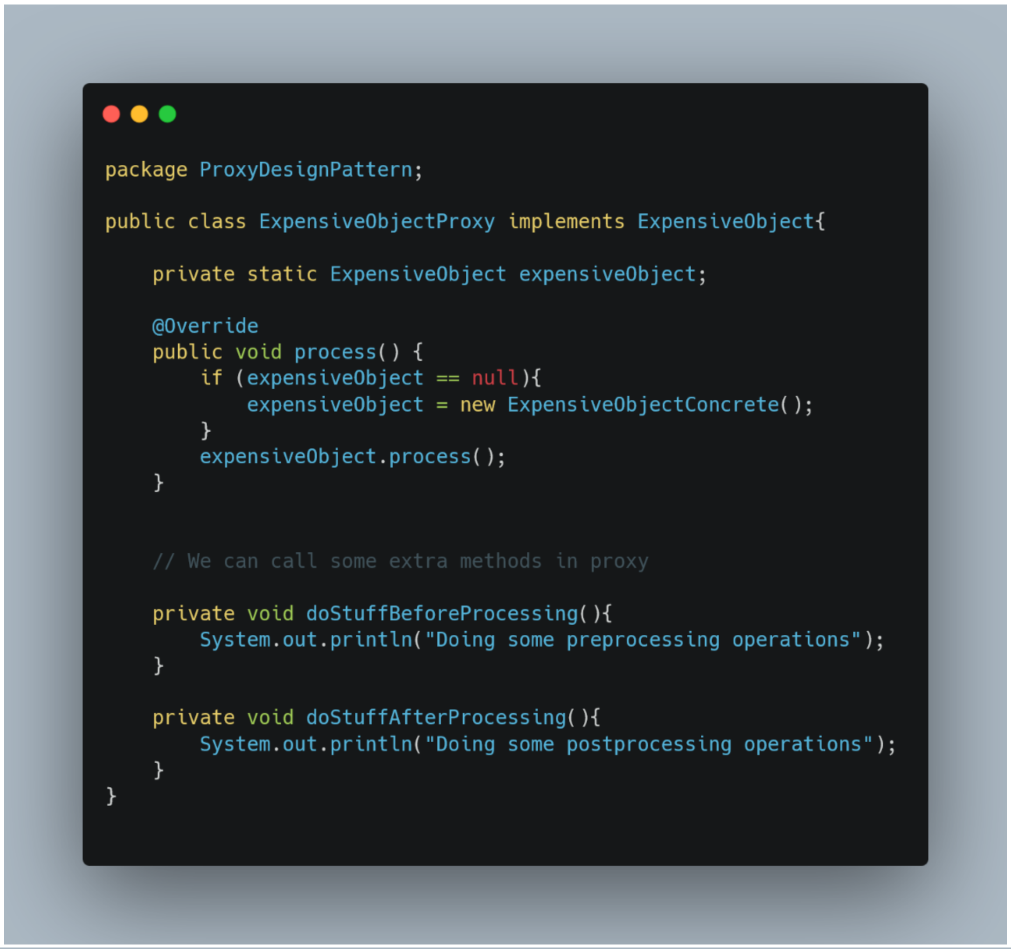
```
if (checkAccess()) {
   realService.operation()
}
```



