Adapter Design Pattern

Adapter Design Pattern

- Adapter pattern works as a bridge between two incompatible interfaces.
- This type of design pattern comes under **structural pattern** as this pattern combines the capability of two independent interfaces.
- This pattern involves a **single class** which is responsible to **join functionalities of independent or incompatible interfaces**.

Example 1

- A real life example could be a case of card reader which acts as an adapter between memory card and a laptop. You plugin the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.

Example 2

- A real life example could be a Car mobile charger which acts as an adapter between cell phone USB cable and a car cigarette lighter power supply. You plugin the USB cable into car charger interface and car charger into the cigarette lighter power supply so that your cell phone can get the appropriate voltage required for charging.

Example 3

- We are demonstrating use of Adapter pattern via following example in which an audio player device can play mp3 files only and wants to use an advanced audio player capable of playing vlc and mp4 files.

Implementation

- We have a *MediaPlayer* interface and a concrete class *AudioPlayer* implementing the *MediaPlayer* interface. *AudioPlayer* can play mp3 format audio files by default.

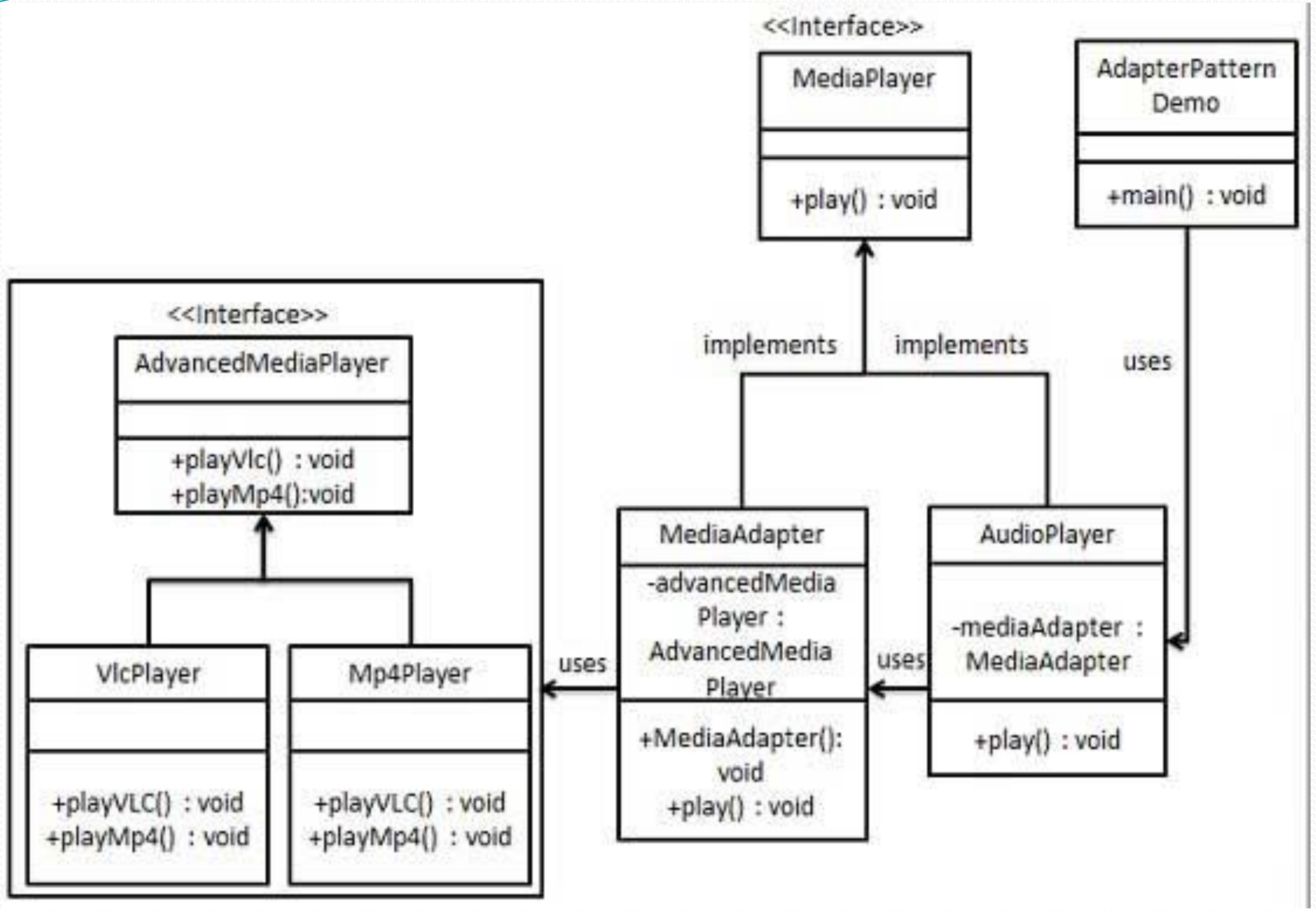
- We are having another interface *AdvancedMediaPlayer* and concrete classes implementing the *AdvancedMediaPlayer* interface. These classes can play vlc and mp4 format files.

