Nesneye Yönelik Yazılım Mühendisliği (376)

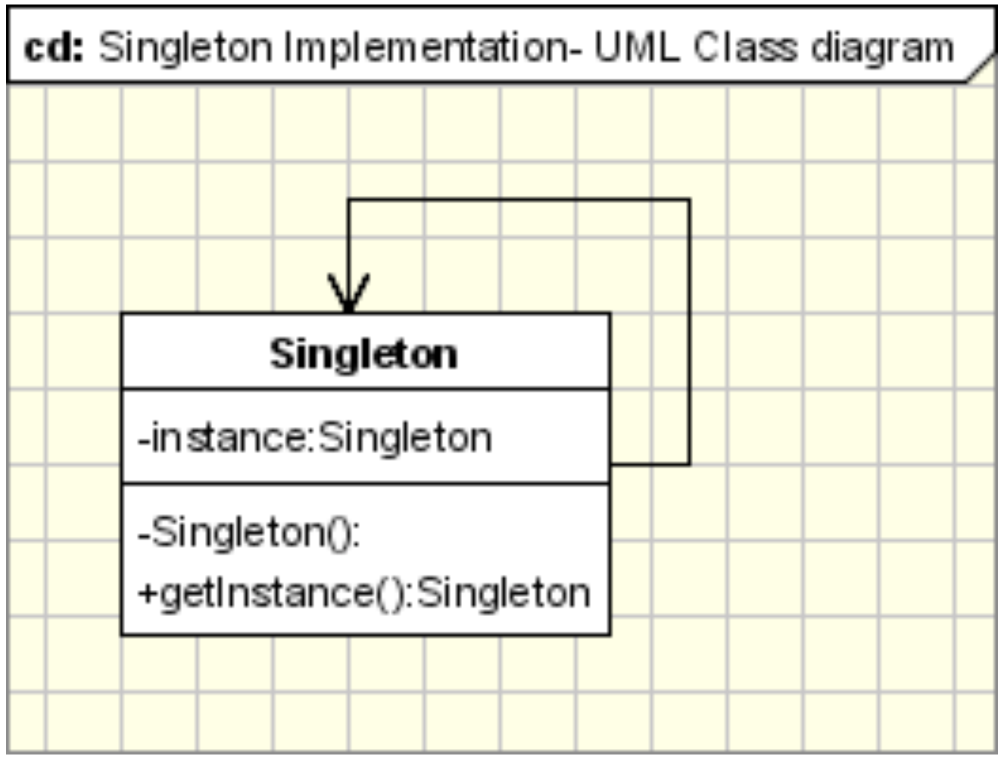
Dr. Öğr. Üyesi Ahmet Arif AYDIN

Singleton (Creational) Design Pattern

- ***Bir sınıfın sadece bir örneğinin oluşturulmasını*** ve uygulama boyunca kullanılmasını sağlar (*enables to create one object of a class*)
- Çok iş parçacıklı bir ortamda bile, bir sınıfın birden fazla örneğinin **oluşturulmamasını** sağlar (ensure **not more than one instance of a class** is ever instantiated, *even in a **multithreaded** environment)*
- *Double-Checked Locking*
    - Eşzamanlı olarak çalışan birden çok iş parçasının (multithreaded) kullanıldığı ortamlarda singleton kalıbının görevini yerine getirir.

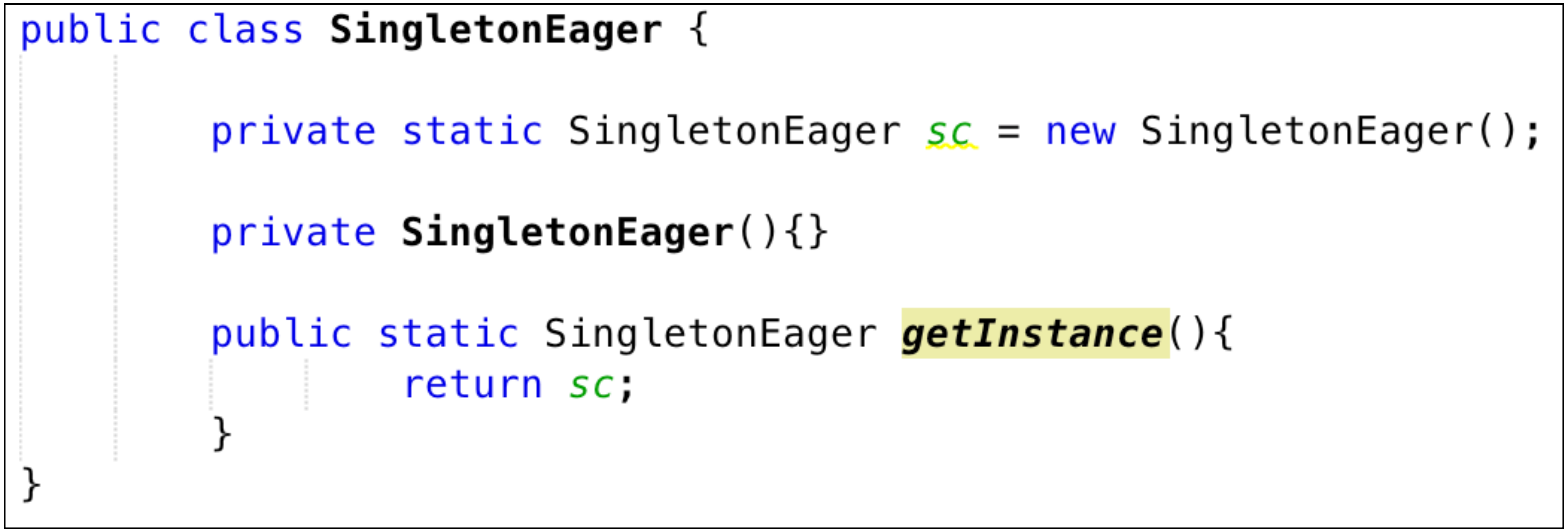
Singleton (Creational) Design Pattern

- Singleton Pattern aşağıdaki problemlerin çözümünde kullanılmaktadır:
  - Hatalı program davranışı (incorrect program behavior)
  - Kaynakların fazla kullanılması (overuse of resources)
  - Tutarsız sonuç (inconsistent results)