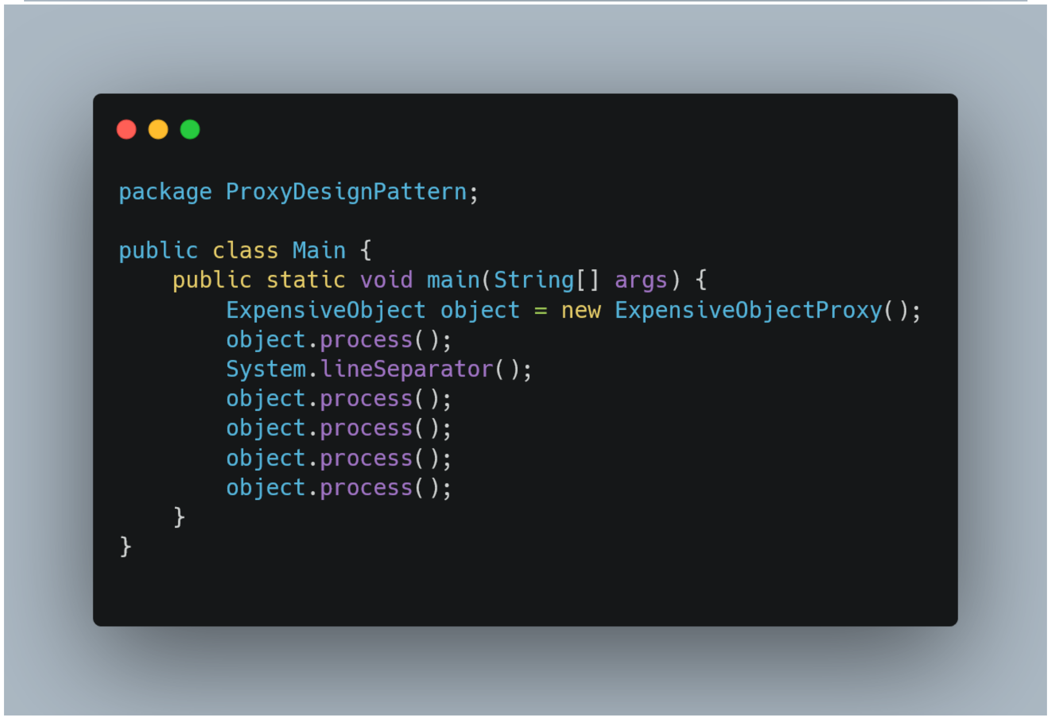
```java
package ProxyDesignPattern;

public interface ExpensiveObject {
    void process();
}
```



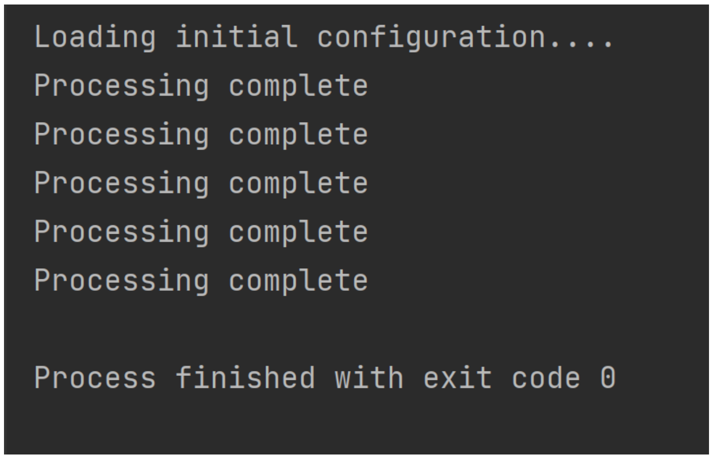
```java
package ProxyDesignPattern;

public class ExpensiveObjectConcrete  implements ExpensiveObject{

    public ExpensiveObjectConcrete() {
        heavyInitialConfiguration();
    }

    @Override
    public void process() {
        System.out.println("Processing complete");
    }

    // PRIVATE
    private void heavyInitialConfiguration() {
        System.out.println("Loading initial configuration....");
    }
}
```



```java
package ProxyDesignPattern;

public class ExpensiveObjectProxy implements ExpensiveObject{

    private static ExpensiveObject expensiveObject;

    @Override
    public void process() {
        if (expensiveObject == null){
            expensiveObject = new ExpensiveObjectConcrete();
        }
        expensiveObject.process();
    }


    // We can call some extra methods in proxy

    private void doStuffBeforeProcessing(){
        System.out.println("Doing some preprocessing operations");
    }

    private void doStuffAfterProcessing(){
        System.out.println("Doing some postprocessing operations");
    }
}
```



```java
package ProxyDesignPattern;

public class Main {
    public static void main(String[] args) {
        ExpensiveObject object = new ExpensiveObjectProxy();
        object.process();
        System.lineSeparator();
        object.process();
        object.process();
        object.process();
        object.process();
    }
}
```


```
Loading initial configuration....
Processing complete
Processing complete
Processing complete
Processing complete
Processing complete

Process finished with exit code 0
```