Implementation

- We want to make *AudioPlayer* to play other formats as well. To attain this, we have created an adapter class *MediaAdapter* which implements the *MediaPlayer* interface and uses *AdvancedMediaPlayer* objects to play the required format.

- *AudioPlayer* uses the adapter class *MediaAdapter* passing it the desired audio type without knowing the actual class which can play the desired format. *AdapterPatternDemo*, our demo class will use *AudioPlayer* class to play various formats.

MediaAdapter



Step 1

- Create interfaces for Media Player and Advanced Media Player.
- *MediaPlayer.java*

```
public interface MediaPlayer {
   public void play(String audioType, String fileName);
}
```

- *AdvancedMediaPlayer.java*

```
public interface AdvancedMediaPlayer {
   public void playVlc(String fileName);
   public void playMp4(String fileName);
}
```

Step 2

- Create concrete classes implementing the *AdvancedMediaPlayer* interface.
- *VlcPlayer.java*

```
public class VlcPlayer implements AdvancedMediaPlayer{
   @Override
   public void playVlc(String fileName) {
      System.out.println("Playing vlc file. Name: "+ fileName);
   }

   @Override
   public void playMp4(String fileName) {
      //do nothing
   }
}
```

Step 2

- Create concrete classes implementing the *AdvancedMediaPlayer* interface.
- *Mp4Player.java*

```
public class Mp4Player implements AdvancedMediaPlayer{
   @Override
   public void playVlc(String fileName) {
      //do nothing
   }
   @Override
   public void playMp4(String fileName) {
      System.out.println("Playing mp4 file. Name: "+ fileName);

   }
}
```

Step 3

- Create adapter class implementing the *MediaPlayer* interface. (*MediaAdapter.java)*

```
public class MediaAdapter implements MediaPlayer {
   AdvancedMediaPlayer advancedMusicPlayer;
   public MediaAdapter(String audioType){
      if(audioType.equalsIgnoreCase("vlc") ){
         advancedMusicPlayer = new VlcPlayer();
      } else if (audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer = new Mp4Player();
   } }
   @Override
   public void play(String audioType, String fileName) {
      if(audioType.equalsIgnoreCase("vlc")){
         advancedMusicPlayer.playVlc(fileName);
      } else if(audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer.playMp4(fileName);
      }
   }
}
```

Step 4

- Create concrete class implementing the *MediaPlayer* interface.
- *AudioPlayer.java*

```
public class AudioPlayer implements MediaPlayer {
   MediaAdapter mediaAdapter;
   @Override
   public void play(String audioType, String fileName) {
      //inbuilt support to play mp3 music files
      if(audioType.equalsIgnoreCase("mp3")){
         System.out.println("Playing mp3 file. Name: " + fileName);
      }
      //mediaAdapter is providing support to play other file formats
      else if(audioType.equalsIgnoreCase("vlc") ||
         audioType.equalsIgnoreCase("mp4")){
         mediaAdapter = new MediaAdapter(audioType);
         mediaAdapter.play(audioType, fileName);
      } else{
         System.out.println("Invalid media. " + audioType + " format not
         supported");
      }
   }}
```

Step 5

- Use the AudioPlayer to play different types of audio formats.
- *AdapterPatternDemo.java*

