What are design patterns?

- ✓ The patterns typically show relationships and interactions between classes or objects
- ✓ Patterns and their consequences of use have been thought through by experienced designers
- ✓ Helpful for good OO design

Main rule of design patterns

Simple. Basically when you try to **encapsulate** the things that vary from object to object as separate classes, you are executing some form of a pattern already to keep common codes together. So most likely, we are doing it unconsciously. However, is it correct? hehe

Common Design Patterns

- ✓ Singleton Pattern
- ✓ State-based Pattern
- ✓ MVC (Model View Controller)

Singleton pattern

Singleton: an object that is the only object of its type

- ensures that a class has at most one instance
- provides a global access point to that instance

Restricting object creation

why try restrict?

- ✓ creating lots of objects can take a lot of time
- ✓ extra objects take up memory
- ✓ it is a pain to deal with different objects floating around if they are essentially the same
- ✓ I use this style specially If I want a common variable for many objects. You will see in the example

Implementing Singleton

- ✓ make constructor(s) private so that they can not be called from outside
- ✓ declare a single static private instance of the class
- ✓ write a public getInstance() or similar method that allows access to the single instance

Take a look at this example..

We followed the rule of Singleton Pattern:

Our constructor is private. (Pls don't tell me you don't know what Constructor is...hehe galet)

We followed the rule of Singleton Pattern:

We created a method with "getInstance to allow access to a single instance

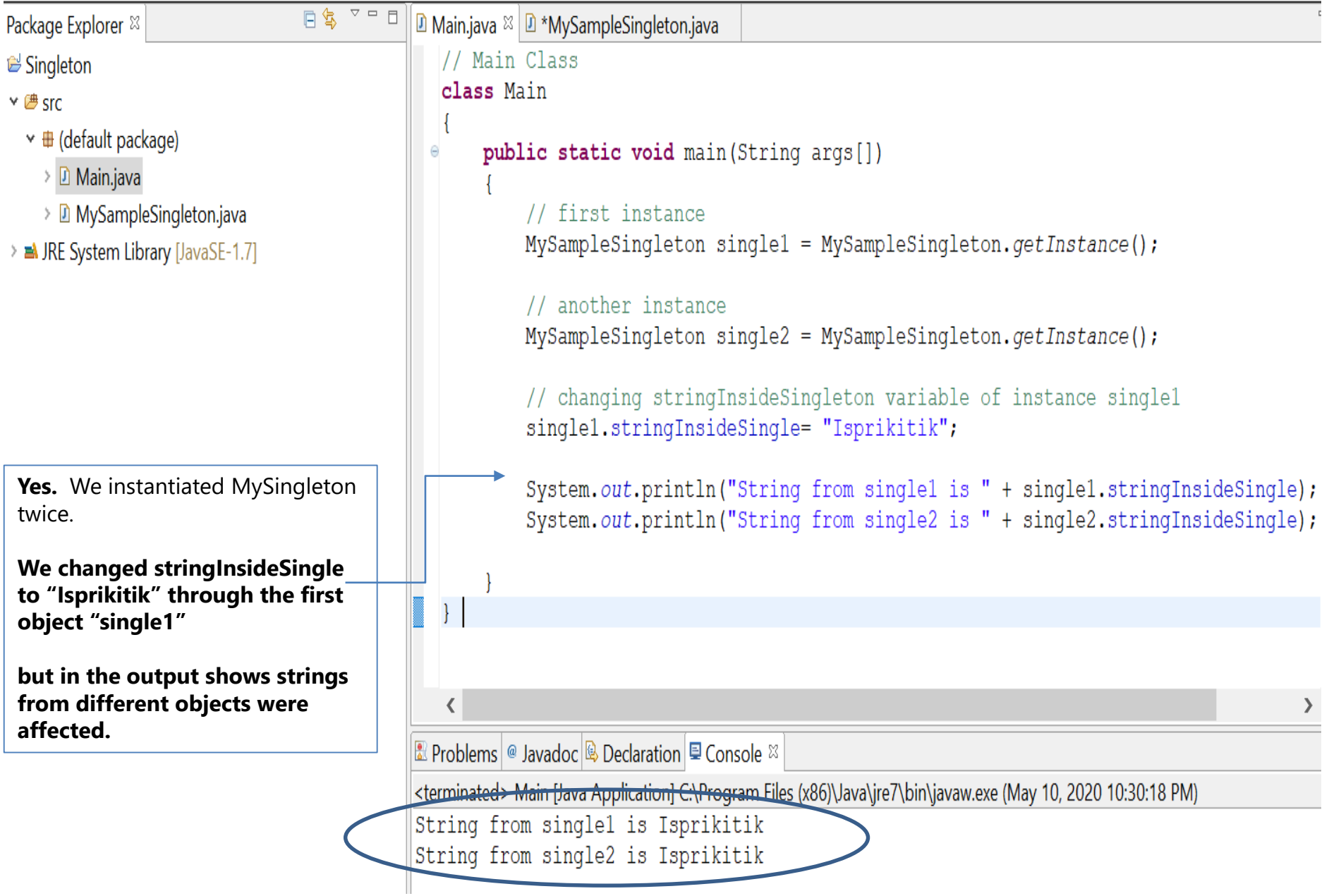
```
class MySampleSingleton
{
    // declaration of static single_instance of type Singleton
    private static MySampleSingleton single_instance = null;

    // variable inside Singleton Class
    public String stringInsideSingle;

    // private class constructor
    private MySampleSingleton()
    {
        stringInsideSingle = "Wakokok";
    }

    // static method to create instance of Singleton class
    public static MySampleSingleton getInstance()
    {
        if (single_instance == null)
            single_instance = new MySampleSingleton();

        return single_instance;
    }
}
```

