Nesneye Yönelik Yazılım Mühendisliği (376)

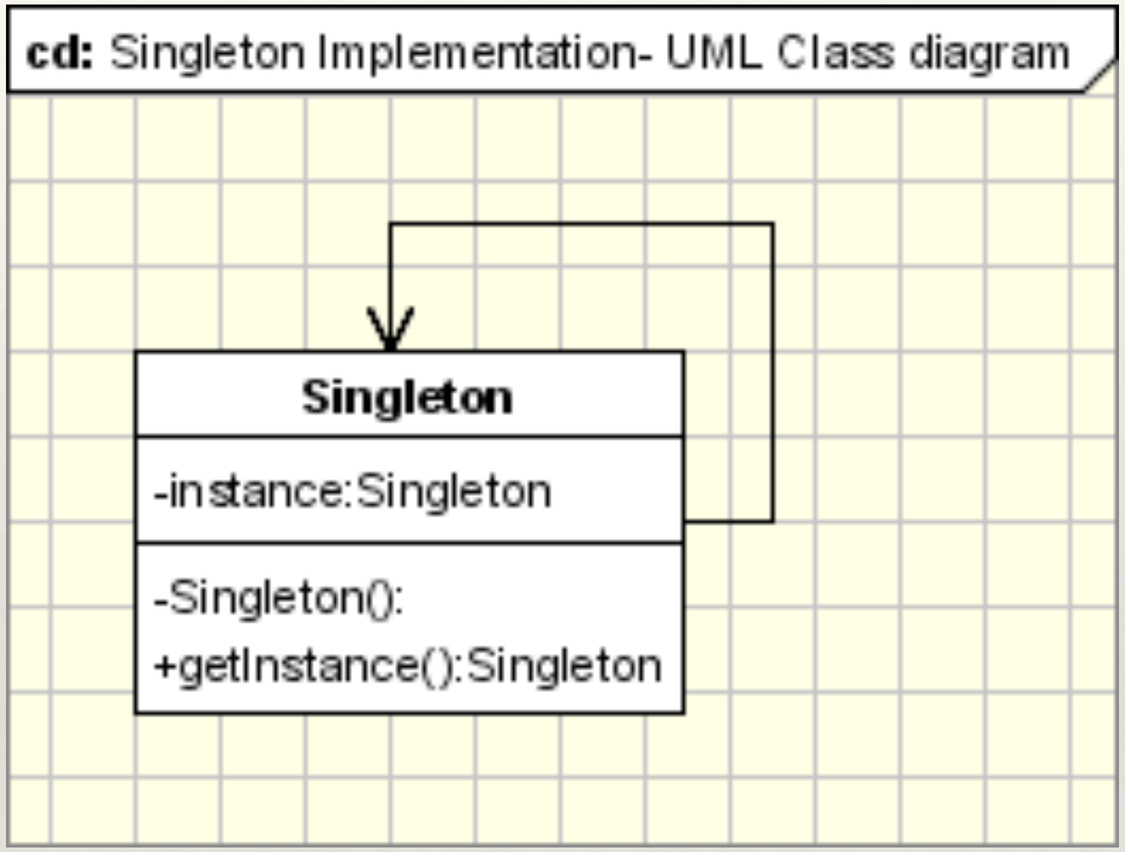
Yrd. Doç. Dr. Ahmet Arif AYDIN

Creational Design Patterns : Singleton

- Singleton Patterns
  - ***Bir sınıfın sadece bir örneğinin oluşturulmasını*** ve uygulama boyunca kullanılmasını sağlar (*enables to create one object of a class*)
  - 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.

Creational Design Patterns : Singleton

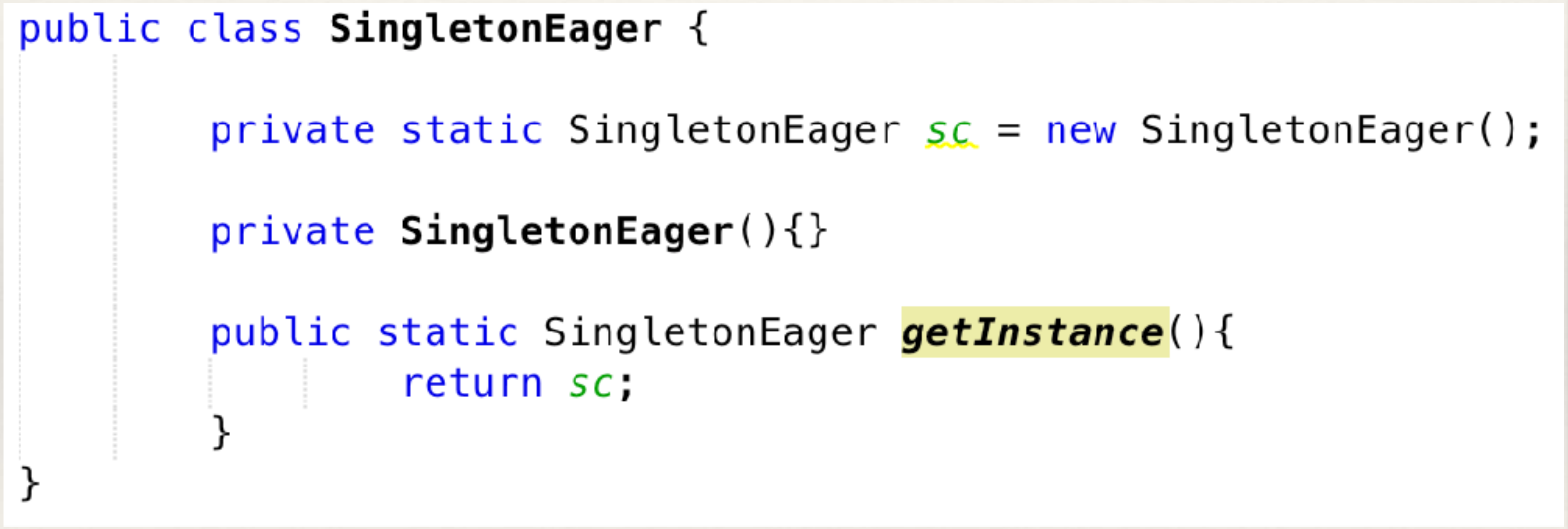
- Singleton Pattern neden kullanılmaktadır ?
  - incorrect program behavior
  - overuse of resources
  - inconsistent results