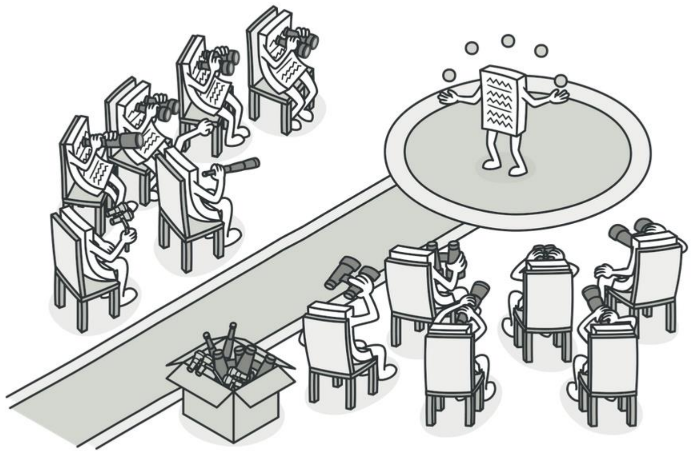
Behavioural Design Patterns

• Observer Design Pattern

Of all the behavioural design patterns, the Observer is the most used and best known.

It helps in defining a subscription mechanism in order to notify object about anything that happens to the observed object.

It specifies communication between entities: the **observable** and the **observers**. The observable is the object which notifies the others when it changes its state.

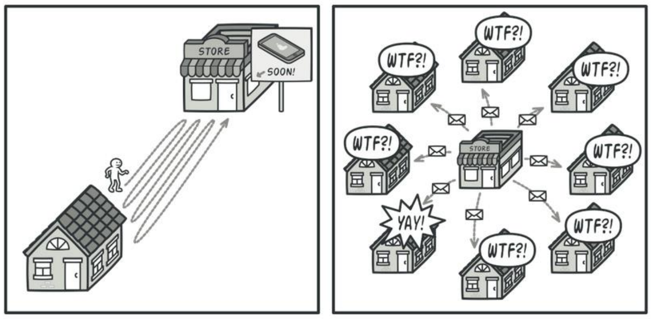


Problem which can be solved with Observer:

Two classes – Customer and Store. The customers are interested when to know when the new Iphone will become available.

One way – the customer goes every day to check the store.

Second way – the store will spam all the customers each time there's a new product available.