Look....

We instantiated MySingleton twice. Wait? Isn't it single instance.

"SingleInstance" is not about how many objects. It's about allow access to a single instance

```
// Main Class
class Main
{
    public static void main(String args[])
    {
        // first instance
        MySampleSingleton single1 = MySampleSingleton.getInstance();

        // another instance
        MySampleSingleton single2 = MySampleSingleton.getInstance();

        // changing stringInsideSingleton variable of instance single1
        single1.stringInsideSingle= "Isprikitik";

        System.out.println("String from single1 is " + single1.stringInsideSingle);
        System.out.println("String from single2 is " + single2.stringInsideSingle);

    }
}
```



Yes. We instantiated MySingleton twice.

We changed stringInsideSingle to "Isprikitik" through the first object "single1"

but in the output shows strings from different objects were affected.

Singleton Summary

In the main class, we instantiate the singleton class with 2 objects single1 and single2 by calling **static method getInstance().** After creation of object **single1**, variable **stringInsideSingle** of single2 is pointed to object single1. Hence, if we change the variables of object single1, that is reflected when we access the variables of object single1. Also, if we change the variables of object single2, that is reflected when we access the variables of objects single1.

Class vs Interface

Class and **Interface** both are used to create new reference types.

A **class** is a collection of fields and methods that operate on fields.

An **interface** has fully abstract methods(i.e. methods with no body.)

CLASS vs INTERFACE

CLASS

```
class Example1{
  public void display1(){
    System.out.println("yeah method1");
  }
}
```

INTERFACE

```
public interface MyInterface {
    public void display();
    public void displayUlit();
}
```

Interface

- ✓ **Interface** looks like a class but it is **not a class**.
- ✓ **Interface** can have **methods** and **variables** just like the class but the methods declared in interface are by default **abstract** (only method signature, no body)
- ✓ The variables declared in an interface are **public, static & final** by default.

Some more points to ponder upon..