http://www.oodesign.com/

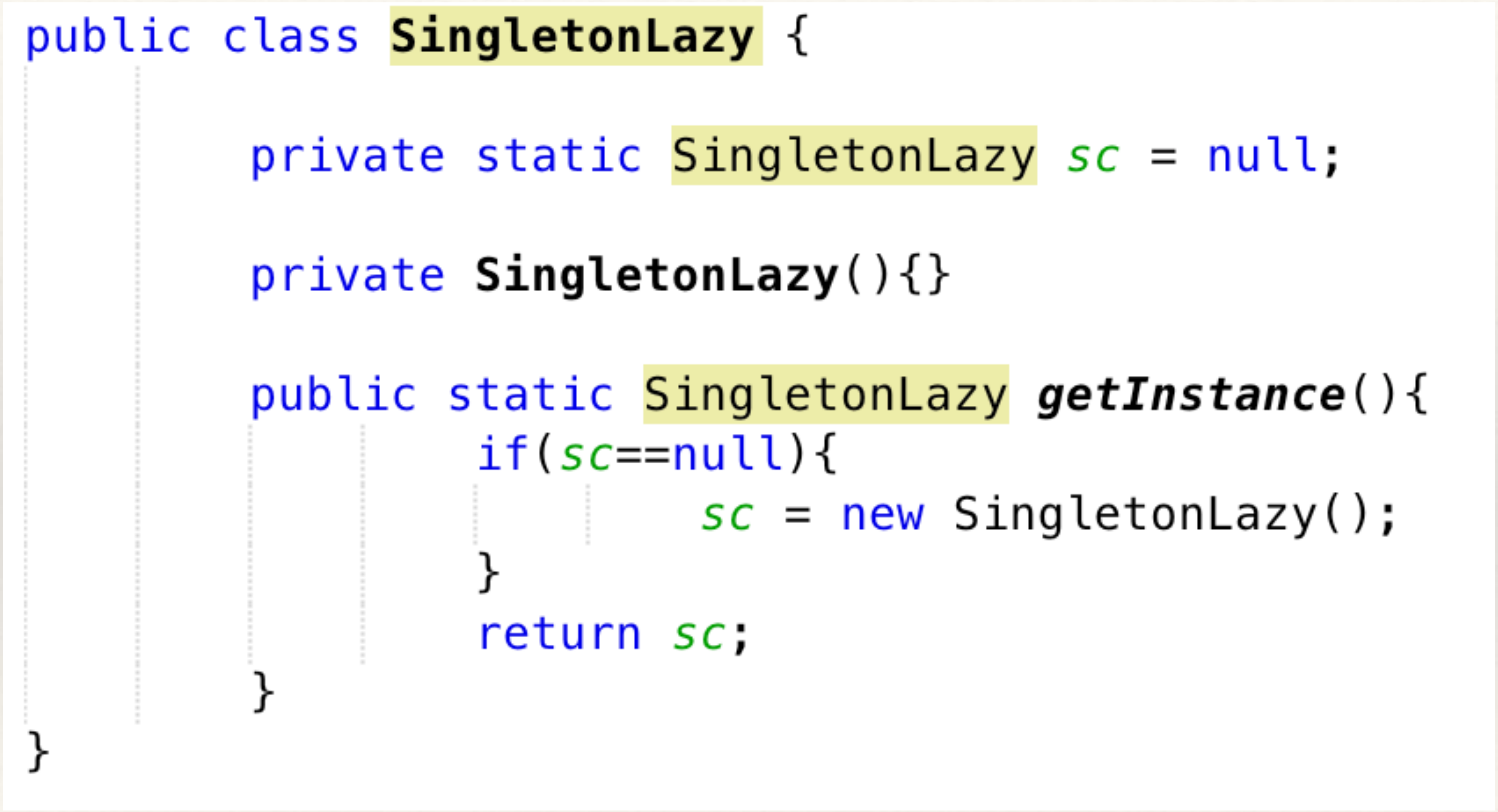
Creational Design Patterns : Singleton

- 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



```java
public class SingletonEager {

	private static SingletonEager sc = new SingletonEager();

	private SingletonEager(){}

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

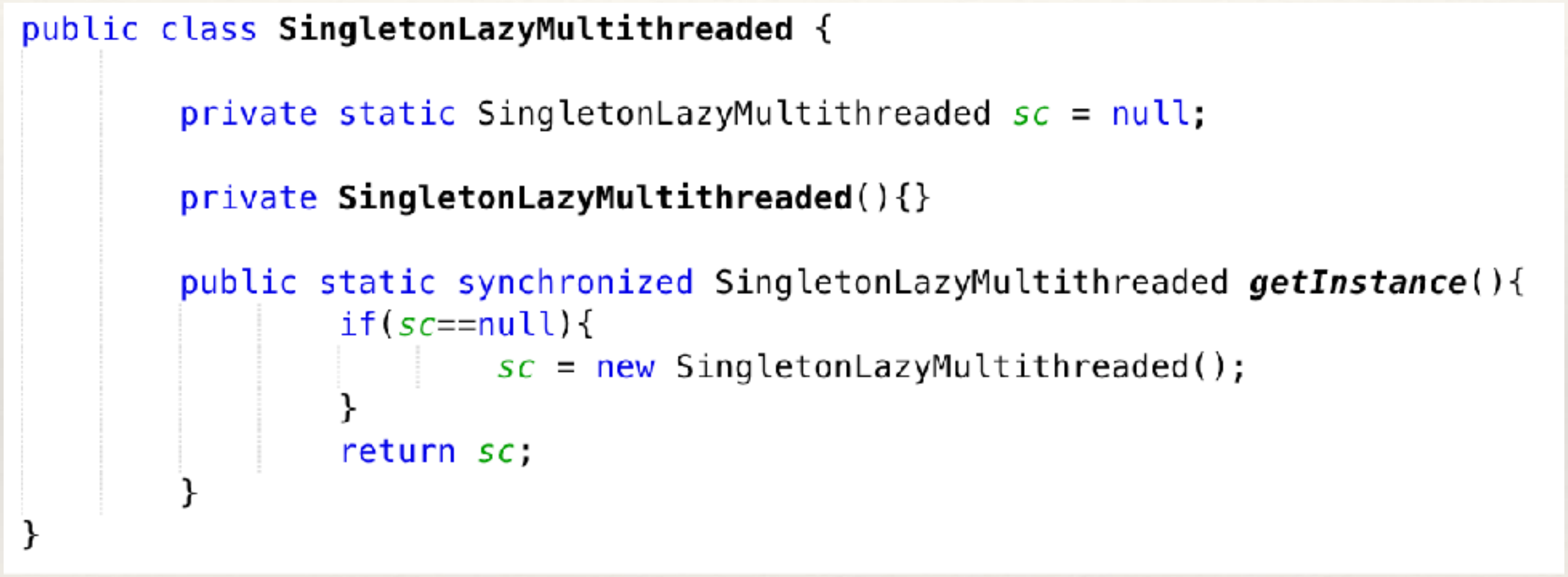
Creational Design Patterns : Singleton



```
public class SingletonLazy {

        private static SingletonLazy sc = null;