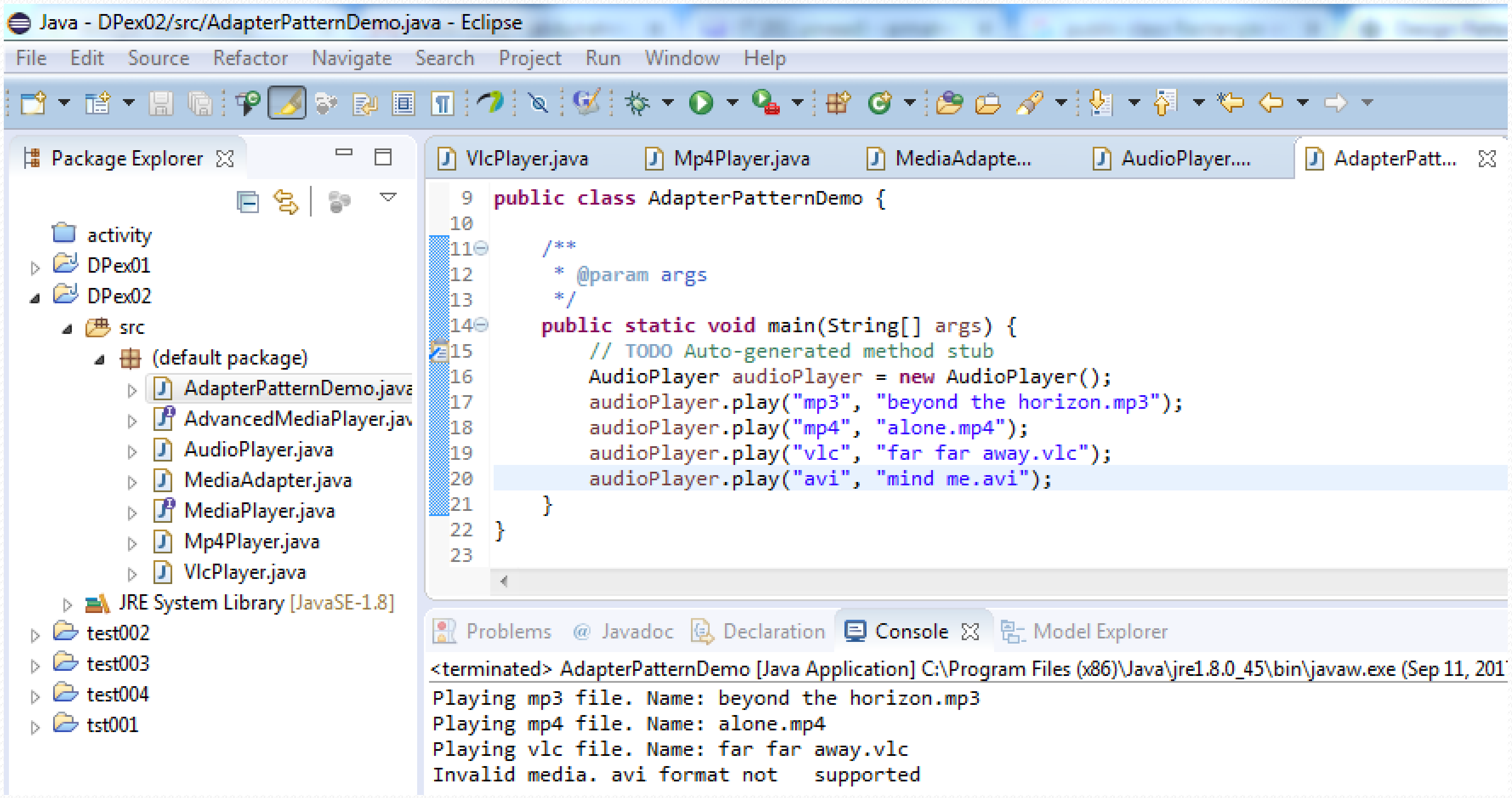
```
public class AdapterPatternDemo {
   public static void main(String[] args) {
      AudioPlayer audioPlayer = new AudioPlayer();
      audioPlayer.play("mp3", "beyond the horizon.mp3");
      audioPlayer.play("mp4", "alone.mp4");
      audioPlayer.play("vlc", "far far away.vlc");
      audioPlayer.play("avi", "mind me.avi");
   }
}
```

Step 6

- Verify the output.

```
Playing mp3 file. Name: beyond the horizon.mp3
Playing mp4 file. Name: alone.mp4
Playing vlc file. Name: far far away.vlc
Invalid media. avi format not supported
```

Eclipse workspace



```java
public class AdapterPatternDemo {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        AudioPlayer audioPlayer = new AudioPlayer();
        audioPlayer.play("mp3", "beyond the horizon.mp3");
        audioPlayer.play("mp4", "alone.mp4");
        audioPlayer.play("vlc", "far far away.vlc");
        audioPlayer.play("avi", "mind me.avi");
    }
}
```

```
<terminated> AdapterPatternDemo [Java Application] C:\Program Files (x86)\Java\jre1.8.0_45\bin\javaw.exe (Sep 11, 201
Playing mp3 file. Name: beyond the horizon.mp3
Playing mp4 file. Name: alone.mp4
Playing vlc file. Name: far far away.vlc
Invalid media. avi format not   supported
```

Singleton Design Pattern

Singleton Pattern

- Singleton pattern is one of the **simplest design patterns** in Java.