http://www.oodesign.com/

Singleton (Creational) Design Pattern

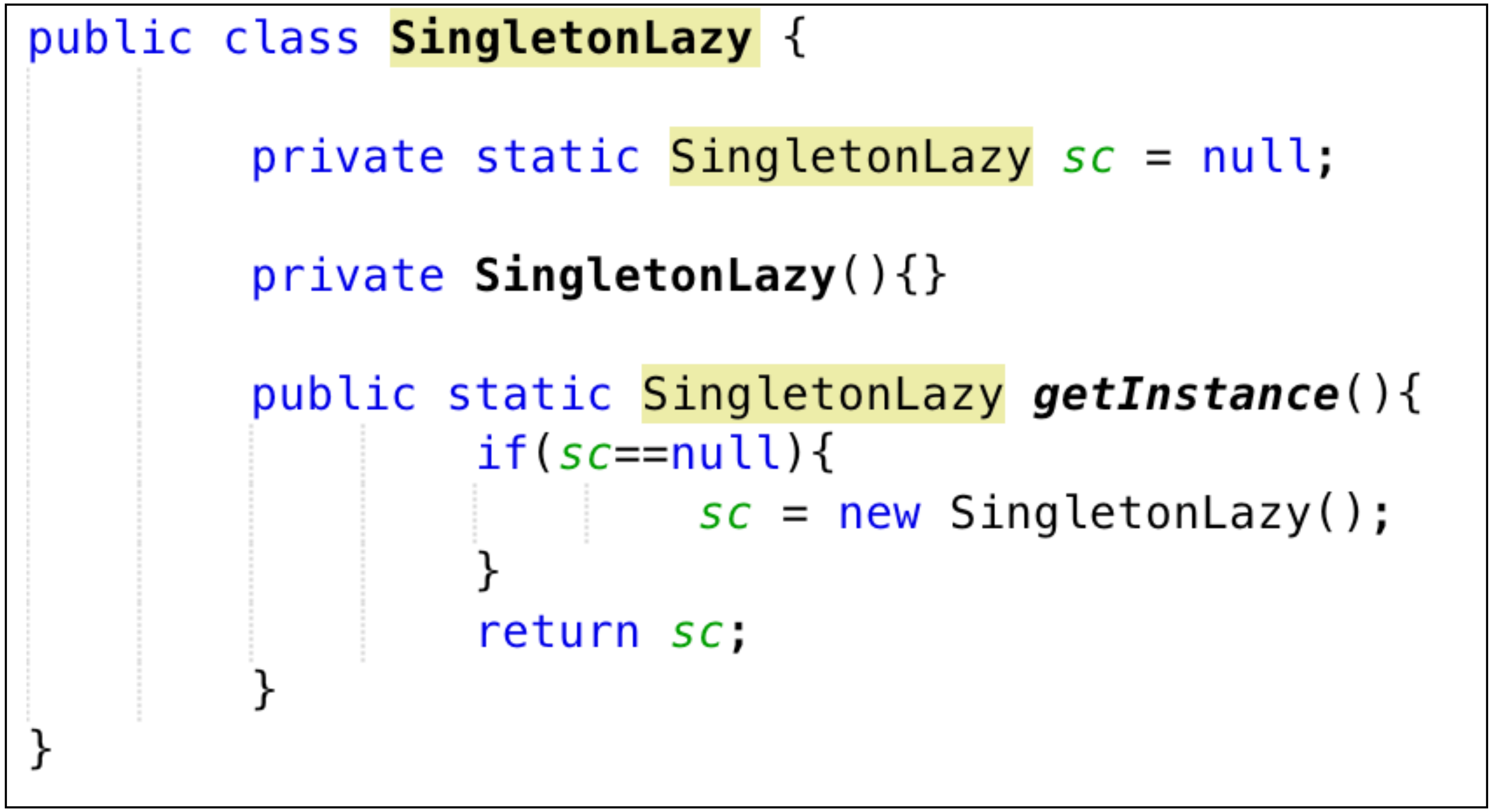


```
public class SingletonEager {

        private static SingletonEager sc = new SingletonEager();

        private SingletonEager(){}

        public static SingletonEager getInstance(){
                return sc;
        }
}
```

- SingletonEager makes sure that *only one object of the class gets created* and even if there are several requests, only *the same instantiated object will be returned*

- **Problem:** the *object would get created as soon as the class gets loaded into the JVM*. If the object is never requested, there would be an object useless inside the memory.

Singleton (Creational) Design Pattern



```java
public class SingletonLazy {

	private static SingletonLazy sc = null;

	private SingletonLazy(){}

	public static SingletonLazy getInstance(){
		if(sc==null){
			sc = new SingletonLazy();
		}
		return sc;
	}
}
```

Bir nesne gerektiği zaman oluşturulmalıdır.

(an object should get created when it is required)!

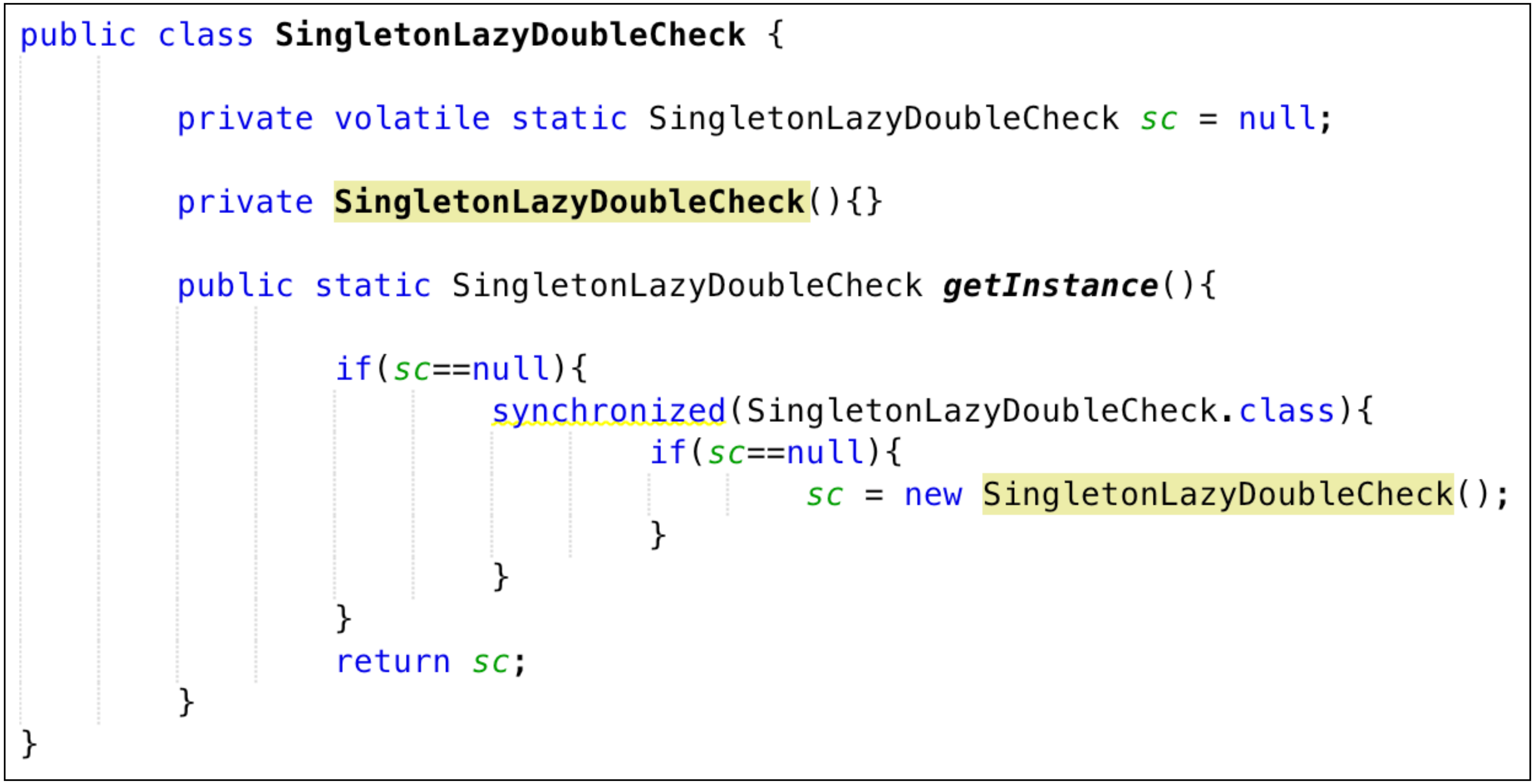
Yukarıda verilen kod multithreaded ortamlarda hata verir