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.->that's the real deal 'coz we can only extend ONE if it is just a CLASS

Interface: Syntax

interface is a reserved word

```
public interface Doable
{
   public static final String NAME;

   public void doThis();
   public int doThat();
   public void doThis2 (float value, char ch);
   public boolean doTheOther (int num);
}
```

NOTE: No method in an interface has a definition (body)

Let's see examples…(I will provide via BBL)

```
public interface MyInterface {
        public void display();
        public void displayUlit();
}


class NewClass implements MyInterface
{

  public void display()
  {
        System.out.println("implementation of method1");
  }
  public void displayUlit()
  {
        System.out.println("implementation of method2");
  }

  public static void main(String arg[])
  {
        NewClass obj = new NewClass();
        obj.display();
        obj.displayUlit();
  }

}
```

Point to ponder upon again…

Important Note:

The class that implements interface must implement **ALL** methods of that interface

So what?

- class can extend only one class

- interface can extend any number of interfaces at a time

Summary and Conclusion

a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements

- A **class** is used when we need to define **how the task would be done.**
- An **interface** is used when we need to know **what task has to be done**.

.

Summar and More insights..

- In reality, more than just multiple implementation/extension, **Interface** is an **Integrative Coding** that allows us to make our coding style extra **readable and organized.**
- You know, we can just simply make a lot of classes with methods right? ***Why complicate with Interfaces?*** Eh kase in normal classes diba we do not declare methods before we define or use them right?
- If you have plenty of methods and classes already, I regret to read the list of codes inside your methods and classes just to check one method. Instead, I can just simply look at your interface first, specially if I will just be checking for something and I just need to understand your code generally.
- I won't do that in a code with thousands of lines!!!! Thanks Interface

Let’s see basic sample coding style using state-based Pattern via Interface. The code is in java. As I’ve mentioned, we usually use it in games.

State Pattern

- ✓ Real time use case
- ✓ Cause an object to behave in a manner determined by its state
- ✓ **State pattern is one of the heavily used pattern in game development.**
- ✓ Different states are used in form of Interfaces

Example, here we create GameContext class, given a Player Class

```
public class GameContext {
private Player player = new Player();
public void gameAction(String state)
{
    if (state == "GodLike") {
            player.attack();
            player.layBomb();
            player.firePistol();
    } else if (state == "Weak") {
             player.weak();
             player.fireSemiPistol();
    } else if (state == "dead") {
        player.dead();
    }
}
```

```
public class Player {

    public void attack() {
        System.out.println("Attack");
    }

    public void layBomb() {
        System.out.println("Lay Bomb");
    }

    public void fireSemiPistol() {
        System.out.println("Semi Pistols");
    }

    public void firePistol() {
        System.out.println("Fire Pistol");
    }

    public void weak() {
        System.out.println("I am WEAK!");
    }

    public void dead() {
        System.out.println("Game Over!");
    }

}
```

Let’s create an Interface for PlayerState. This will perform action

```
public interface PlayerState {
    void action(Player p);
}
```

We create Separate Classes per State, implementing PlayerState interface

```
public class GodLikeState implements PlayerState {

    public void action(Player p) {
        p.attack();
        p.firePistol();
        p.layBomb();
    }
}
```

```
public class WeakState implements PlayerState{
    public void action(Player p) {
        p.fireSemiPistol();
        p.attack();
    }
}
```

```
public class DeadState implements PlayerState {
    public void action(Player p) {
        p.dead();
    }
}
```

Game context to instantiate players player state and game action to spawn

```
public class GameContext {
    private PlayerState state = null;
    private Player player = new Player();

    public void setState(PlayerState state) {
        this.state = state;
    }

    public void gameAction() {
        state.action(player);
    }
}
```