- This type of design pattern comes under **creational pattern** as this pattern provides one of the best ways to create an object.

Singleton Pattern

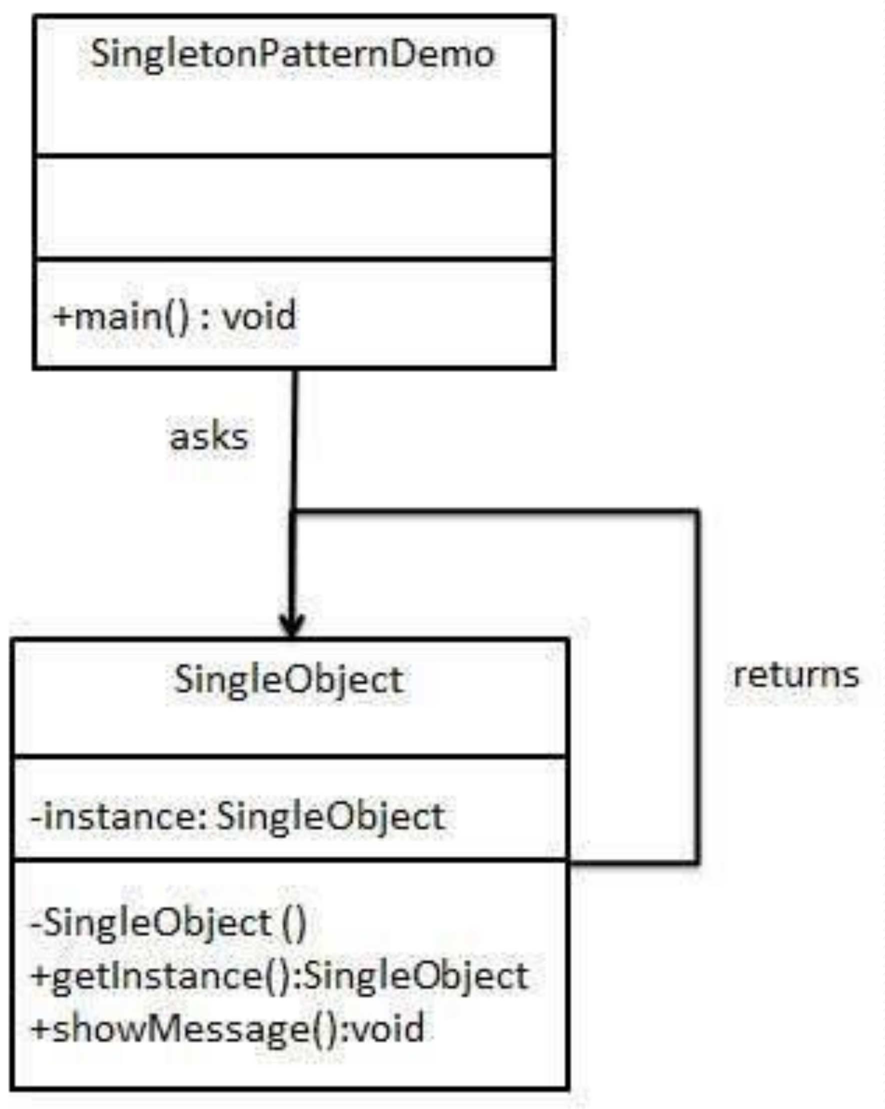
- This pattern involves a **single class** which is responsible to create an object while making sure that **only single object gets created**.

- This class provides a way to access its only object which can be accessed directly **without need to instantiate the object of the class.**

Implementation

- We're going to create a *SingleObject* class.
- *SingleObject* class have its **constructor as private and have a static instance of itself**.
- *SingleObject* class **provides a static method to get its static instance** to outside world.
- *SingletonPatternDemo*, our demo class will **use *SingleObject* class** to get a *SingleObject* object.