Singleton (Creational) Design Pattern

```
public class SingletonLazyMultithreaded {

    private static SingletonLazyMultithreaded sc = null;

    private SingletonLazyMultithreaded(){}

    public static synchronized SingletonLazyMultithreaded getInstance(){
        if(sc==null){
            sc = new SingletonLazyMultithreaded();
        }
        return s we force
    }
}
```

❖ **synchronized**: *sadece bir thread işlem yapabilir* (every thread to wait its turn before it can enter the method. *no two threads will enter the method at the same time*)

Singleton (Creational) Design Pattern



```java
public class SingletonLazyDoubleCheck {

        private volatile static SingletonLazyDoubleCheck sc = null;

        private SingletonLazyDoubleCheck(){}

        public static SingletonLazyDoubleCheck getInstance(){

                if(sc==null){
                        synchronized(SingletonLazyDoubleCheck.class){
                                if(sc==null){
                                        sc = new SingletonLazyDoubleCheck();
                                }
                        }
                }
                return sc;
        }
}
```

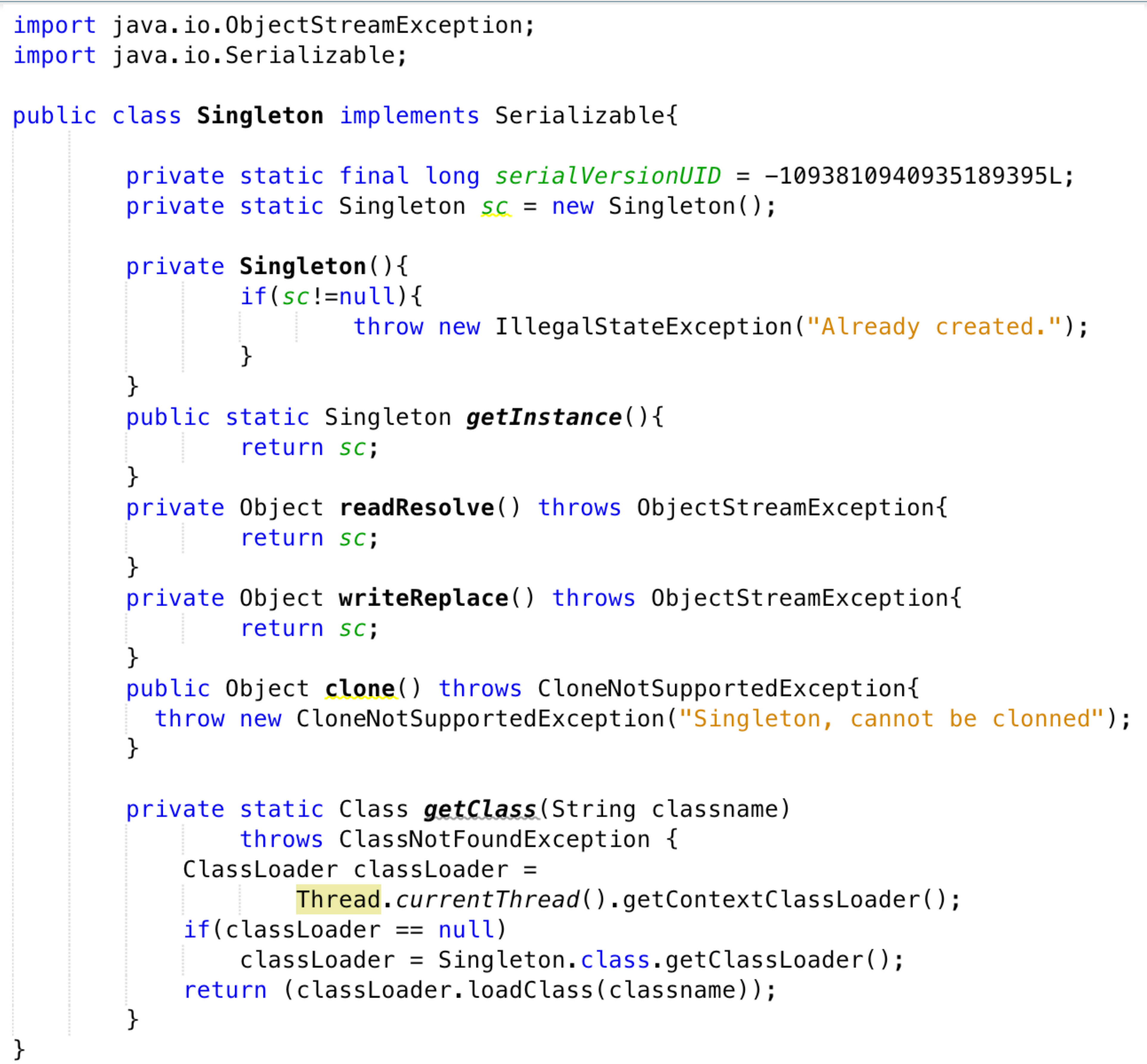
- volatile: the ***variable's value will be modified by different threads.***
- first check to see if an instance is created, and if not, then we synchronize. This way, we only synchronize the first time

Singleton (Creational) Design Pattern

- Singleton kalıbı ile gerçekleştirilmek istenileni aşağıdaki durumlar engelleyebilir
- If the class is Serializable.
- If it’s Clonable.
- It can be break by Reflection.
- if, the class is loaded by multiple class loaders.

https://www.oodlestechnologies.com/blogs/How-To-Save-Singleton-Pattern-from-Reflection-Serialization-and-Cloning

