Software Design and Architecture

Engr. Abdul-Rahman Mahmood

DPM, MCP, QMR(ISO9001:2000)

armahmood786@yahoo.com	alphasecure@gmail.com
alphapeeler.sf.net/pubkeys/pkey.htm	http://alphapeeler.sourceforge.net
pk.linkedin.com/in/armahmood	http://alphapeeler.tumblr.com
www.twitter.com/alphapeeler	armahmood786@jabber.org
www.facebook.com/alphapeeler	alphapeeler@aim.com
abdulmahmood-sss alphasecure	mahmood_cubix 48660186
armahmood786@hotmail.com	alphapeeler@icloud.com
http://alphapeeler.sf.net/me	http://alphapeeler.sf.net/acms/

Single 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

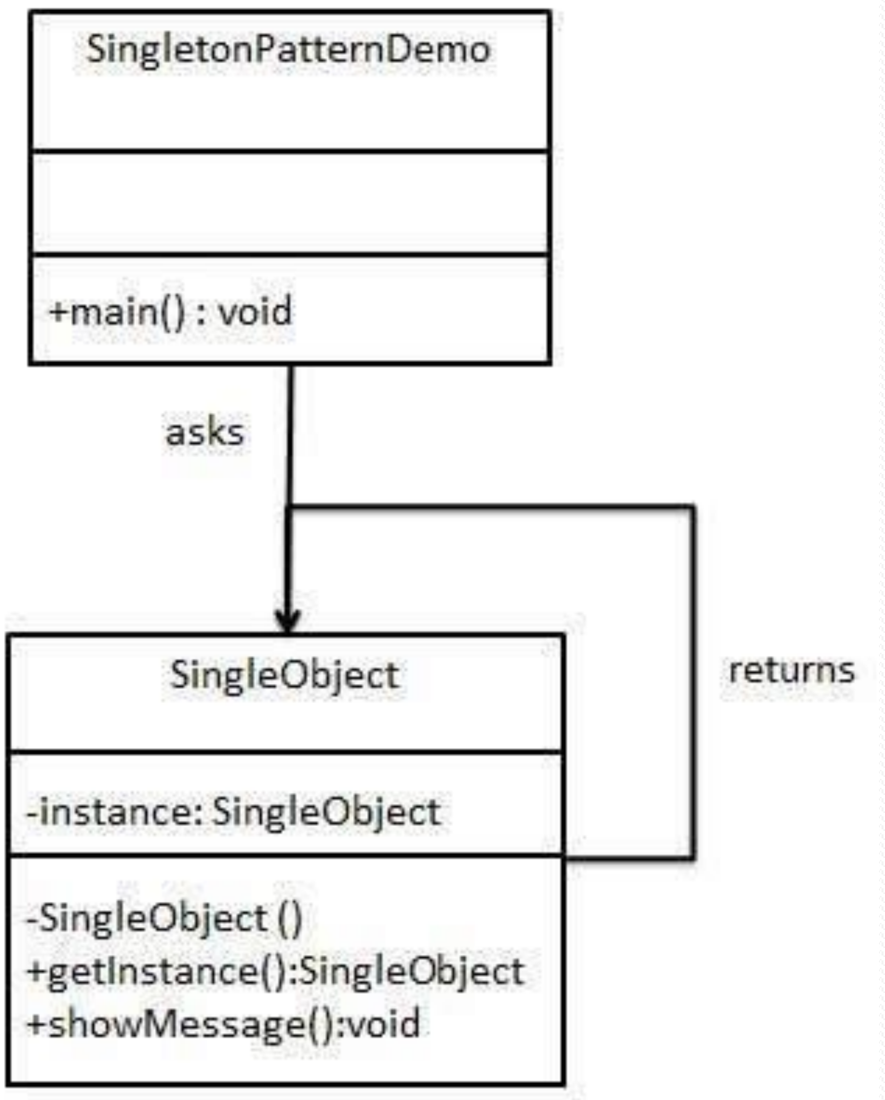
- This pattern involves a **single class** which is responsible to create an object while making sure that **only single object gets created**.

- 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!");
   }
}
```

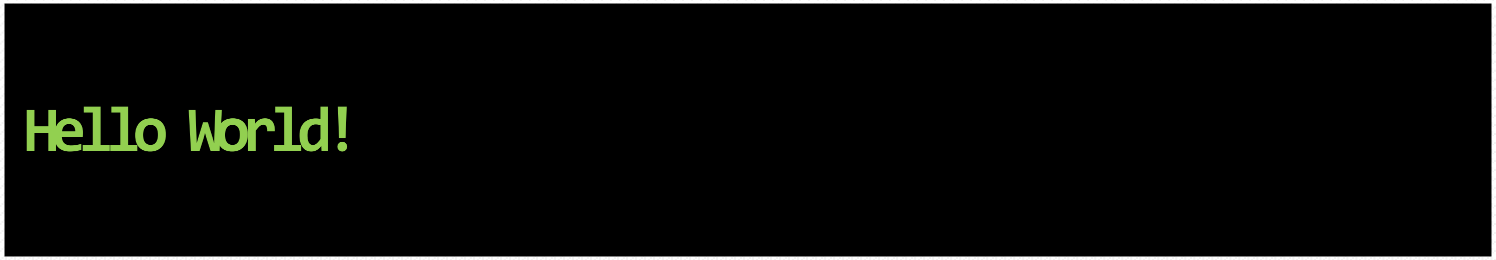
Step 2

- 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();
   }
}
```