Implementation



Step 1

- Create a Singleton Class. (*SingleObject.java*)

```
public class SingleObject {
   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();
   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}
   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }
   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

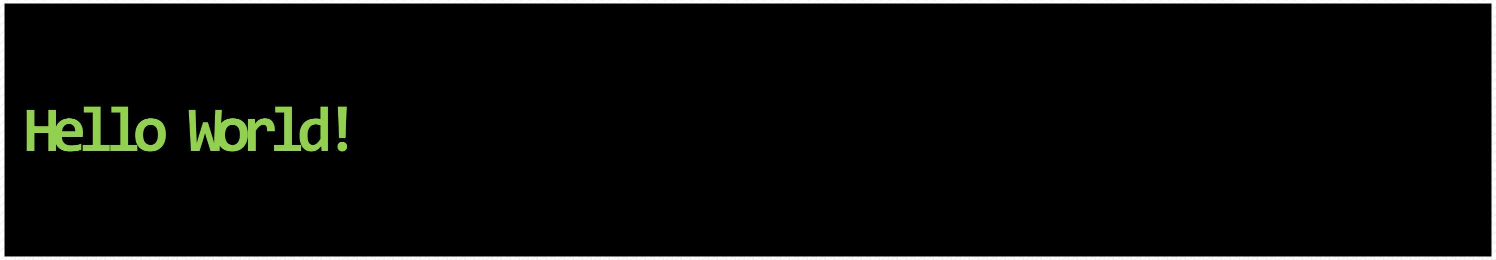
Step 2

- Get the only object from the singleton class.
- *SingletonPatternDemo.java*

```
public class SingletonPatternDemo {
   public static void main(String[] args) {
      //illegal construct
      //Compile Time Error: The constructor SingleObject() is
       not visible
      //SingleObject object = new SingleObject();
      //Get the only object available
      SingleObject object = SingleObject.getInstance();
      //show the message
      object.showMessage();
   }
}
```

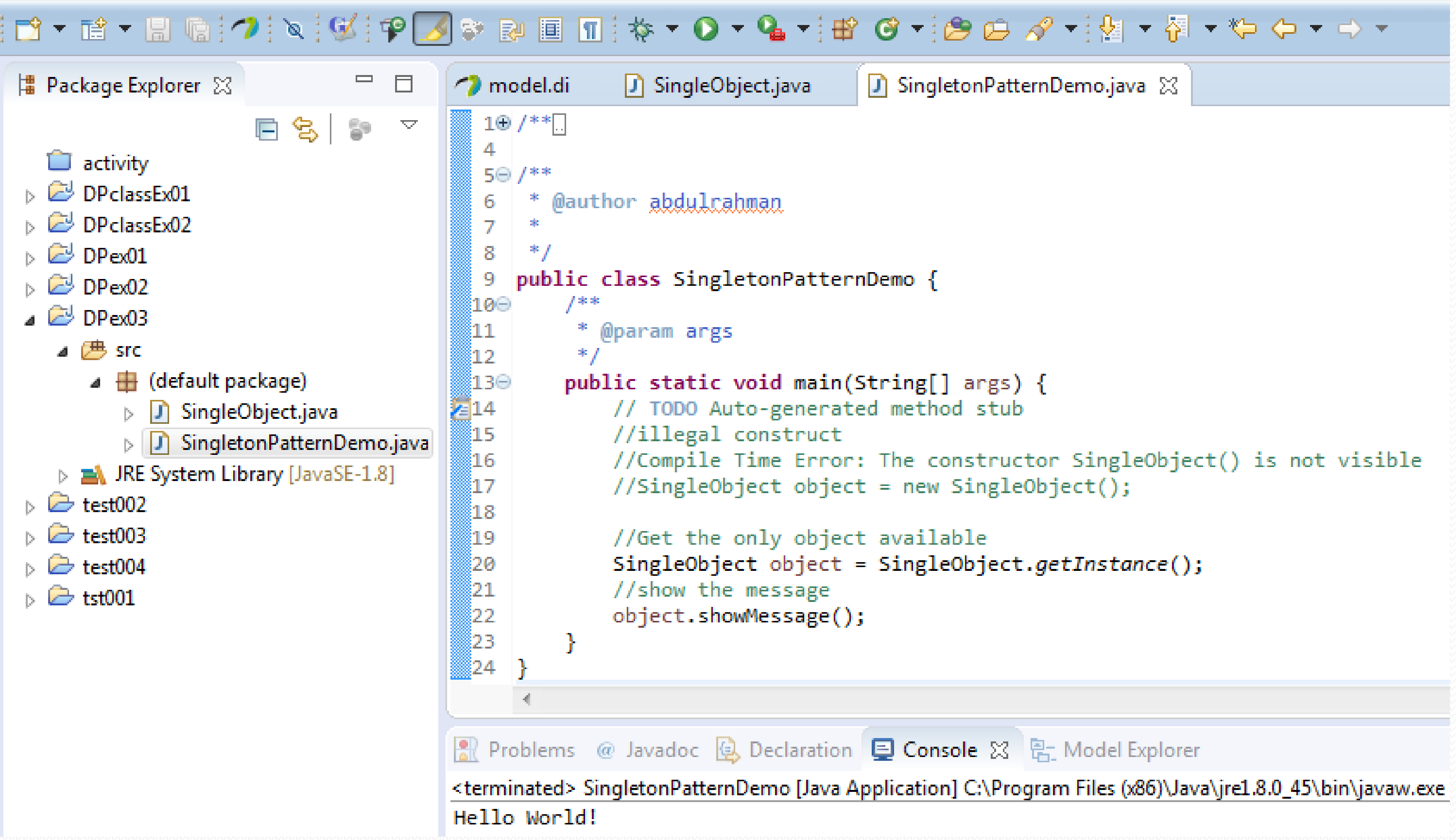
Step 3

- Verify the output.