Singleton (Creational) Design Pattern

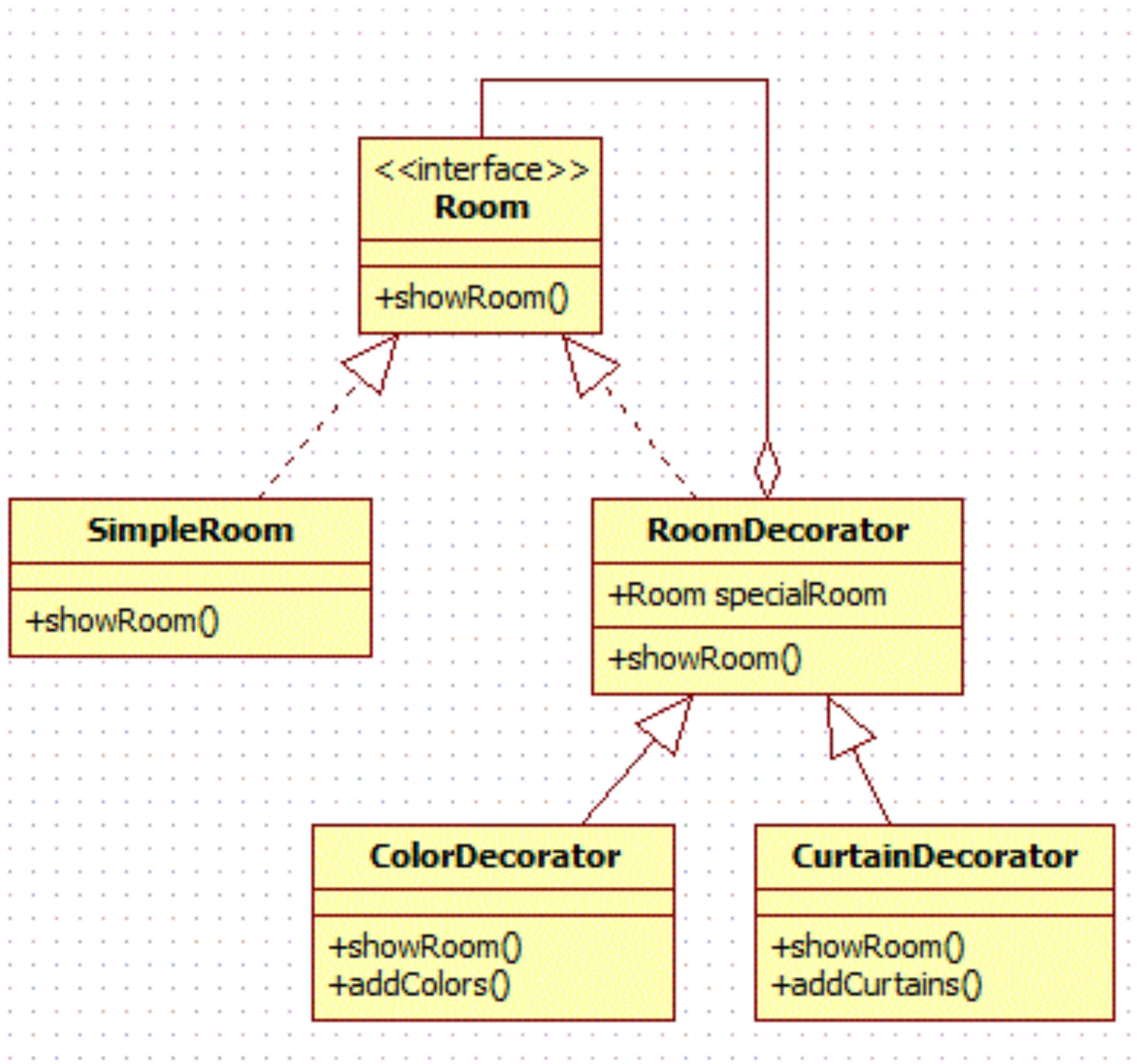


```java
import java.io.ObjectStreamException;
import java.io.Serializable;

public class Singleton implements Serializable{

        private static final long serialVersionUID = -1093810940935189395L;
        private static Singleton sc = new Singleton();

        private Singleton(){
                if(sc!=null){
                        throw new IllegalStateException("Already created.");
                }
        }
        public static Singleton getInstance(){
                return sc;
        }
        private Object readResolve() throws ObjectStreamException{
                return sc;
        }
        private Object writeReplace() throws ObjectStreamException{
                return sc;
        }
        public Object clone() throws CloneNotSupportedException{
          throw new CloneNotSupportedException("Singleton, cannot be clonned");
        }

        private static Class getClass(String classname)
                throws ClassNotFoundException {
            ClassLoader classLoader =
                    Thread.currentThread().getContextClassLoader();
            if(classLoader == null)
                classLoader = Singleton.class.getClassLoader();
            return (classLoader.loadClass(classname));
        }
}
```

Decorator (Structural) Design Pattern

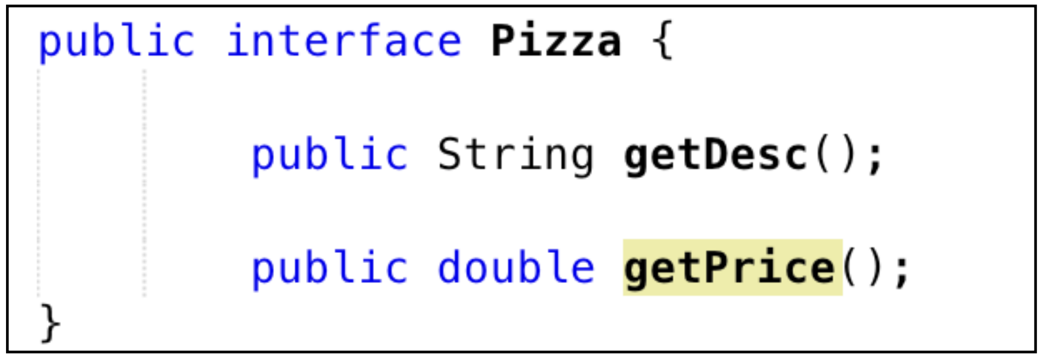
- Decorator pattern used to ***extend the functionality of an object dynamically*** without having to change the original class source or using inheritance.
- Bir nesneye dinamik olarak özellik ve sorumluluk eklemek için kullanılır.
- Dinamik özellik eklenirken sınıfın original yapısı değiştirilmez veya kalıtım kullanılır