        private SingletonLazy(){}

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

Creational Design Patterns : Singleton

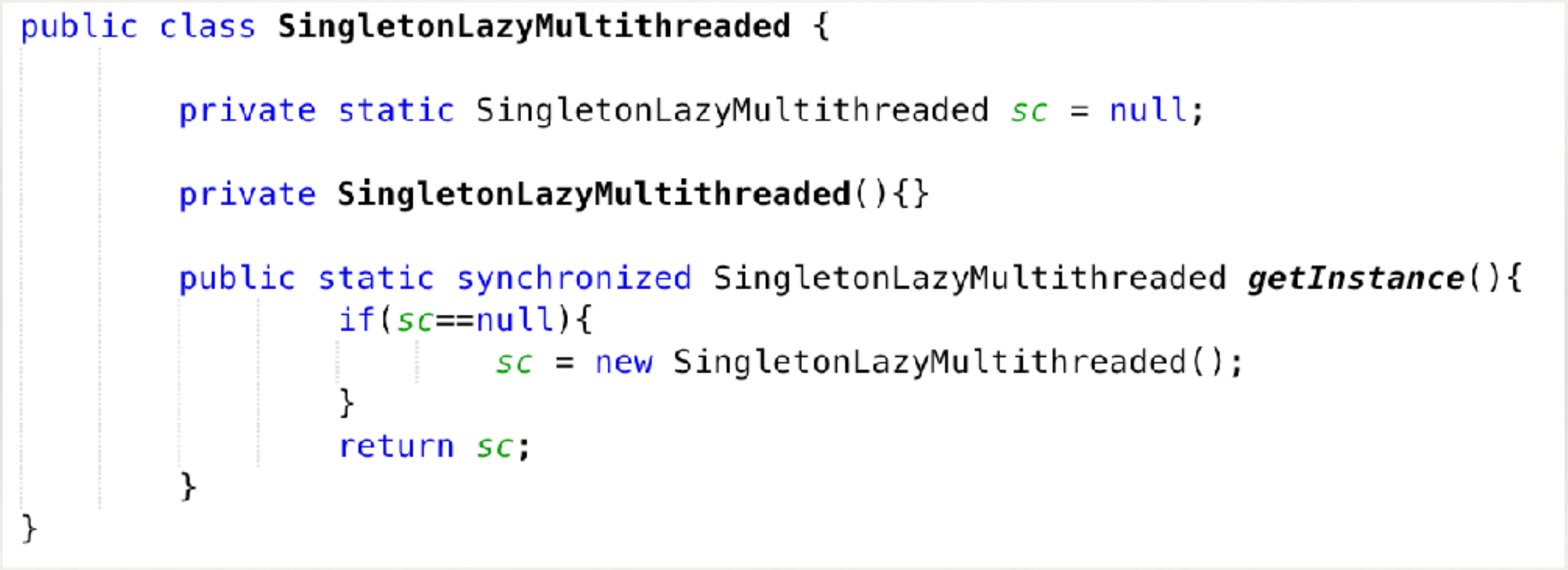
❖ It's always a good approach that an object should get created when it is required



```java
public class SingletonLazyMultithreaded {

        private static SingletonLazyMultithreaded sc = null;

        private SingletonLazyMultithreaded(){}

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

Creational Design Patterns : Singleton

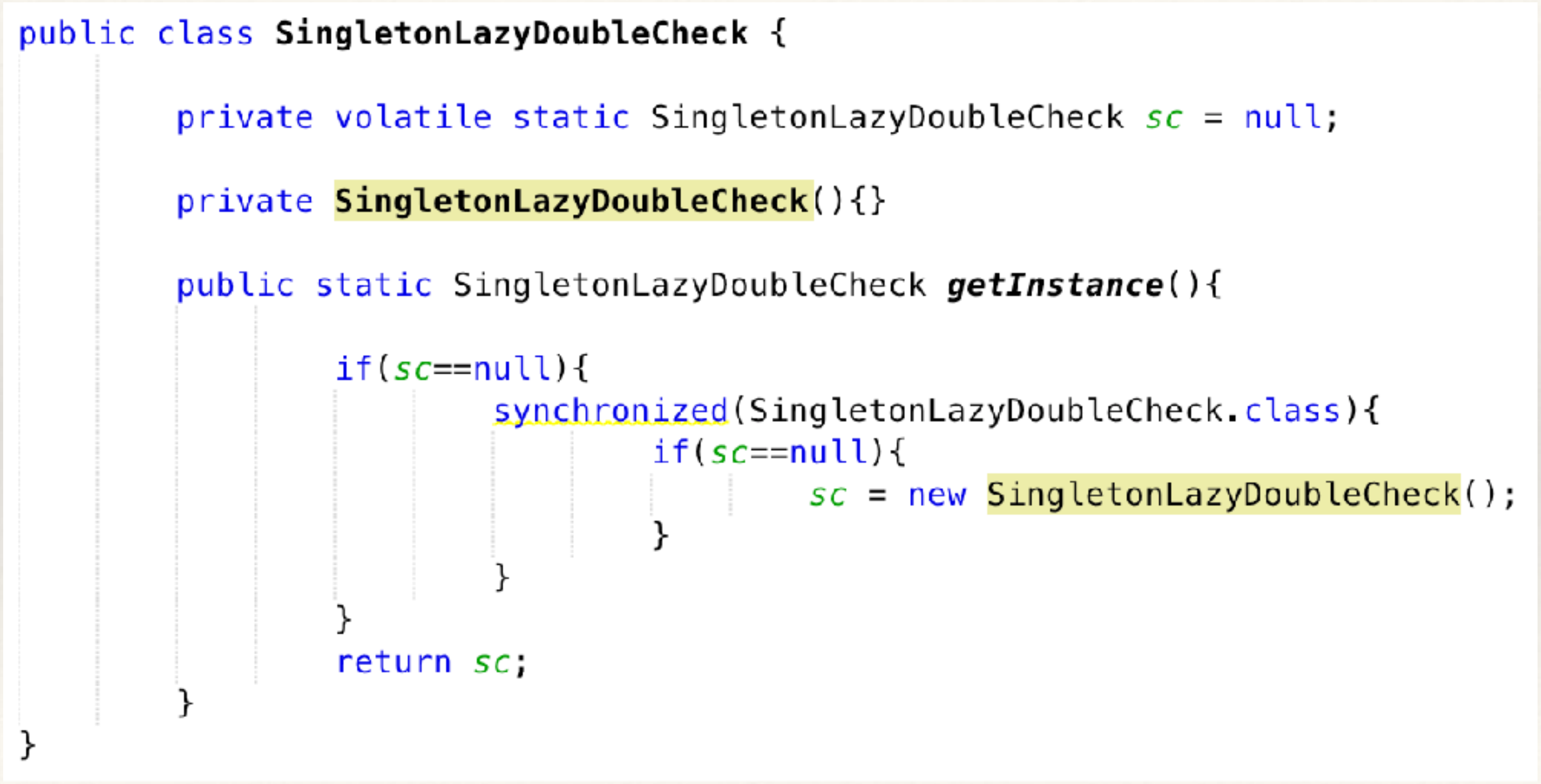
❖ **synchronized**: sadece bir thread işlem yapabilir (*no two threads will enter the method at the same time*)



```
public class SingletonLazyMultithreaded {

        private static SingletonLazyMultithreaded sc = null;

        private SingletonLazyMultithreaded(){}

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

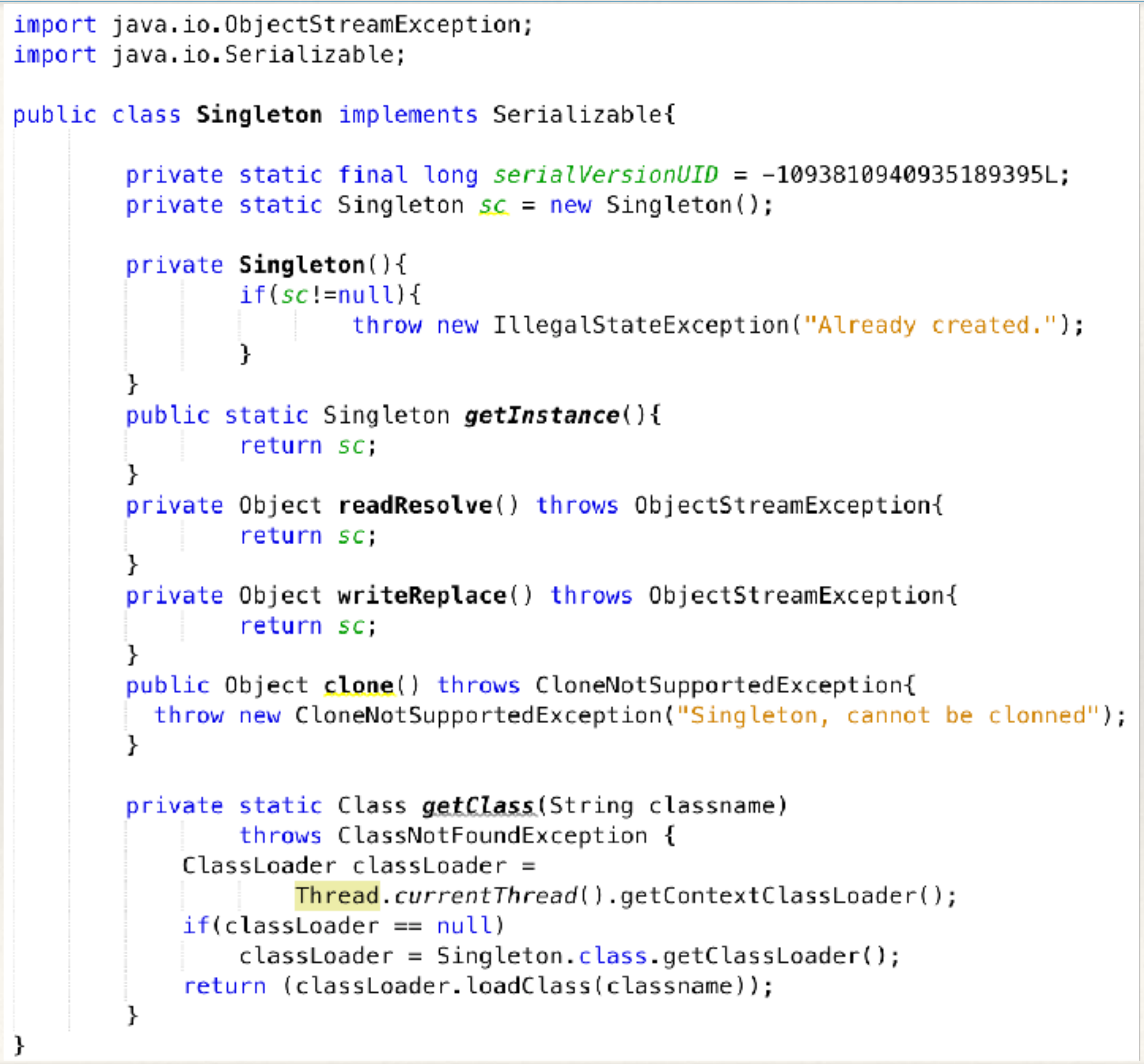
Creational Design Patterns : Singleton

- volatile: the *variable's value will be modified by different threads.*



```
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;
        }
}
```