```
Hello World!
```

Eclipse workspace



```
/**

/**
 * @author abdulrahman
 *
 */
public class SingletonPatternDemo {
	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		//illegal construct
		//Compile Time Error: The constructor SingleObject() is not visible
		//SingleObject object = new SingleObject();

		//Get the only object available
		SingleObject object = SingleObject.getInstance();
		//show the message
		object.showMessage();
	}
}
```

```
<terminated> SingletonPatternDemo [Java Application] C:\Program Files (x86)\Java\jre1.8.0_45\bin\javaw.exe
Hello World!
```

Factory Design Pattern

Factory Pattern

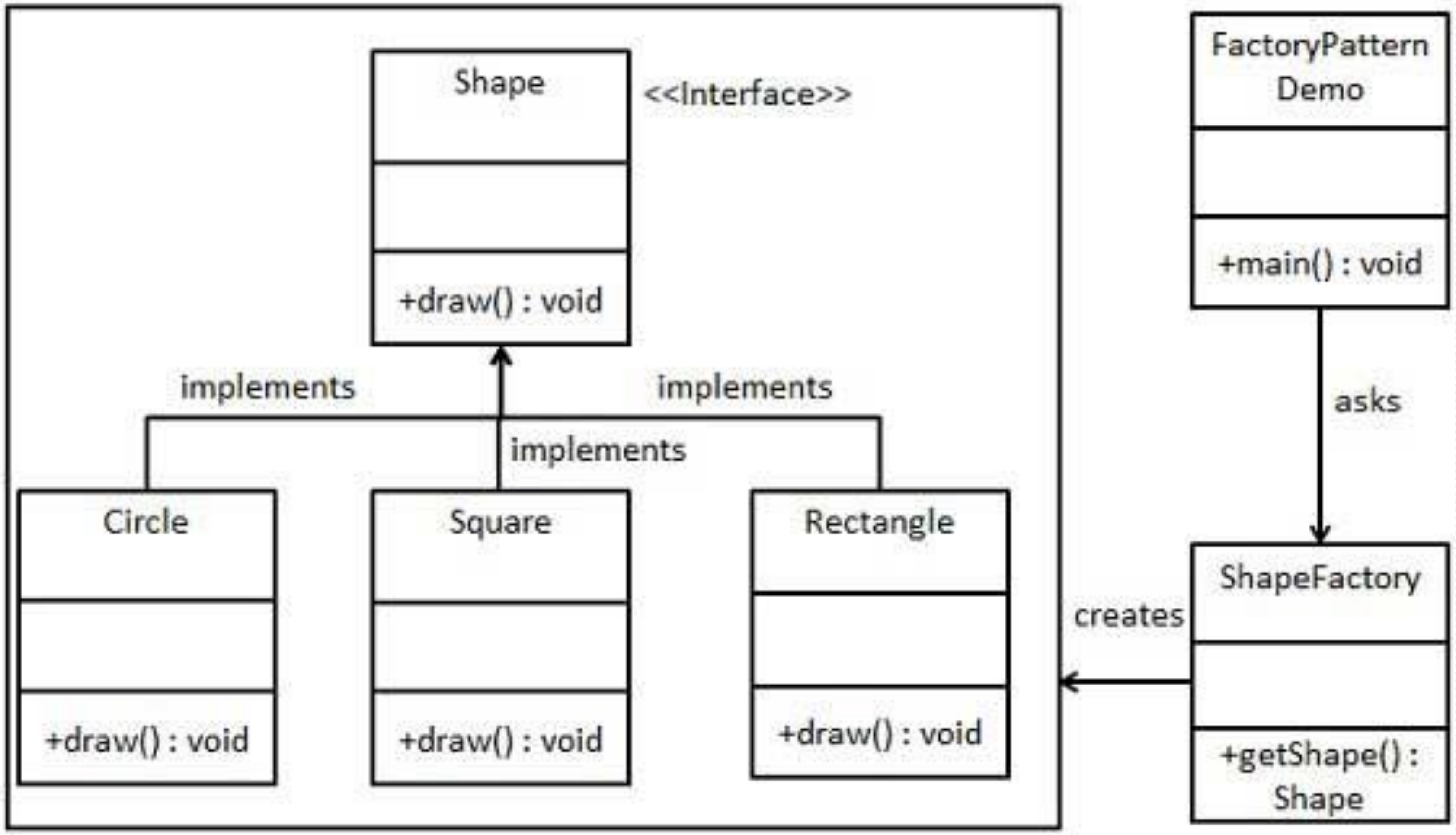
- Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

- In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

Implementation

- We're going to create a *Shape* interface and concrete classes implementing the *Shape* interface. A factory class *ShapeFactory* is defined as a next step.
- *FactoryPatternDemo*, our demo class will use *ShapeFactory* to get a *Shape* object. It will pass information (*CIRCLE* / *RECTANGLE* / *SQUARE*) to *ShapeFactory* to get the type of object it needs.

Implementation



Step 1

- Create an interface. (*Shape.java*)

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

Step 2

- Create concrete classes implementing the same interface.

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

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

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

Step 3

- Create a Factory to generate object of concrete class based on given information. (*ShapeFactory.java)*