Decorator (Structural) Design Pattern

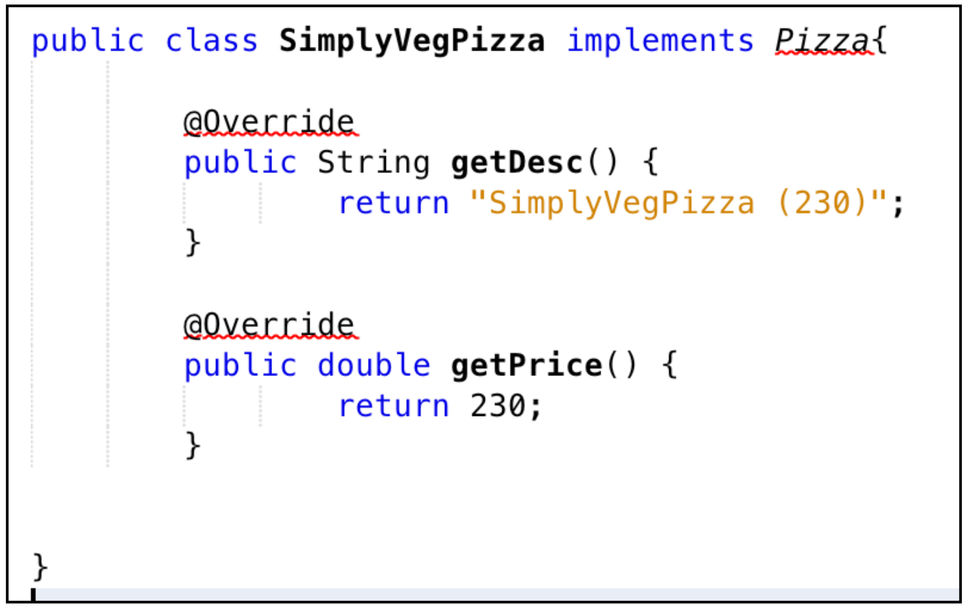


https://www.codeproject.com/Tips/468951/Decorator-Design-Pattern-in-Java

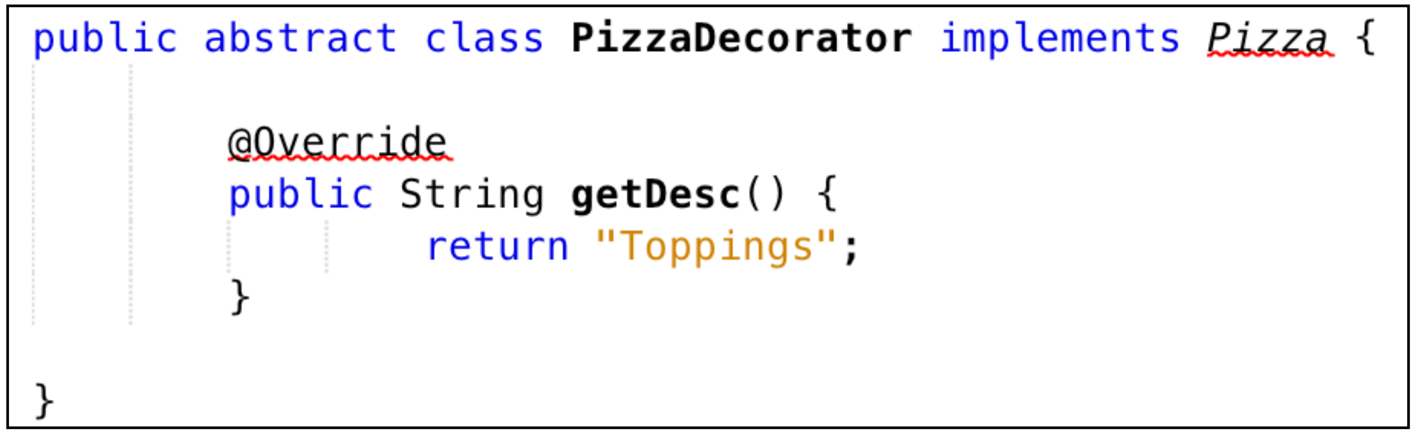
Decorator (Structural) Design Pattern



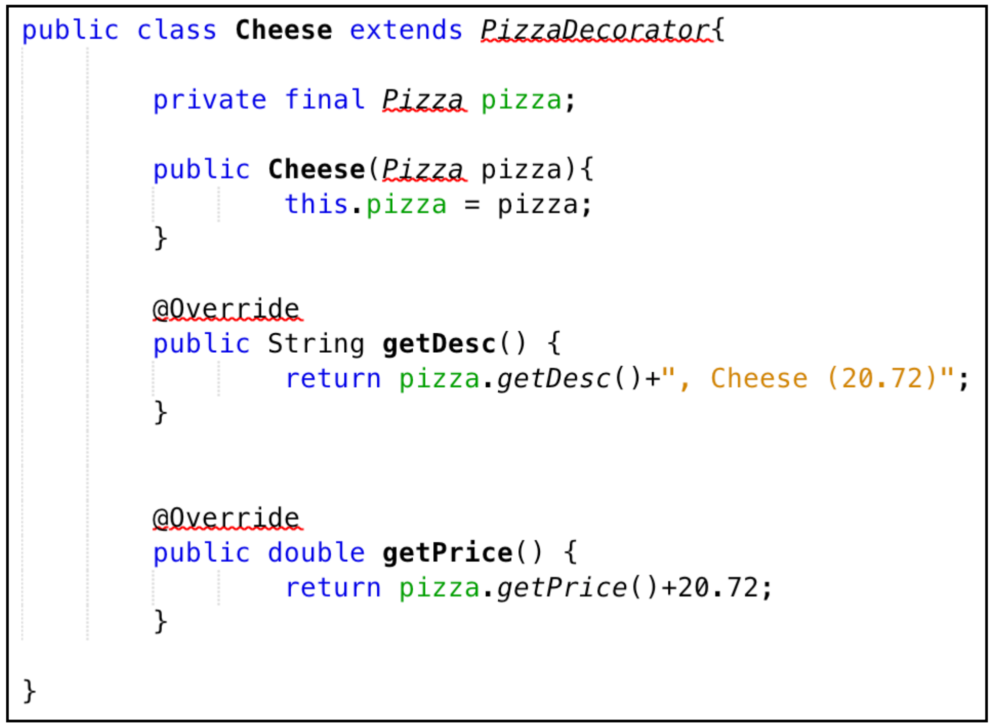
```java
public interface Pizza {

        public String getDesc();

        public double getPrice();
}
```



```java
public class SimplyVegPizza implements Pizza{

        @Override
        public String getDesc() {
                return "SimplyVegPizza (230)";
        }

        @Override
        public double getPrice() {
                return 230;
        }


}
```

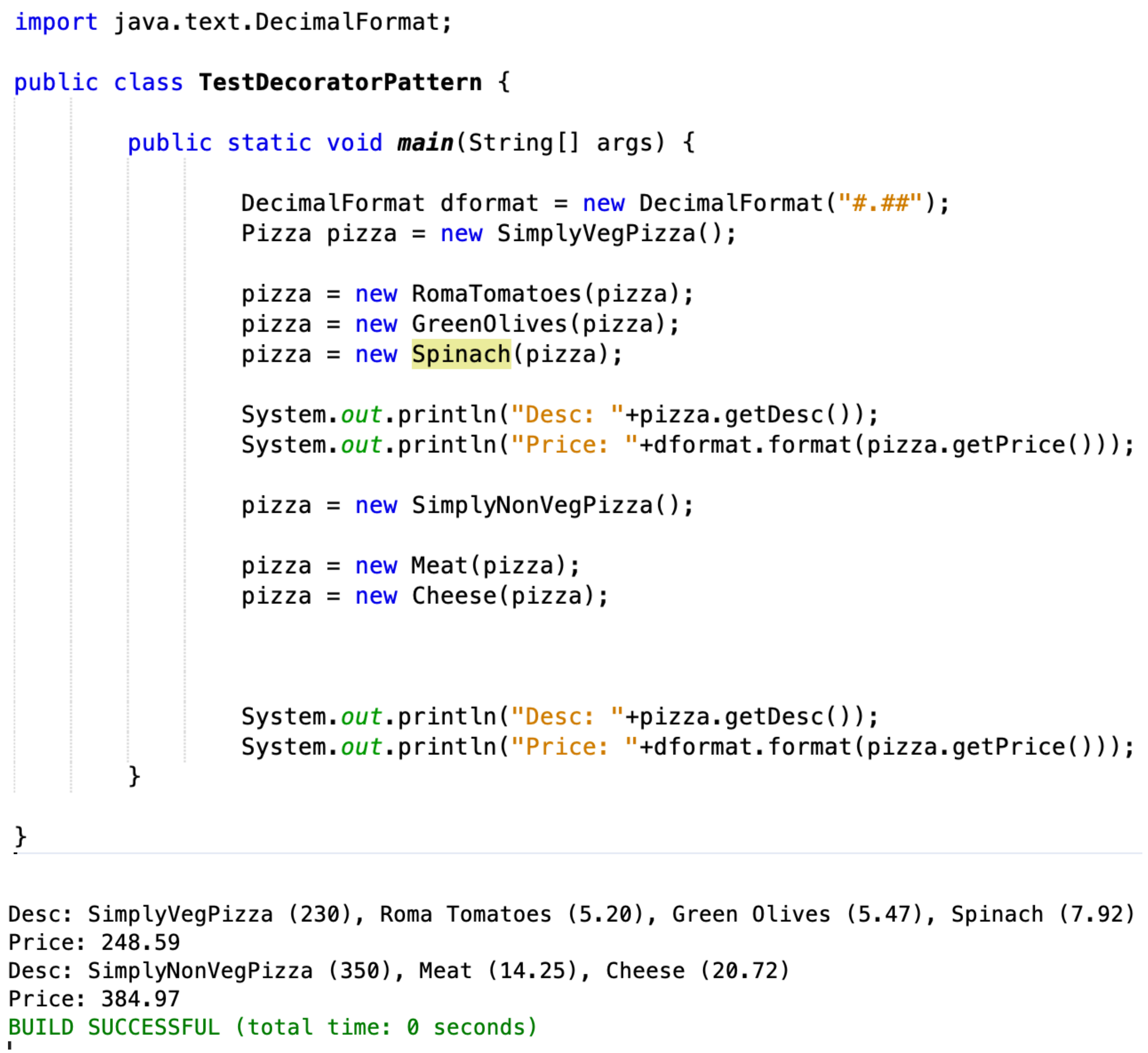


```java
public abstract class PizzaDecorator implements Pizza {

        @Override
        public String getDesc() {
                return "Toppings";
        }

}
```



```java
public class Cheese extends PizzaDecorator{

        private final Pizza pizza;

        public Cheese(Pizza pizza){
                this.pizza = pizza;
        }

        @Override
        public String getDesc() {
                return pizza.getDesc()+", Cheese (20.72)";
        }


        @Override
        public double getPrice() {
                return pizza.getPrice()+20.72;
        }

}
```