Creational Design Patterns : Singleton



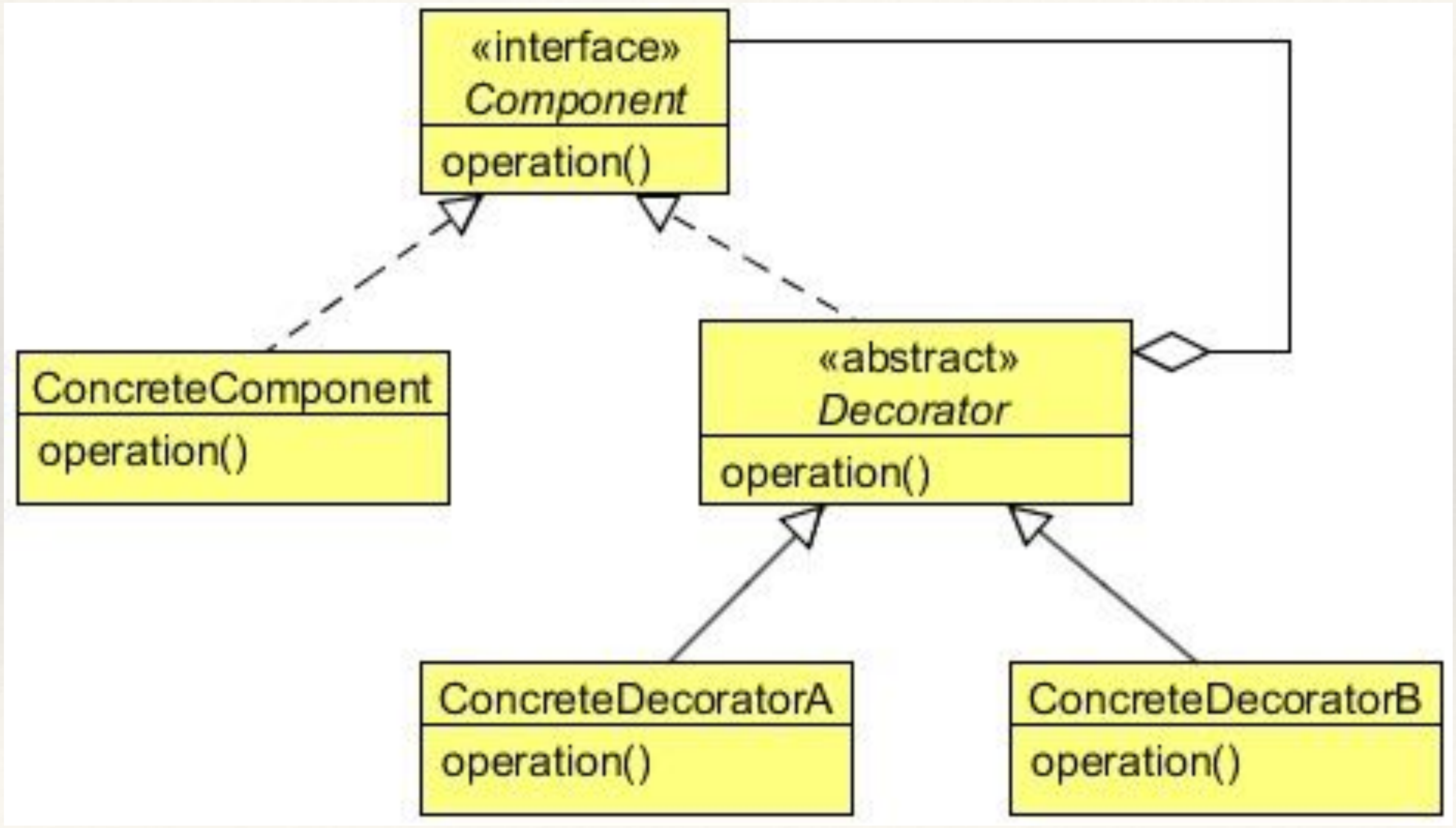
```java
import java.io.ObjectStreamException;
import java.io.Serializable;

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));
        }
}
```

Structural Design Patterns: Decorator

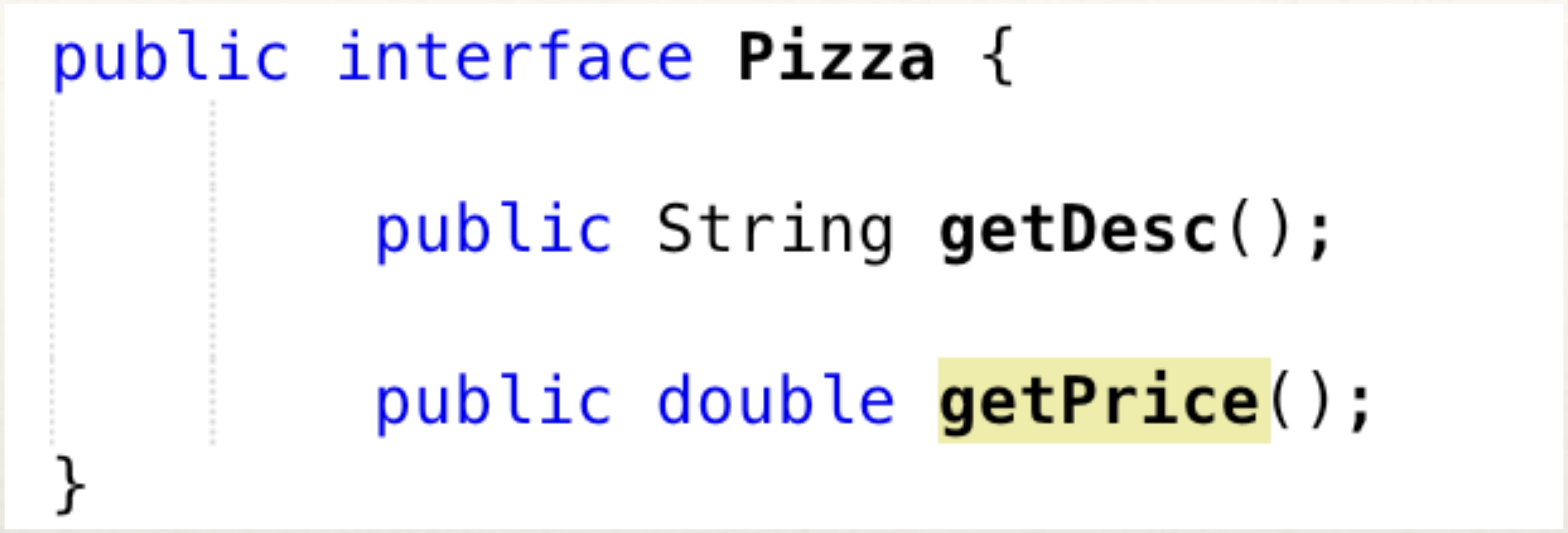
- Bir nesneye dinamik olarak özellik ve sorumluluk eklemek için kullanılır.
- Decorator pattern used to ***extend the functionality of an object dynamically*** without having to change the original class source or using inheritance.

Structural Design Patterns: Decorator

- Decorator pattern used to extend the functionality of an object dynamically without having to change the original class source or using inheritance.