```
public class ShapeFactory {
   //use getShape method to get object of type shape
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();
      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;
   } }
```

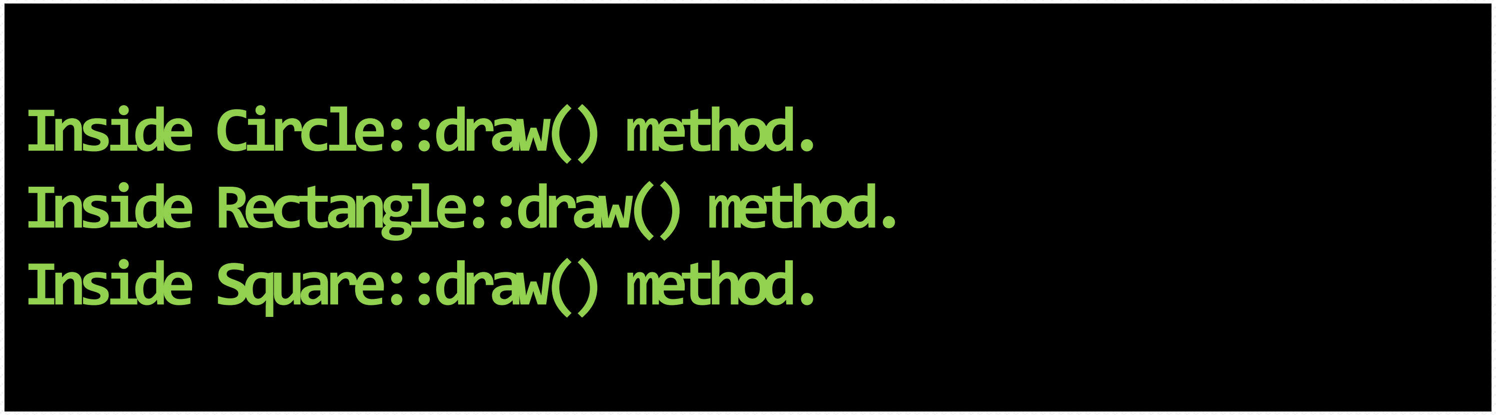
Step 4

- Use Factory to get object of concrete class by passing information such as type. (*FactoryPatternDemo.java)*