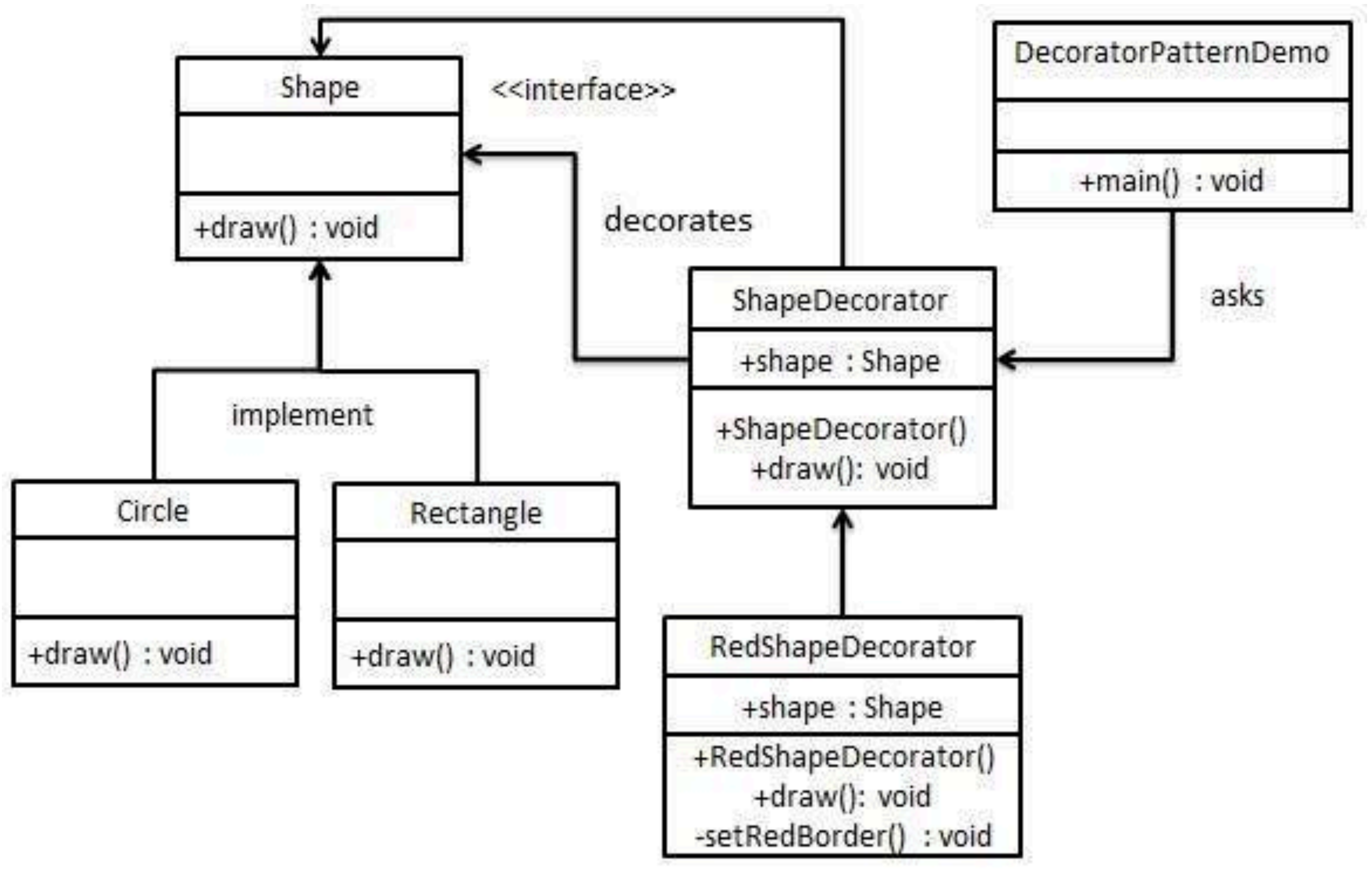
Decorator (Structural) Design Pattern



```java
import java.text.DecimalFormat;

public class TestDecoratorPattern {

	public static void main(String[] args) {

		DecimalFormat dformat = new DecimalFormat("#.##");
		Pizza pizza = new SimplyVegPizza();

		pizza = new RomaTomatoes(pizza);
		pizza = new GreenOlives(pizza);
		pizza = new Spinach(pizza);

		System.out.println("Desc: "+pizza.getDesc());
		System.out.println("Price: "+dformat.format(pizza.getPrice()));

		pizza = new SimplyNonVegPizza();

		pizza = new Meat(pizza);
		pizza = new Cheese(pizza);



		System.out.println("Desc: "+pizza.getDesc());
		System.out.println("Price: "+dformat.format(pizza.getPrice()));
	}

}
```

```
Desc: SimplyVegPizza (230), Roma Tomatoes (5.20), Green Olives (5.47), Spinach (7.92)
Price: 248.59
Desc: SimplyNonVegPizza (350), Meat (14.25), Cheese (20.72)
Price: 384.97
BUILD SUCCESSFUL (total time: 0 seconds)
```

Decorator (Structural) Design Pattern



https://www.tutorialspoint.com/design_pattern/decorator_pattern.htm

Decorator (Structural) Design Pattern

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

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Shape: Rectangle");
   }
}
```

```
public abstract class ShapeDecorator implements Shape {
   protected Shape decoratedShape;

   public ShapeDecorator(Shape decoratedShape){
      this.decoratedShape = decoratedShape;
   }

   public void draw(){
      decoratedShape.draw();
   }
}
```

```
public class RedShapeDecorator extends ShapeDecorator {

   public RedShapeDecorator(Shape decoratedShape) {
      super(decoratedShape);
   }

   @Override
   public void draw() {
      decoratedShape.draw();
      setRedBorder(decoratedShape);
   }

   private void setRedBorder(Shape decoratedShape){
      System.out.println("Border Color: Red");
   }
}
```

https://www.tutorialspoint.com/design_pattern/decorator_pattern.htm

Decorator (Structural) Design Pattern

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

      Shape circle = new Circle();

      Shape redCircle = new RedShapeDecorator(new Circle());

      Shape redRectangle = new RedShapeDecorator(new Rectangle());

     System.out.println("Circle with normal border");
      circle.draw();

      System.out.println("\nCircle of red border");
      redCircle.draw();

      System.out.println("\nRectangle of red border");
      redRectangle.draw();
   }
}
```