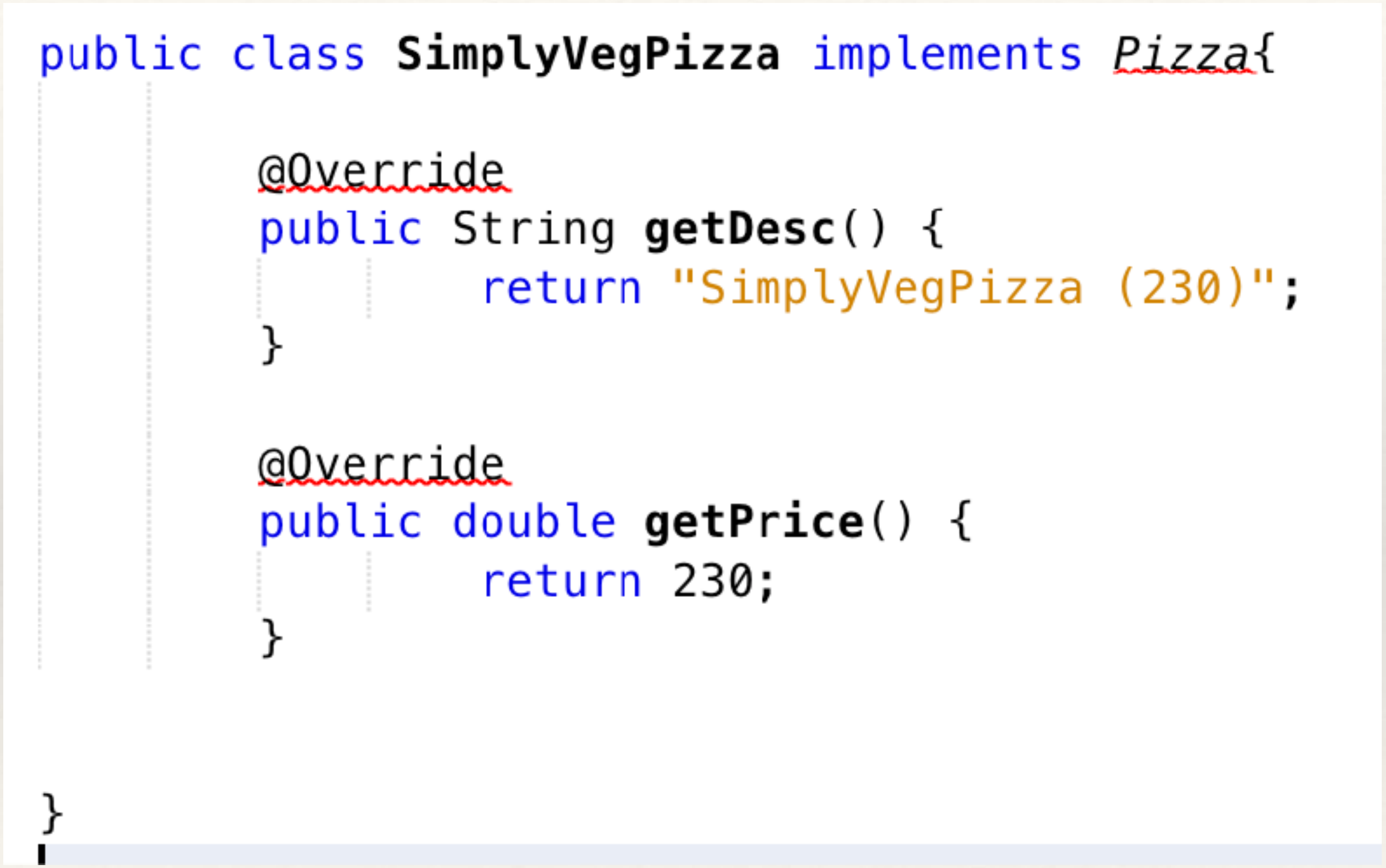


Structural Design Patterns: Decorator



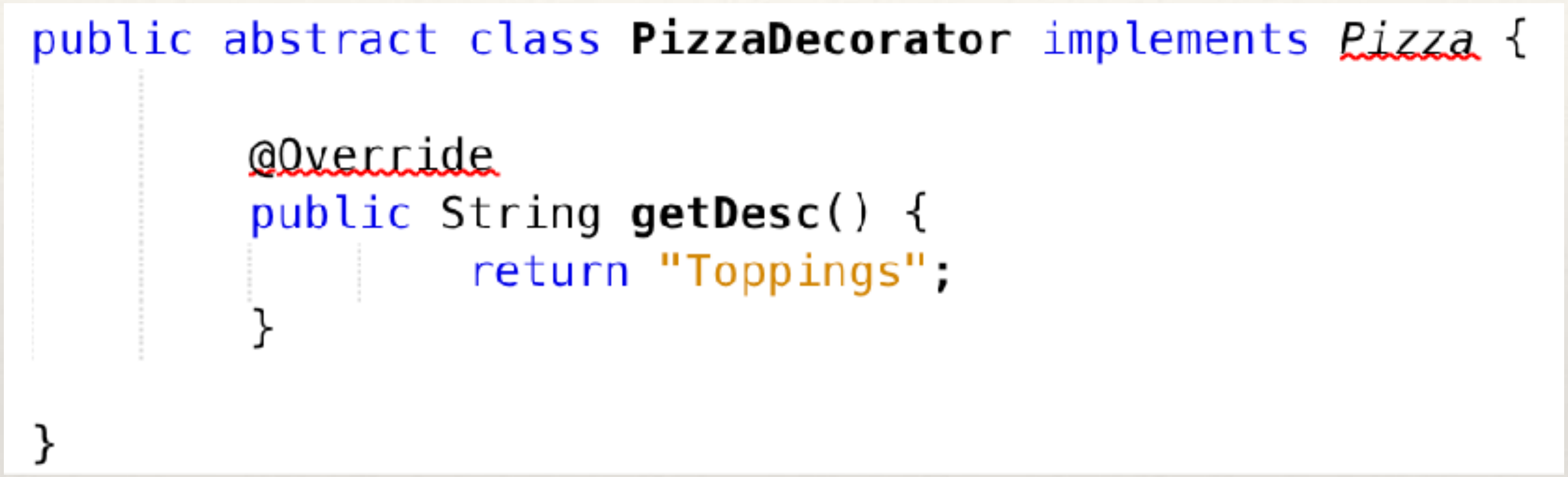
```
public interface Pizza {

        public String getDesc();

        public double getPrice();
}
```

Structural Design Patterns: Decorator



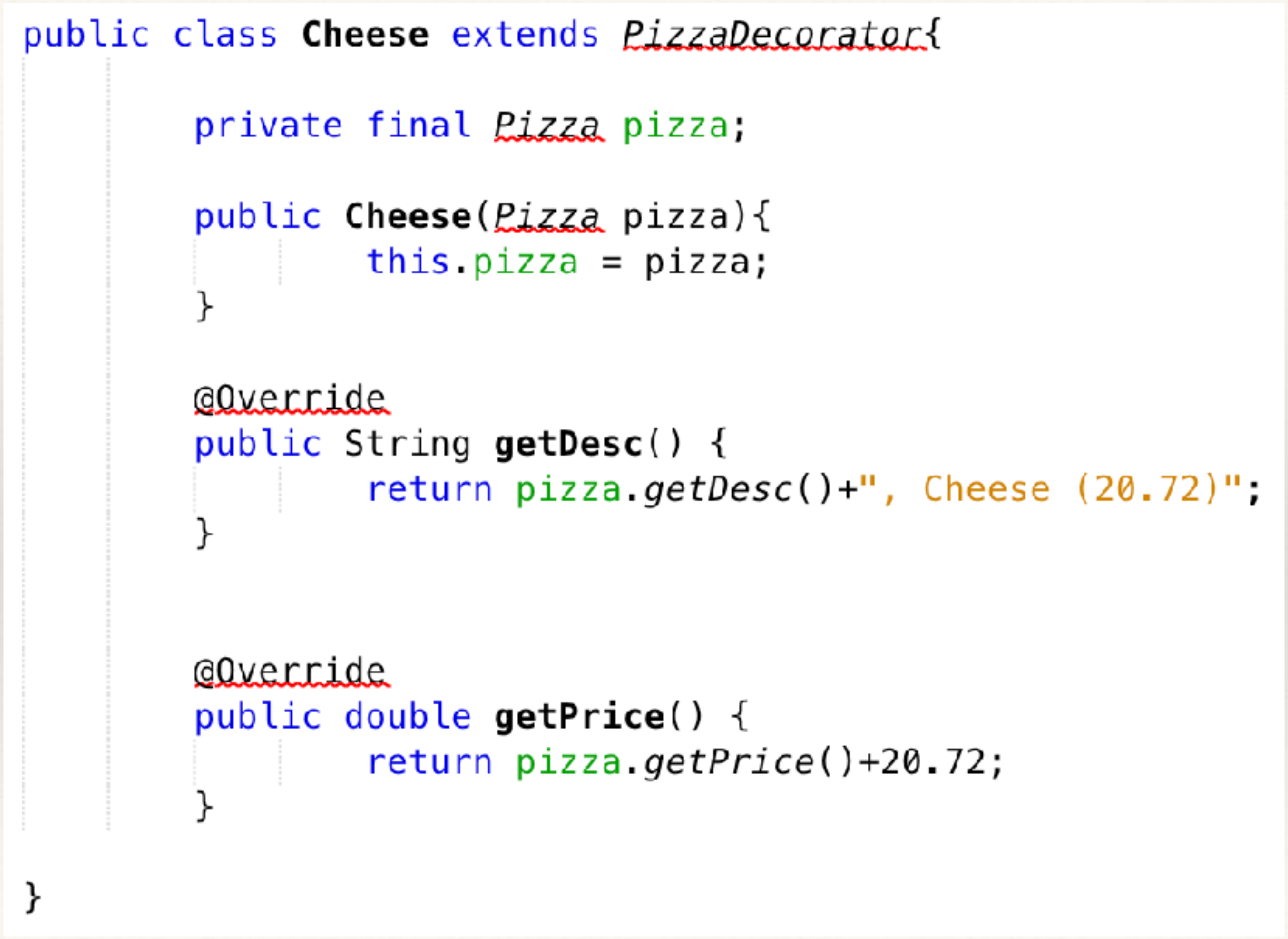
```
public class SimplyVegPizza implements Pizza{

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

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


}
```