Testing the Game

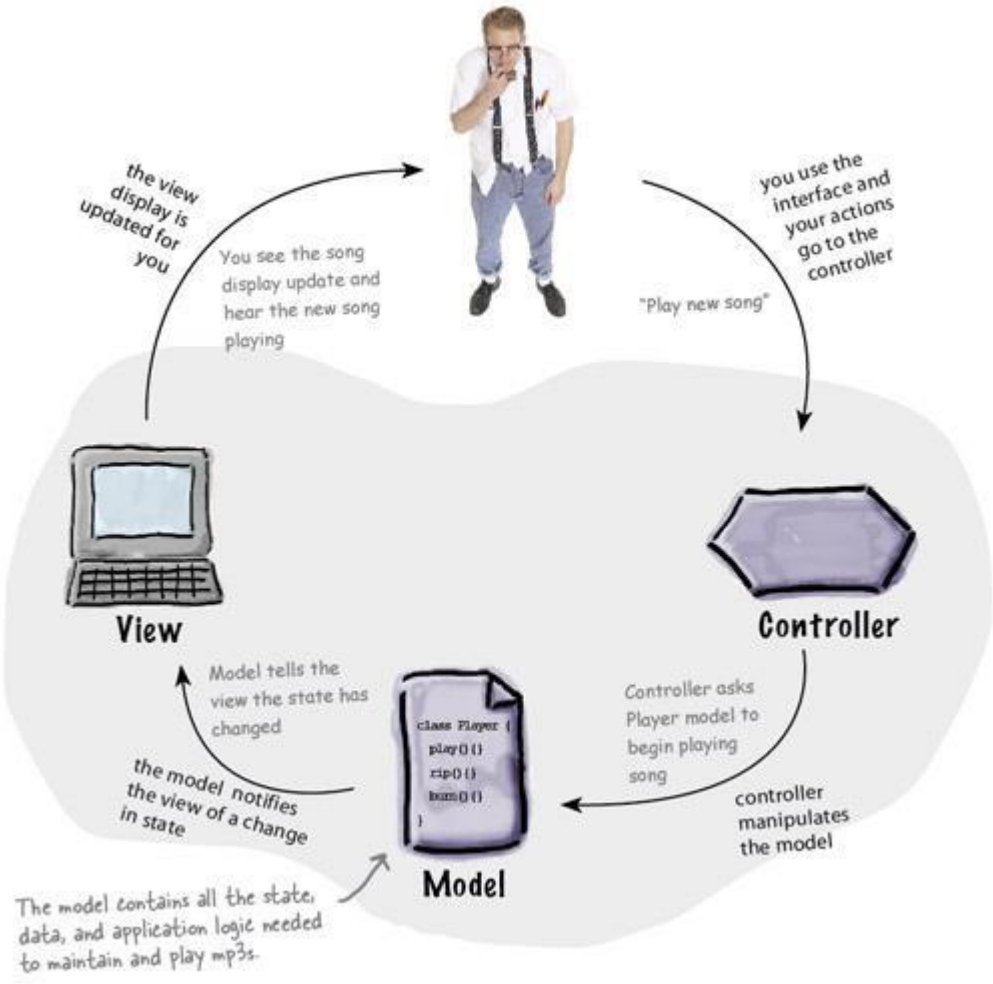
```
public class SampleGameTest {
    public static void main(String[] args) {
        GameContext context = new GameContext();

        context.setState(new GodLikeState());
        context.gameAction();
        System.out.println("******");

        context.setState(new WeakState());
        context.gameAction();
        System.out.println("******");

        context.setState(new DeadState());
        context.gameAction();
        System.out.println("******");
    }
}
```

MVC Pattern

MVC Pattern stands for Model-View-Controller Pattern.

- **Model** - Model represents an object carrying data. It updates controller if its data changes.
- **View** - View represents the visualization of the data that model contains.
- **Controller** - Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

Note: MVC Design Pattern will be discussed in the context of Python Web in the succeeding topics



Quick Summary

Encapsulate the things that vary from object to object as separate classes

- But keep common code together

Use Design to describe what to do and what not to do when implementing the patterns in your project