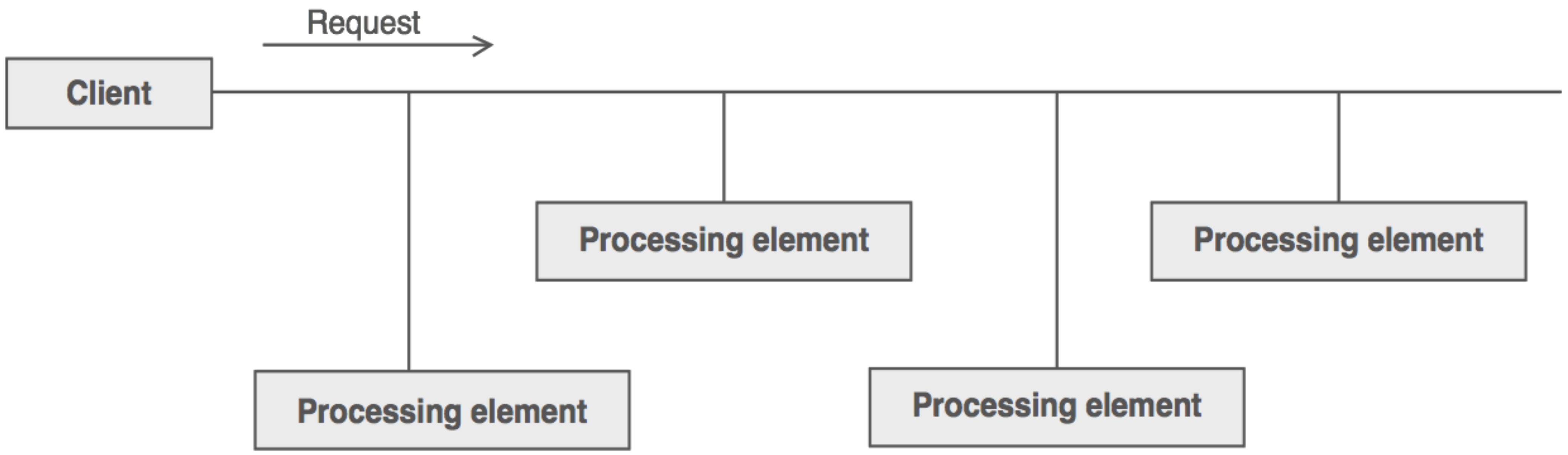
```
Circle with normal border
Shape: Circle

Circle of red border
Shape: Circle
Border Color: Red

Rectangle of red border
Shape: Rectangle
Border Color: Red
```

Decorator (Structural) Design Pattern

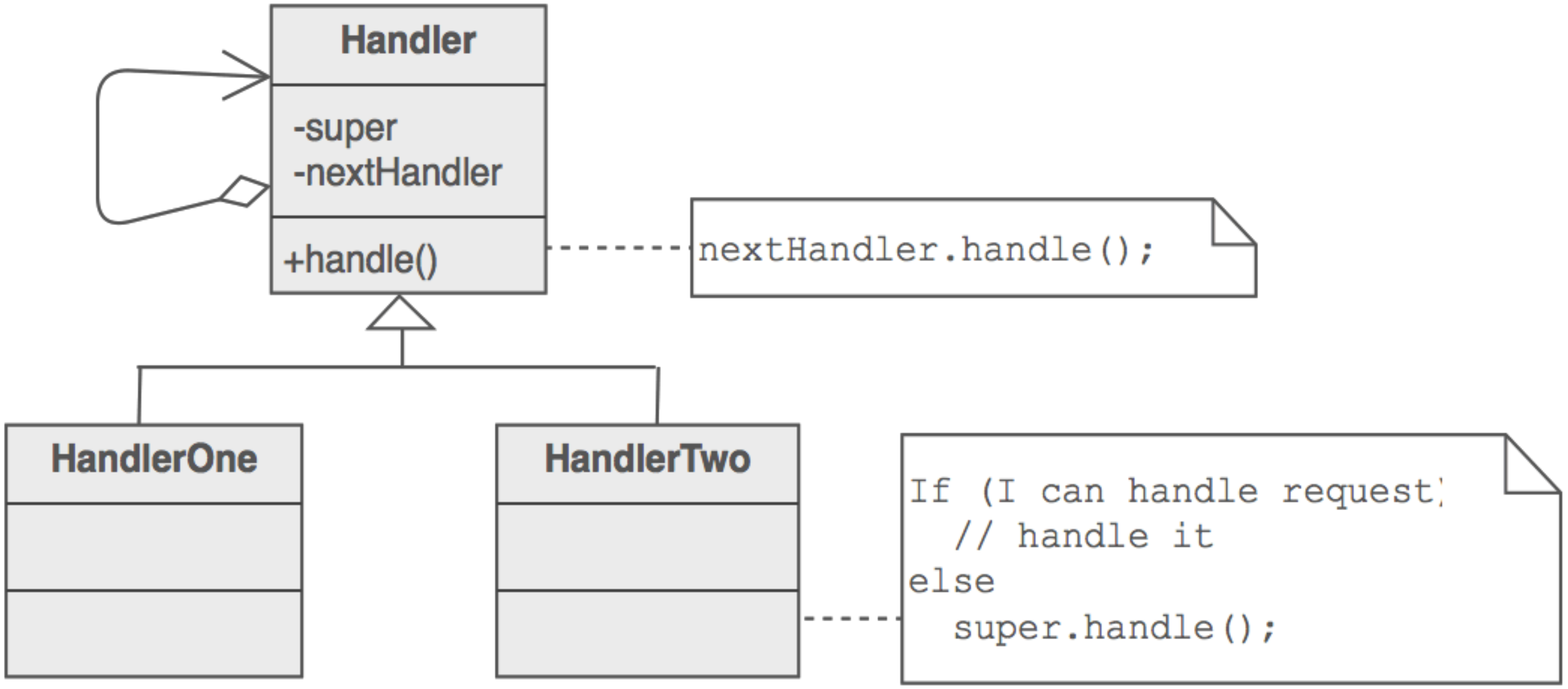
- java.io.BufferedInputStream(InputStream)
- java.io.DataInputStream(InputStream)
- java.io.BufferedOutputStream(OutputStream)
- java.util.zip.ZipOutputStream(OutputStream)
- java.util.Collections#checked[List|Map|Set|SortedSet|SortedMap]()

Chain of Responsibility (Behavioral) Design Pattern

- Bir problemin çözümünde birden fazla nesnenin işlem hazır bir biçimde beklemesi ile oluşur
    - When a request comes to a single object, it will check whether it can process and handle the specific file format. If it can, it will process it; otherwise, it will forward it to the next object chained to it