Structural Design Patterns: Decorator



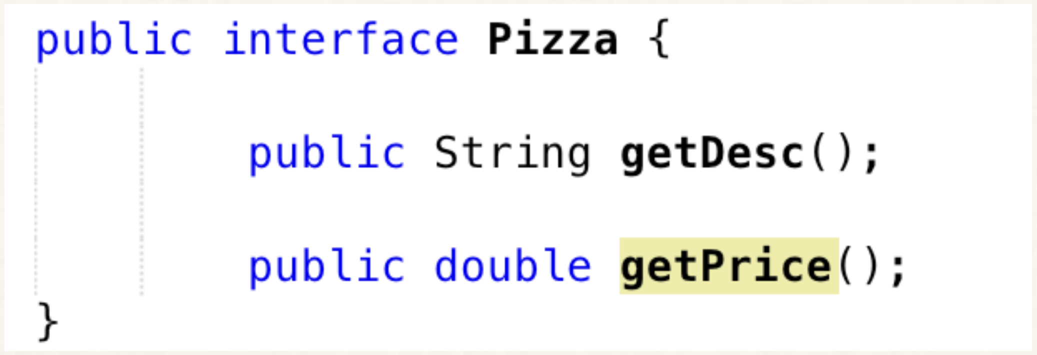
```java
public abstract class PizzaDecorator implements Pizza {

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

}
```

Structural Design Patterns: Decorator

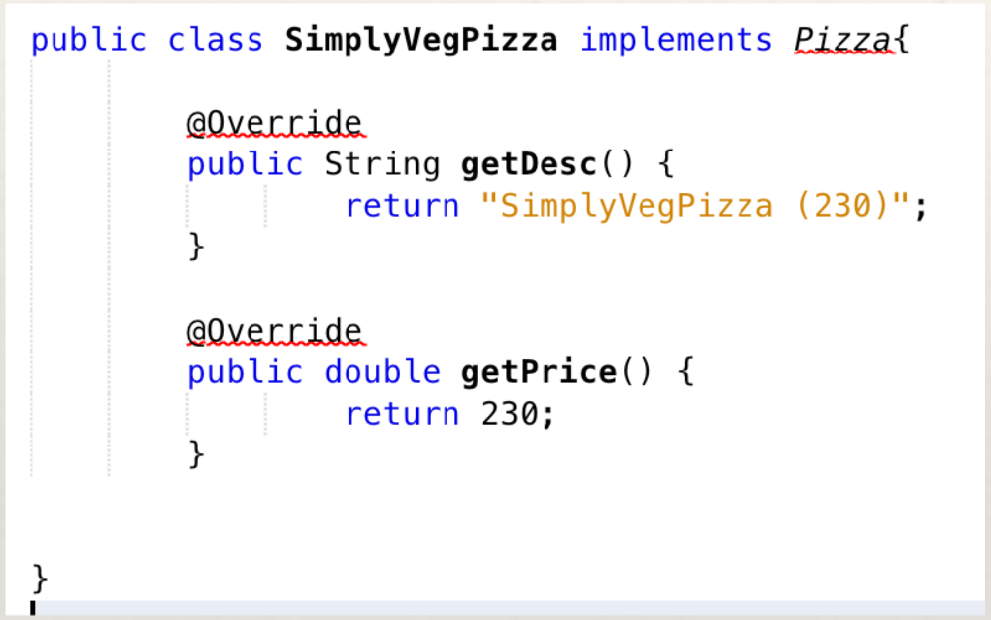


```java
public class Cheese extends PizzaDecorator{

        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;
        }

}
```