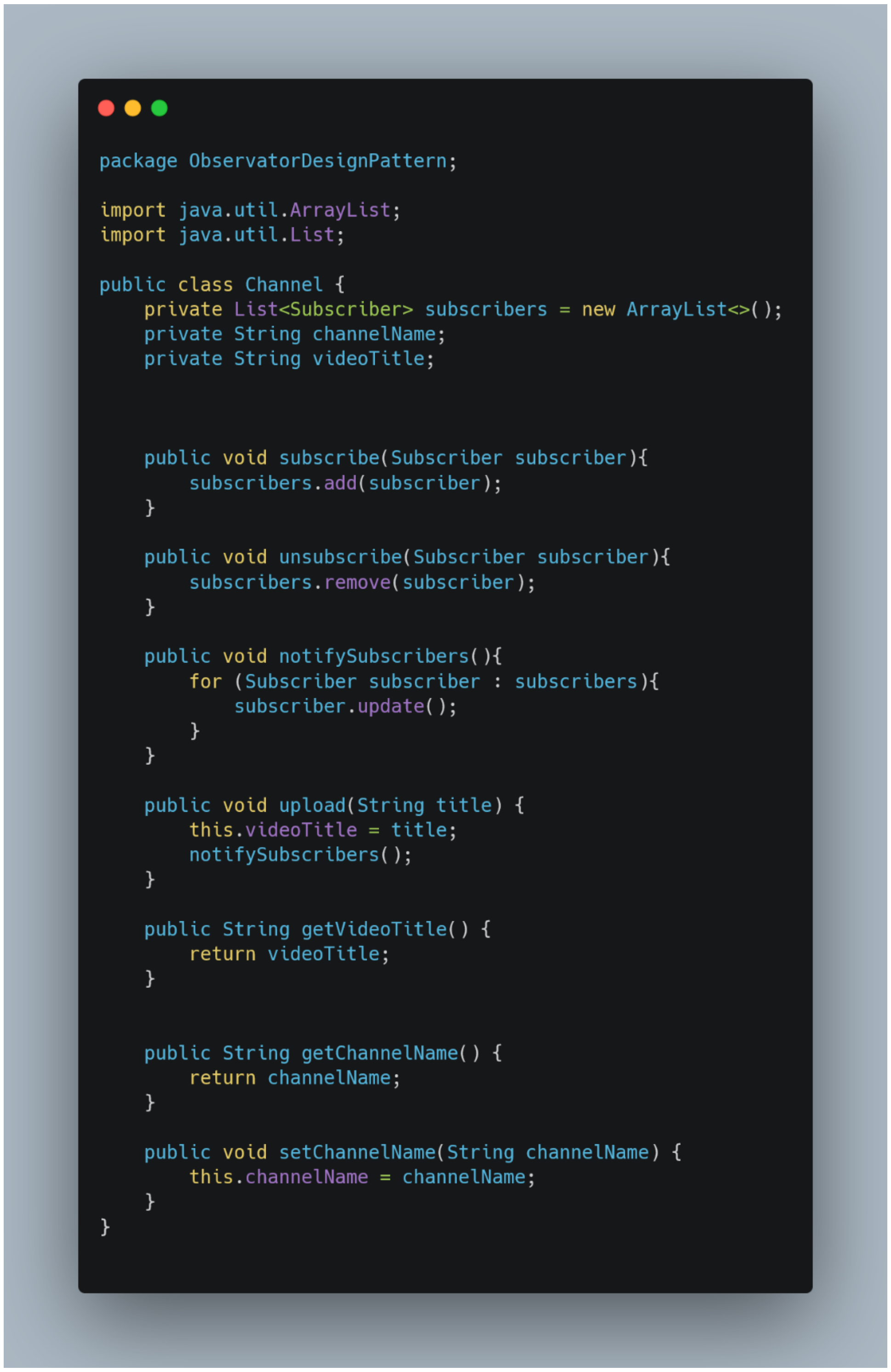


Visiting the store vs. sending spam

Solution:

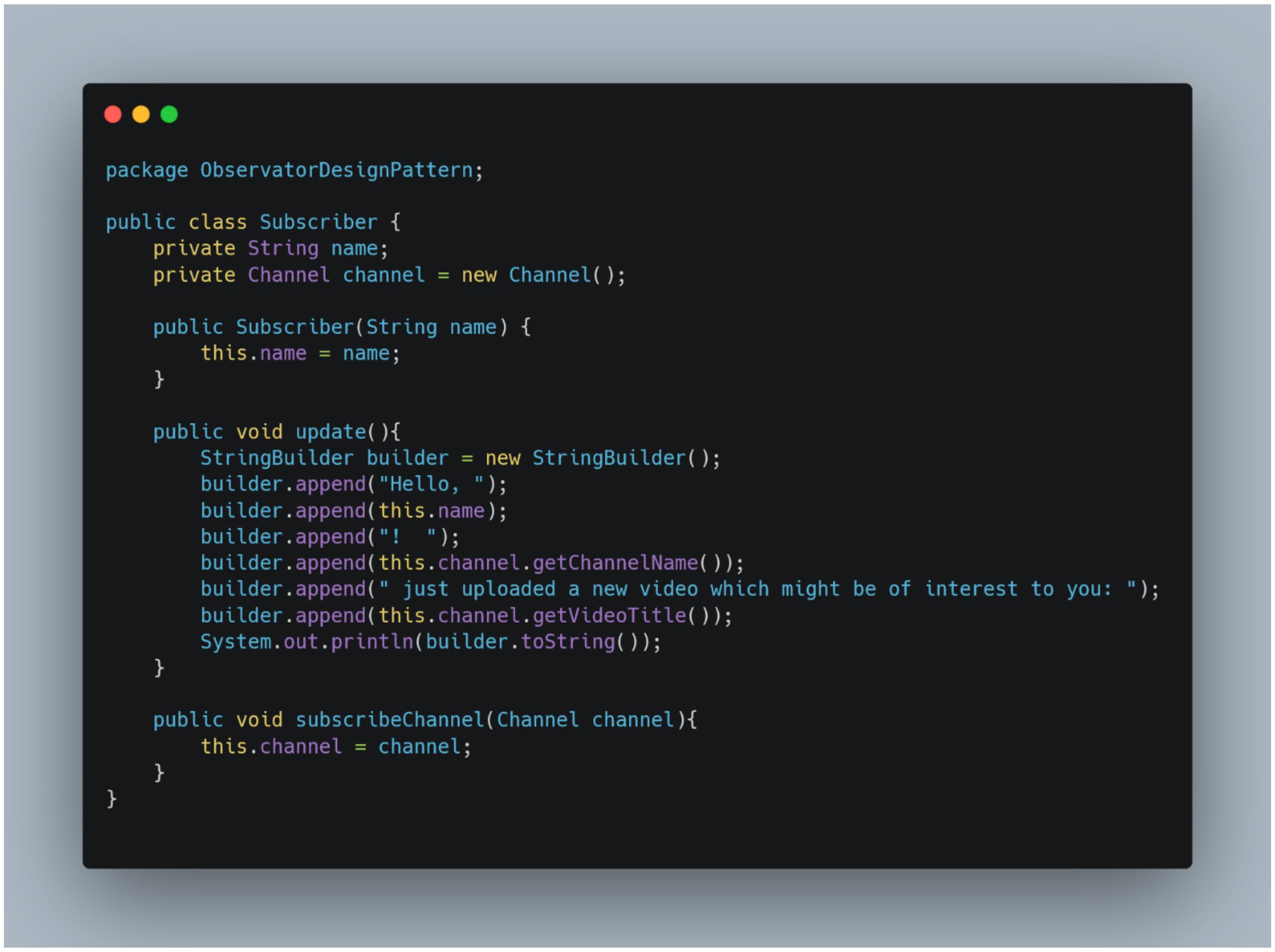
The Observer will have two classes: Store and Subscriber (the previous Customer). The store will keep an array field for storing a list of references to subscribers and some public methods including the ones which will allow the subscribers to subscribe/unsubscribe. Now the store can spam only the subscribers. Everyone is happy.

Implementation:

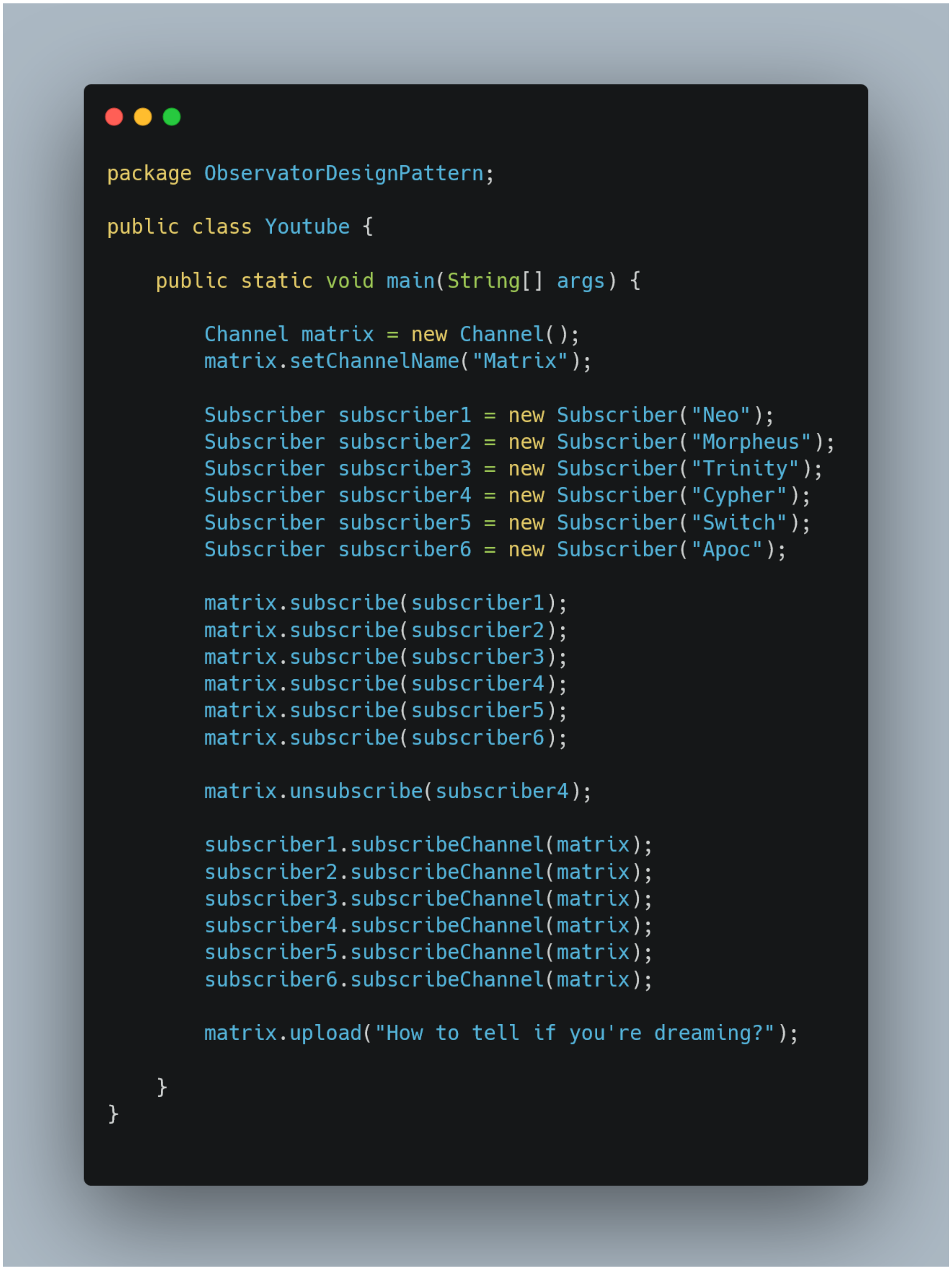


```java
package ObservatorDesignPattern;

import java.util.ArrayList;
import java.util.List;

public class Channel {
    private List<Subscriber> subscribers = new ArrayList<>();
    private String channelName;
    private String videoTitle;



    public void subscribe(Subscriber subscriber){
        subscribers.add(subscriber);
    }

    public void unsubscribe(Subscriber subscriber){
        subscribers.remove(subscriber);
    }

    public void notifySubscribers(){
        for (Subscriber subscriber : subscribers){
            subscriber.update();
        }
    }

    public void upload(String title) {
        this.videoTitle = title;
        notifySubscribers();
    }

    public String getVideoTitle() {
        return videoTitle;
    }


    public String getChannelName() {
        return channelName;
    }

    public void setChannelName(String channelName) {
        this.channelName = channelName;
    }
}
```