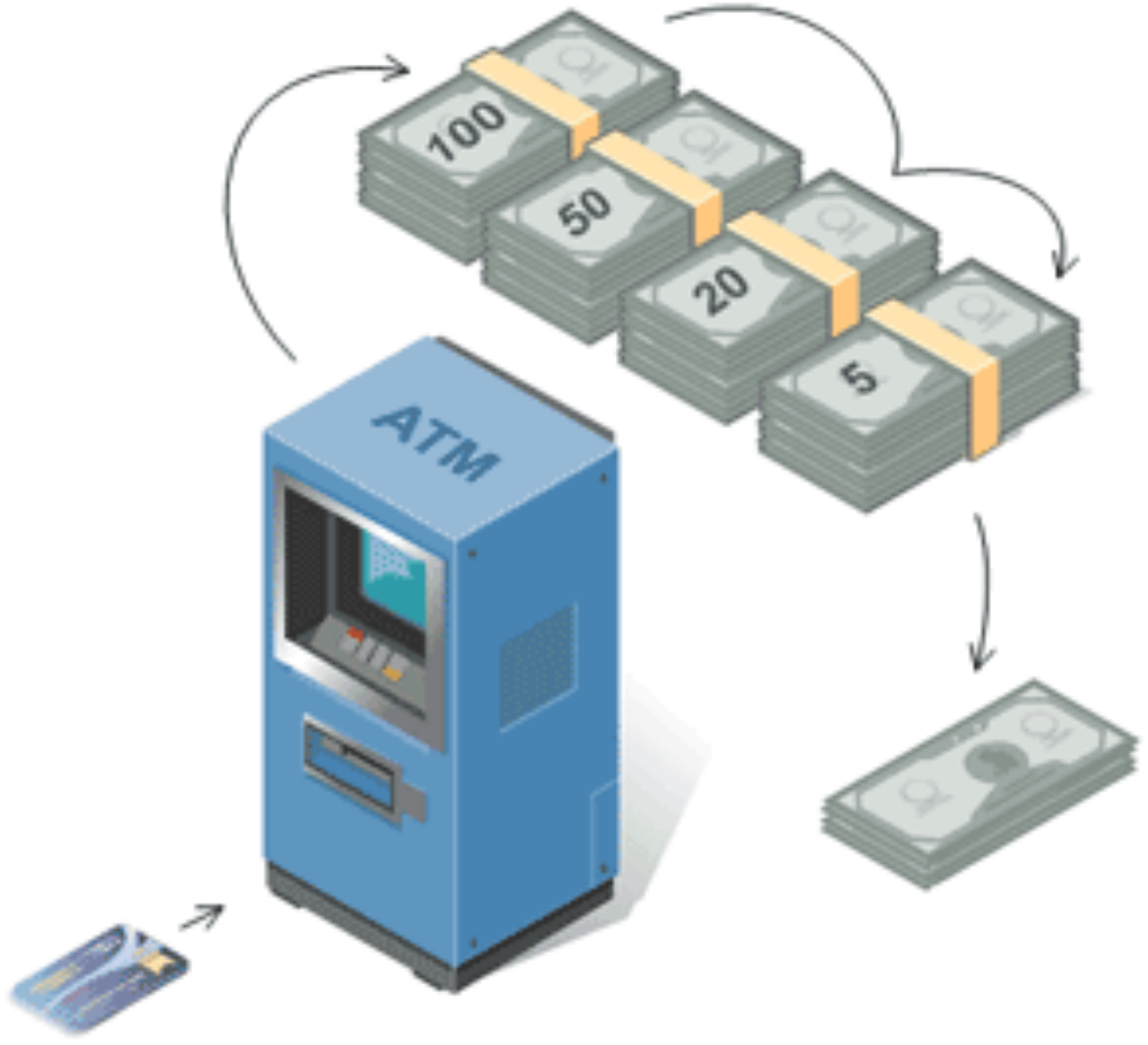
Chain of Responsibility (Behavioral) Design Pattern



```
nextHandler.handle();
```

```
If (I can handle request)
  // handle it
else
  super.handle();
```

Chain of Responsibility (Behavioral) Design Pattern



https://sourcemaking.com/design_patterns/chain_of_responsibility

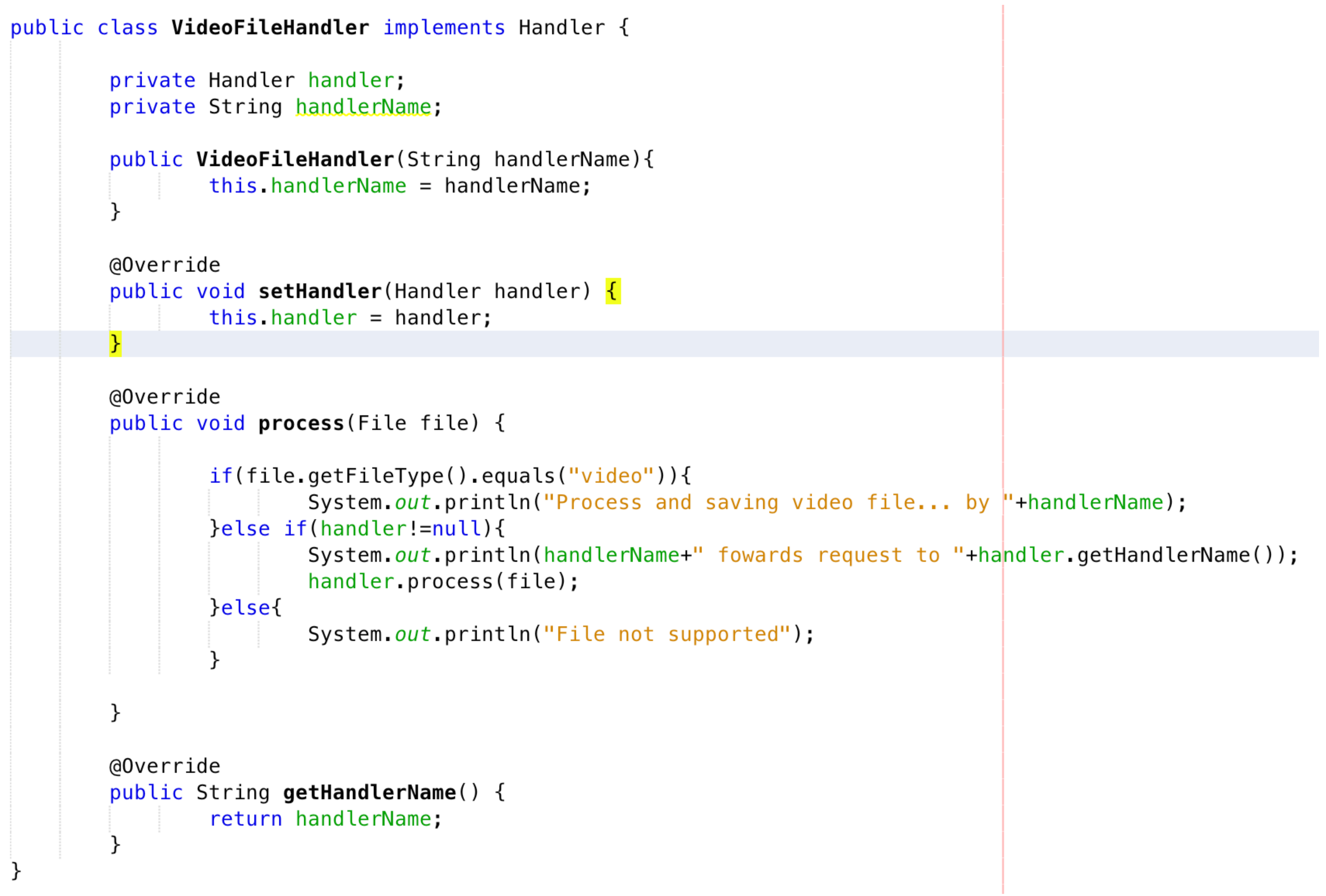
Chain of Responsibility (Behavioral) Design Pattern

```
public interface Handler {

        public void setHandler(Handler handler);
        public void process(File file);
        public String getHandlerName();
}
```

Chain of Responsibility (Behavioral) Design Pattern

```java
public class File {

        private final String fileName;
        private final String fileType;
        private final String filePath;

        public File(String fileName, String fileType, String filePath){
                this.fileName = fileName;
                this.fileType = fileType;
                this.filePath = filePath;
        }

        public String getFileName() {
                return fileName;
        }

        public String getFileType() {
                return fileType;
        }

        public String getFilePath() {
                return filePath;
        }


}
```

Chain of Responsibility (Behavioral) Design Pattern



```java
public class VideoFileHandler implements Handler {

        private Handler handler;
        private String handlerName;

        public VideoFileHandler(String handlerName){
                this.handlerName = handlerName;
        }

        @Override
        public void setHandler(Handler handler) {
                this.handler = handler;
        }

        @Override
        public void process(File file) {

                if(file.getFileType().equals("video")){
                        System.out.println("Process and saving video file... by "+handlerName);
                }else if(handler!=null){
                        System.out.println(handlerName+" fowards request to "+handler.getHandlerName());
                        handler.process(file);
                }else{
                        System.out.println("File not supported");
                }

        }

        @Override
        public String getHandlerName() {
                return handlerName;
        }
}
```