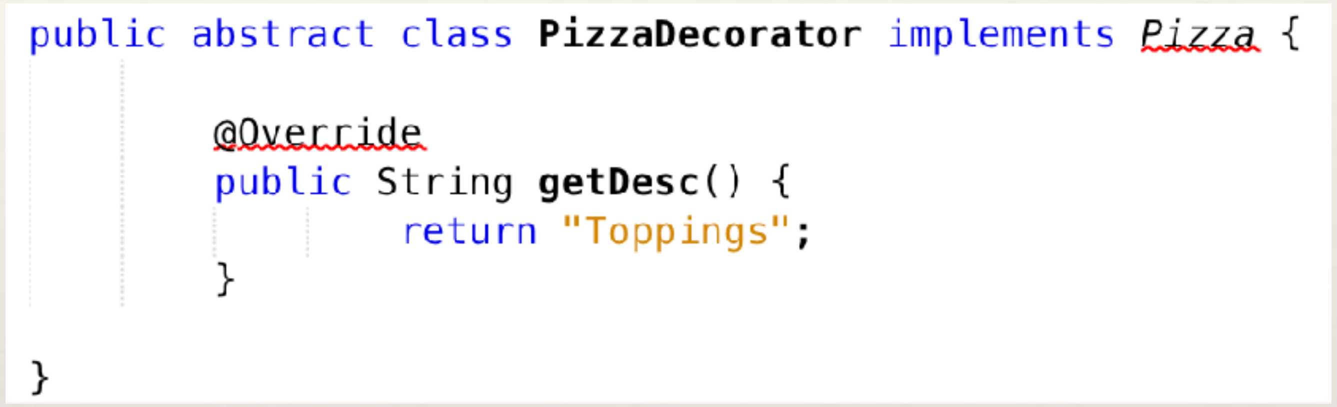
Structural Design Patterns: Decorator



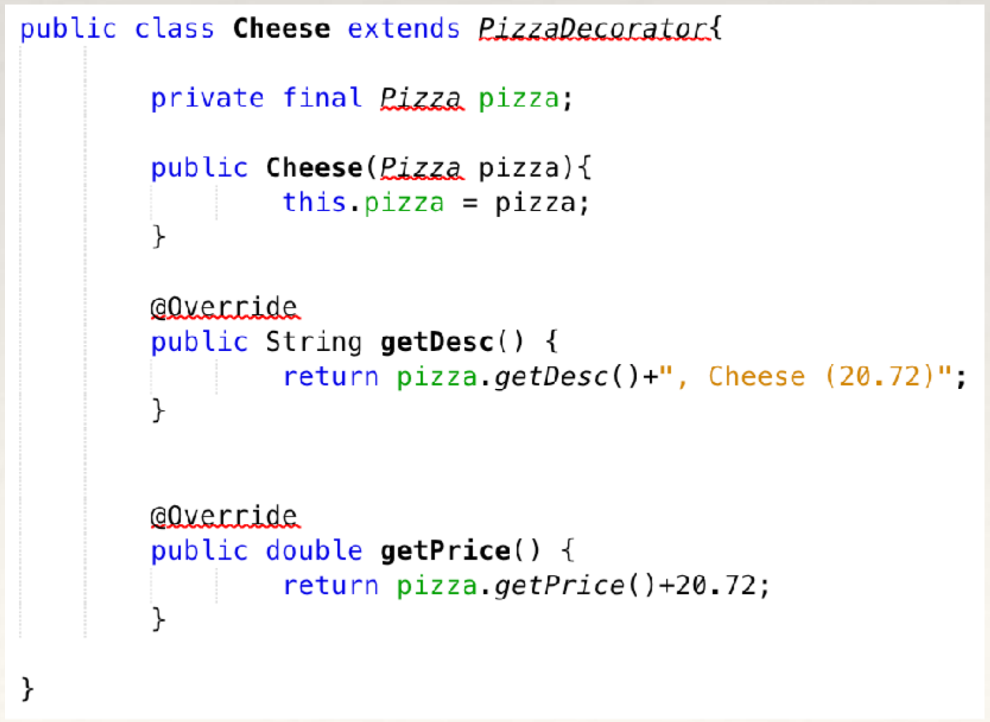
```java
public interface Pizza {

        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;
        }

}
```

Structural Design Patterns: Decorator

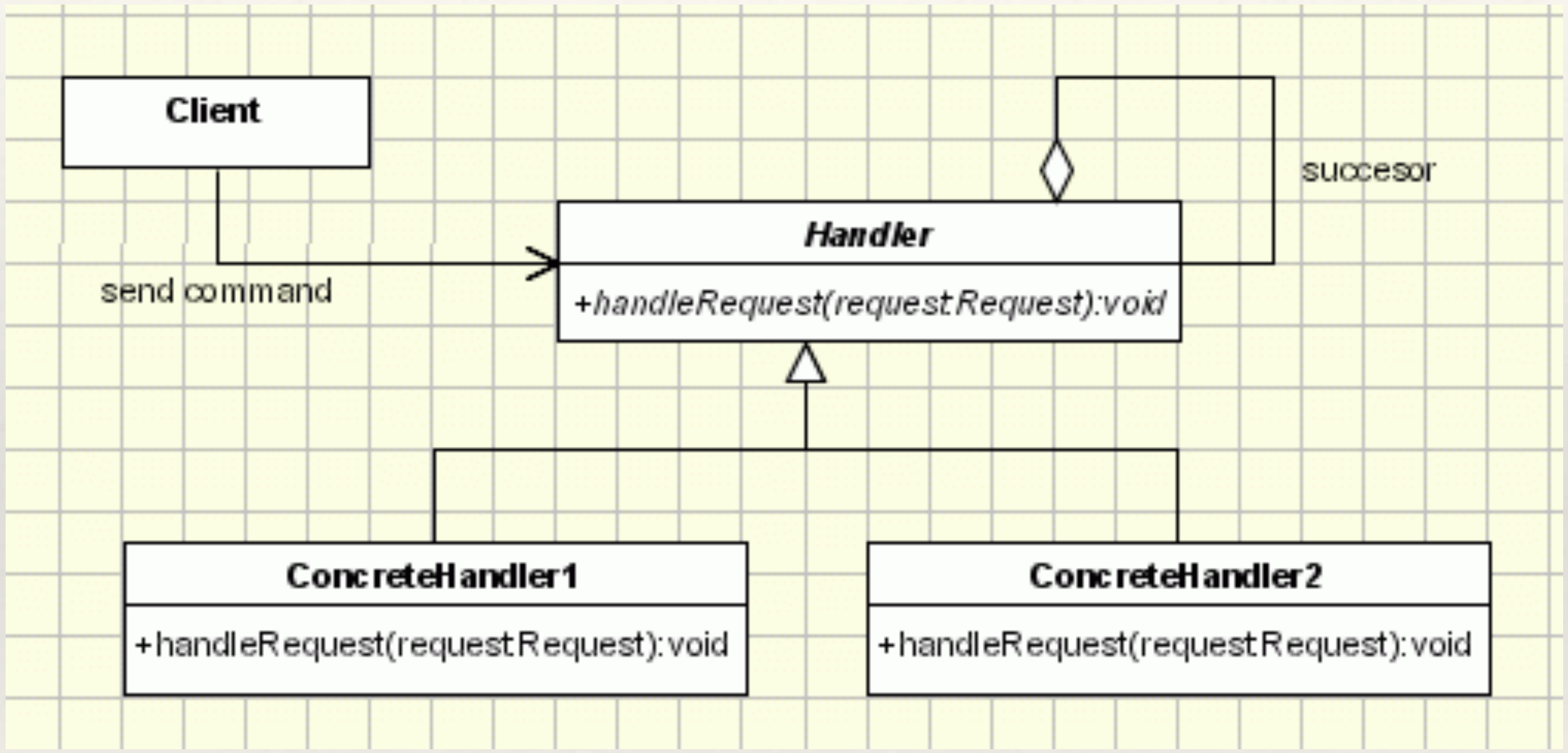
- java.io.BufferedInputStream(InputStream)
- java.io.DataInputStream(InputStream)
- java.io.BufferedOutputStream(OutputStream)
- java.util.zip.ZipOutputStream(OutputStream)
- java.util.Collections#checked[List|Map|Set|SortedSet|SortedMap]()

Behavioral Design Patterns: Chain of Responsibility

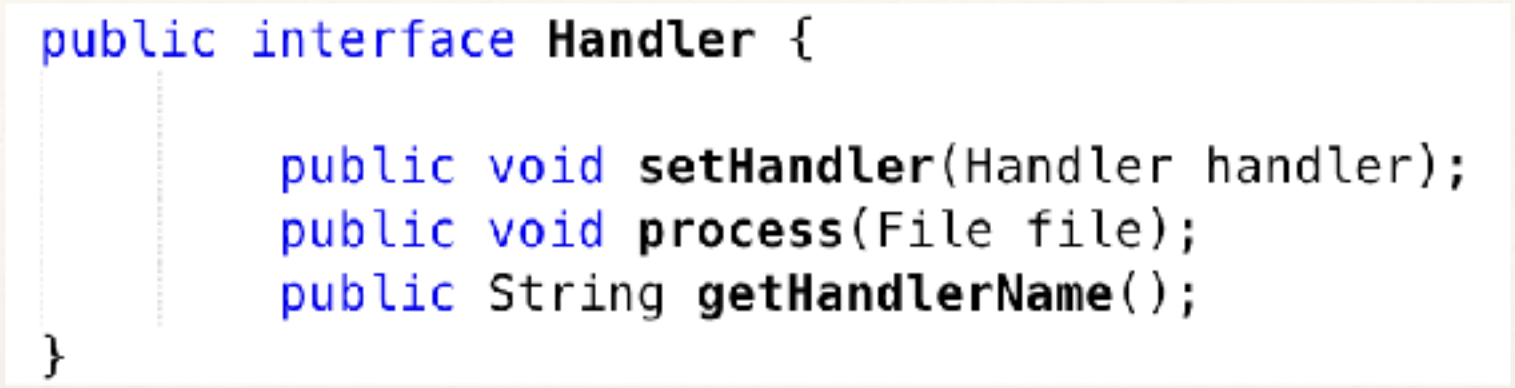
- Bir problemin çözümünün birden fazla nesnenin işlem için beklemesi ile oluşur
    - When a request comes to a single object, it will check whether it can process and handle the specific file format. If it can, it will process it; otherwise, it will forward it to the next object chained to it

    - farklı formatta bulunan verinin analizini gerçekleştirecek farklı nesnelerin oluşturulması
        - text verisini işleyen nesne ile video verisini işleyen nesne farklıdır fakat ikiside veri analizi için işlem gerçekleştirilir