This will be equivalent to the Store class

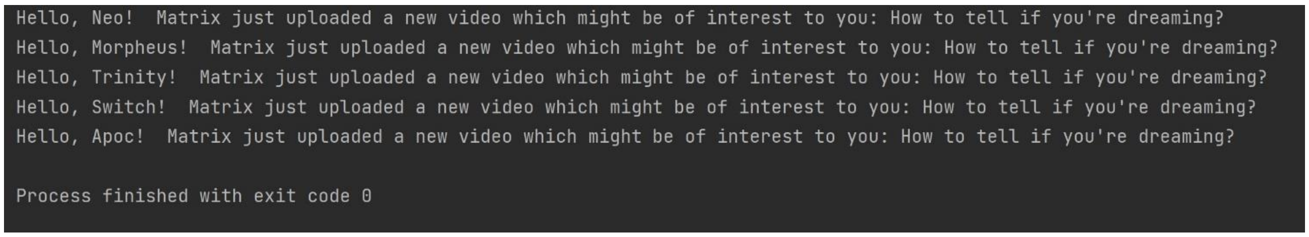


```java
package ObservatorDesignPattern;

public class Subscriber {
    private String name;
    private Channel channel = new Channel();

    public Subscriber(String name) {
        this.name = name;
    }

    public void update(){
        StringBuilder builder = new StringBuilder();
        builder.append("Hello, ");
        builder.append(this.name);
        builder.append("!  ");
        builder.append(this.channel.getChannelName());
        builder.append(" just uploaded a new video which might be of interest to you: ");
        builder.append(this.channel.getVideoTitle());
        System.out.println(builder.toString());
    }

    public void subscribeChannel(Channel channel){
        this.channel = channel;
    }
}
```



```java
package ObservatorDesignPattern;

public class Youtube {

    public static void main(String[] args) {

        Channel matrix = new Channel();
        matrix.setChannelName("Matrix");

        Subscriber subscriber1 = new Subscriber("Neo");
        Subscriber subscriber2 = new Subscriber("Morpheus");
        Subscriber subscriber3 = new Subscriber("Trinity");
        Subscriber subscriber4 = new Subscriber("Cypher");
        Subscriber subscriber5 = new Subscriber("Switch");
        Subscriber subscriber6 = new Subscriber("Apoc");

        matrix.subscribe(subscriber1);
        matrix.subscribe(subscriber2);
        matrix.subscribe(subscriber3);
        matrix.subscribe(subscriber4);
        matrix.subscribe(subscriber5);
        matrix.subscribe(subscriber6);

        matrix.unsubscribe(subscriber4);

        subscriber1.subscribeChannel(matrix);
        subscriber2.subscribeChannel(matrix);
        subscriber3.subscribeChannel(matrix);
        subscriber4.subscribeChannel(matrix);
        subscriber5.subscribeChannel(matrix);
        subscriber6.subscribeChannel(matrix);

        matrix.upload("How to tell if you're dreaming?");

    }
}
```

This is the main class



```
Hello, Neo!  Matrix just uploaded a new video which might be of interest to you: How to tell if you're dreaming?
Hello, Morpheus!  Matrix just uploaded a new video which might be of interest to you: How to tell if you're dreaming?
Hello, Trinity!  Matrix just uploaded a new video which might be of interest to you: How to tell if you're dreaming?
Hello, Switch!  Matrix just uploaded a new video which might be of interest to you: How to tell if you're dreaming?
Hello, Apoc!  Matrix just uploaded a new video which might be of interest to you: How to tell if you're dreaming?

Process finished with exit code 0
```

Conclusion

In conclusion, we've presented the most common and frequently used patterns, but what you should remember from all this, is the fact that design patterns are typical solutions to common problems. They mostly have benefits as long as they are used where they should be used as each of them is meant to solve a specific design flaw.

Bibliography

Anon., n.d. *Design Patterns.* [Online]
Available at: https://www.tutorialspoint.com/design_pattern

Anon., n.d. *Java Core Patterns.* [Online]
Available at: https://www.baeldung.com/java-core-patterns

Anon., n.d. *Java Design Patterns.* [Online]
Available at: https://www.javatpoint.com/java-design-patterns

Freeman, E. & Robson, E., 2020. *Head First Design Patterns.* 2nd ed. s.l.:O'Reilly Media, Inc..

Guru, R., n.d. *Design Patterns.* [Online]
Available at: https://refactoring.guru/design-patterns