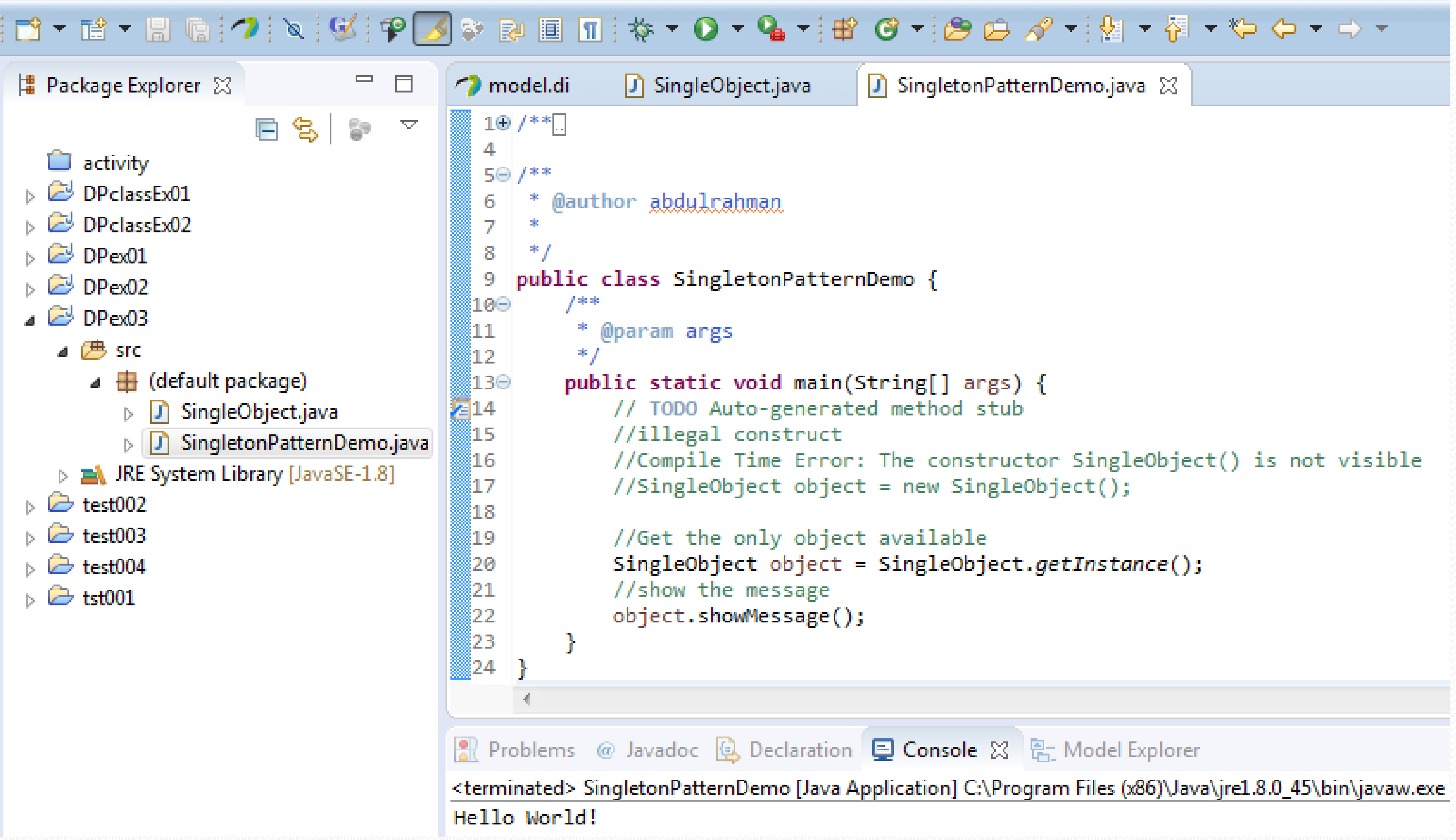
Step 3

- 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!
```

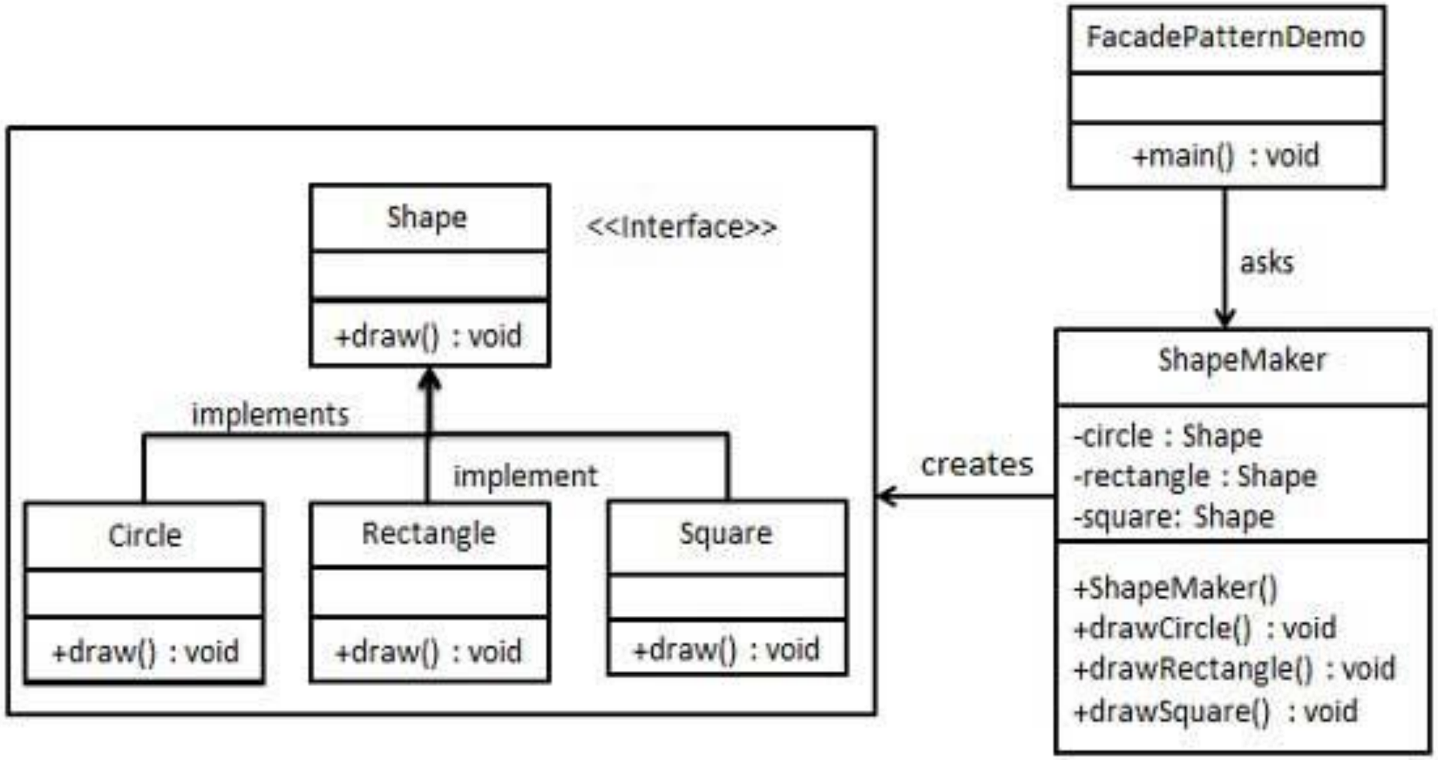
Façade Design Pattern

Façade Design Pattern

- Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system.
- This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities.
- This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.

Implementation

- We are going to create a *Shape* interface and concrete classes implementing the *Shape* interface. A facade class *ShapeMaker* is defined as a next step.
- *ShapeMaker* class uses the concrete classes to delegate user calls to these classes. *FacadePatternDemo*, our demo class, will use *ShapeMaker* class to show the results.



Step 1

- Create an interface.
- *Shape.java*

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

Step 2

- Create concrete classes implementing same interface.

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

Rectangle.java

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

Square.java

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

Circle.java

Step 3

- Create a facade class.
- *ShapeMaker.java*

```
public class ShapeMaker {
   private Shape circle;
   private Shape rectangle;
   private Shape square;

   public ShapeMaker() {
      circle = new Circle();
      rectangle = new Rectangle();
      square = new Square();
   }

   public void drawCircle(){
      circle.draw();
   }
   public void drawRectangle(){
      rectangle.draw();
   }
   public void drawSquare(){
      square.draw();
   }
}
```

Step 4

- Use the facade to draw various types of shapes.
- *FacadePatternDemo.java*

```
public class FacadePatternDemo {
   public static void main(String[] args) {
      ShapeMaker shapeMaker = new ShapeMaker();

      shapeMaker.drawCircle();
      shapeMaker.drawRectangle();
      shapeMaker.drawSquare();
   }
}
```

Step 5

- Verify the output.

```
Circle::draw()
Rectangle::draw()
Square::draw()
```