```
public class FactoryPatternDemo {
   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();
       Shape shape1 = shapeFactory.getShape("CIRCLE");
      shape1.draw();
       Shape shape2 = shapeFactory.getShape("RECTANGLE");
      shape2.draw();
       Shape shape3 = shapeFactory.getShape("SQUARE");
       shape3.draw();
   }
}
```

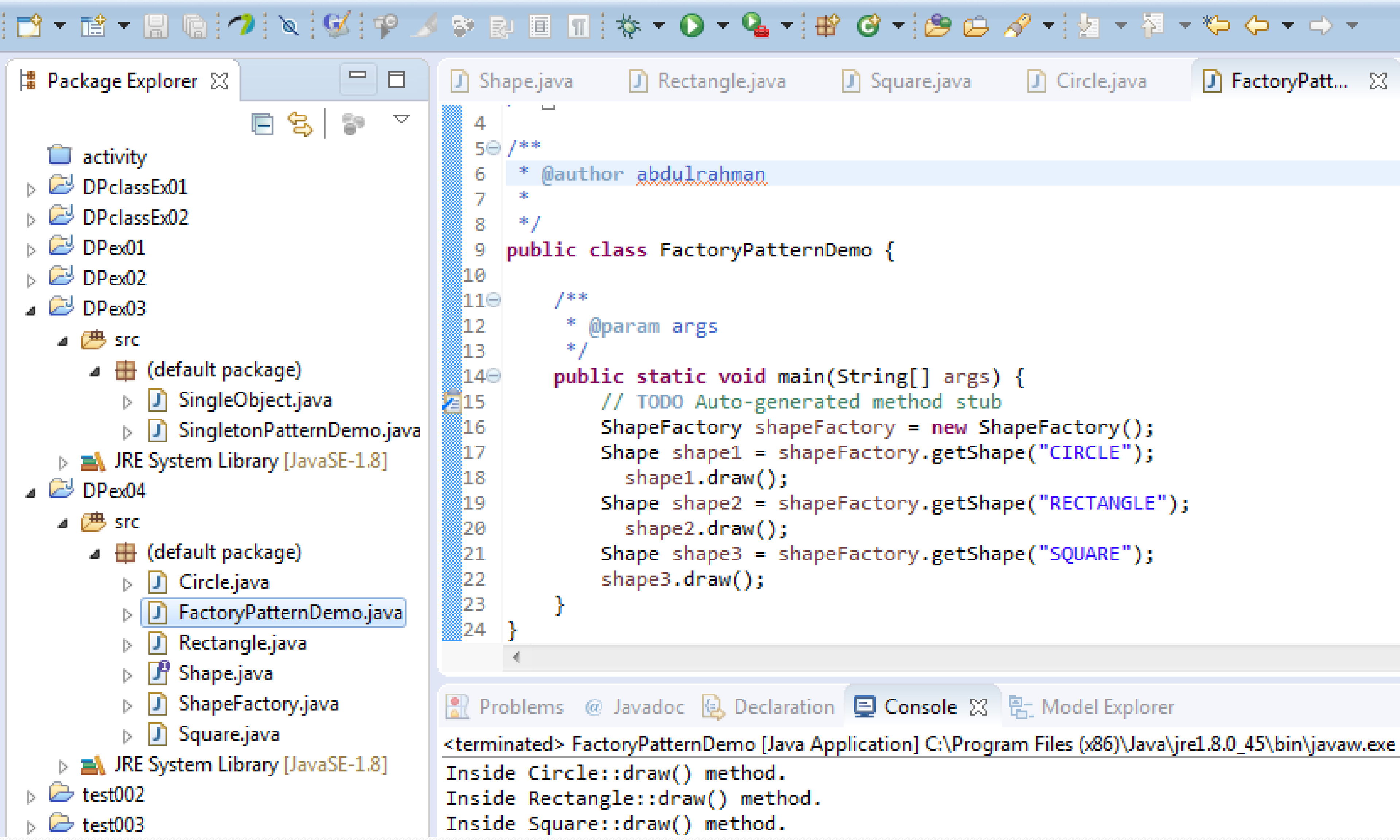
Step 5

- Verify the output.



```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

Eclipse workspace



```java
/**
 * @author abdulrahman
 *
 */
public class FactoryPatternDemo {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ShapeFactory shapeFactory = new ShapeFactory();
        Shape shape1 = shapeFactory.getShape("CIRCLE");
          shape1.draw();
        Shape shape2 = shapeFactory.getShape("RECTANGLE");
          shape2.draw();
        Shape shape3 = shapeFactory.getShape("SQUARE");
        shape3.draw();
    }
}
```

```
<terminated> FactoryPatternDemo [Java Application] C:\Program Files (x86)\Java\jre1.8.0_45\bin\javaw.exe
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```