Behavioral Design Patterns: Chain of Responsibility

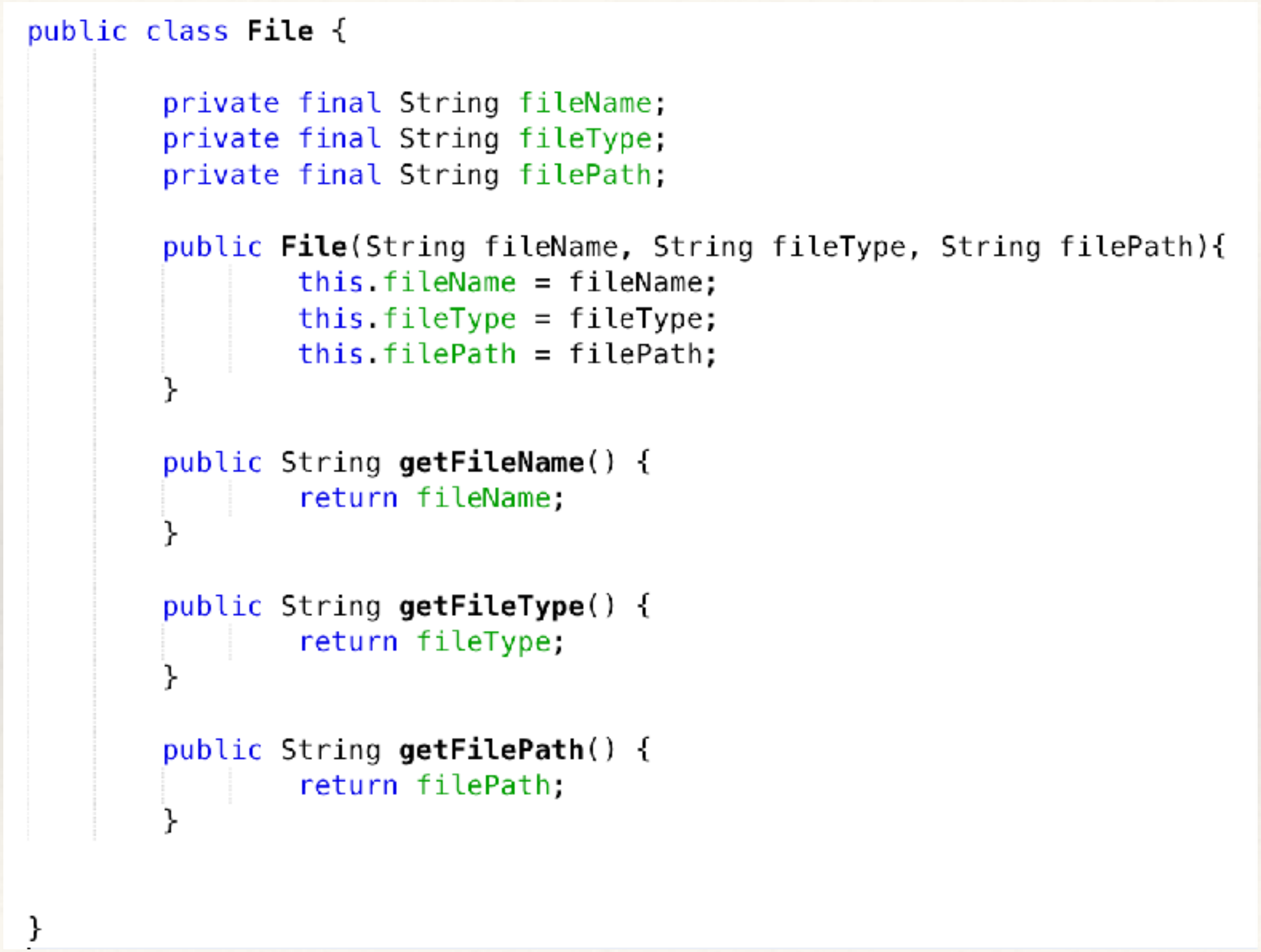

Client
send command
Handler
+handleRequest(request:Request):void
succesor
ConcreteHandler1
+handleRequest(requestRequest):void
ConcreteHandler2
+handleRequest(requestRequest):void

Behavioral Design Patterns: Chain of Responsibility



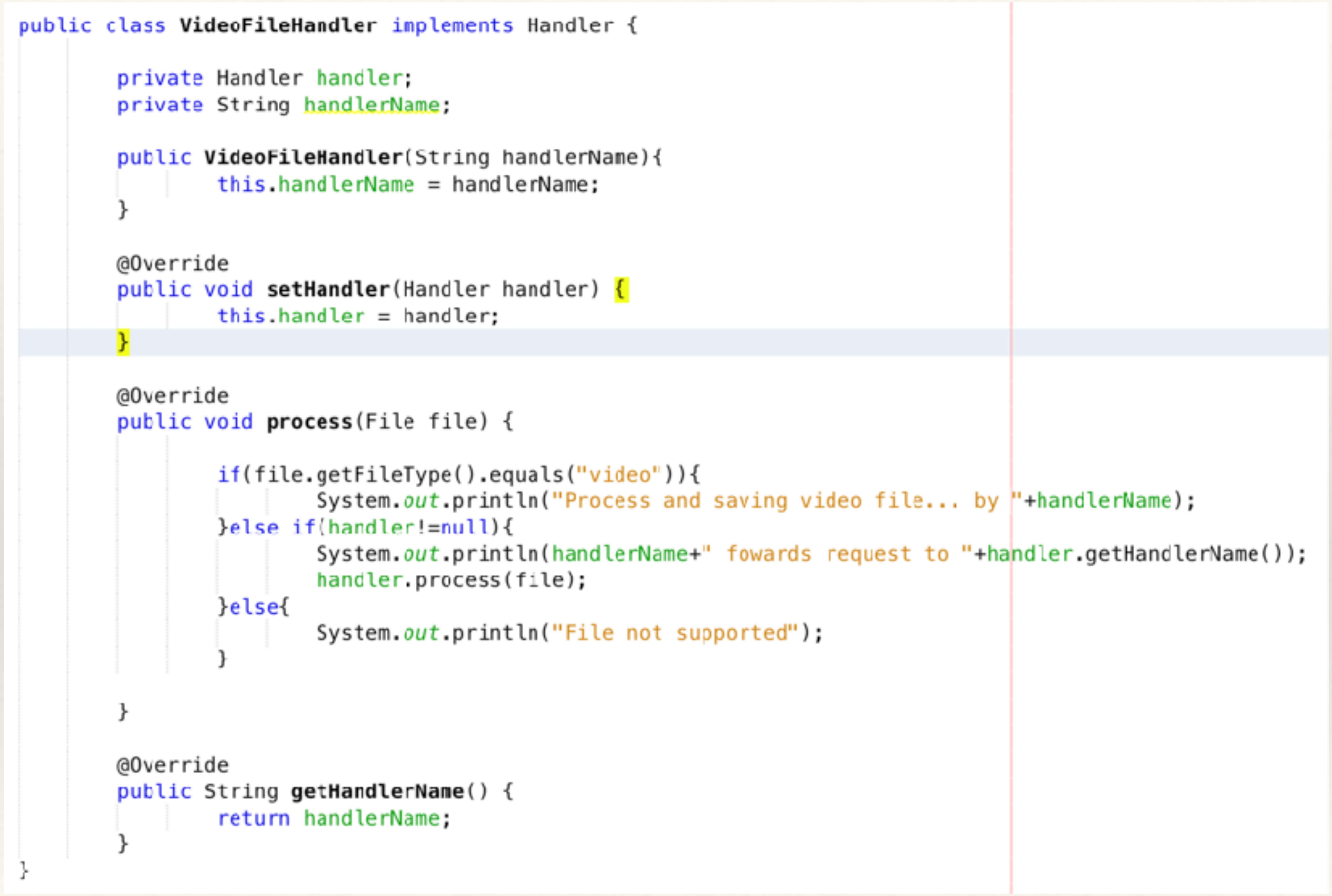
```
public interface Handler {

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

Behavioral Design Patterns: Chain of Responsibility



```
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;
        }


}
```

Behavioral Design Patterns: Chain of Responsibility



```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;
        }
}
```