Design Patterns
(Singleton, factory, abstract factory)

B.Tech. (IT), Sem-6,
Applied Design Patterns and Application Frameworks (ADPAF)

Dharmsinh Desai University
Prof. H B Prajapati

The Singleton Design Pattern

- It is creational design pattern.
- There are many situations in which you want to make sure that only one instance is present for a particular class.
- The pattern implementation provides a single point of access to the class.
- This pattern is a creational design pattern, which means it controls instantiation of the object.
- It has a private constructor and a static method to get the singleton object.
- the singleton design pattern offers two things: one and only one instance of the class, and a global single point of access to that object.

Example: The Singleton Design Pattern

```
public class Logger {
    // declare the constructor private to prevent clients from instantiating an object
       of this class directly
    private Logger() { }
    public static Logger myInstance; // by default, this field is initialized to null
    // the static method to be used by clients to get the instance of the Logger class
    public static Logger getInstance() {
        if(myInstance == null) {
            // this is the first time this method is called, and that's why myInstance
               is null
            myInstance = new Logger();
        }
```


Example: The Singleton Design Pattern

```
    // return the same object reference any time and every time
       getInstance is called
    return myInstance;
    }

    public void log(String s) {
        // a trivial implementation of log where we pass the
           string to be logged to console
        System.err.println(s);
    }
}
```


Solution using “initialization on demand holder” idiom

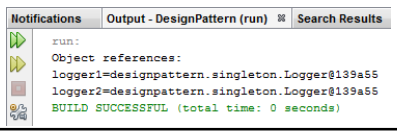
- This idiom uses inner classes and does not use any synchronization construct.
- It exploits the fact that inner classes are not loaded until they are referenced.

```
public class Logger {
    private Logger() {
        // private constructor
    }
    public static Logger
        myInstance;
```

```
public static class LoggerHolder {
    public static Logger logger =
    new Logger();
}
public static Logger getInstance(){
    return LoggerHolder.logger;
}
public void log(String s) {
    // log implementation
    System.err.println(s);
}
}
```


Output of Singleton example

```
public class Test {
  public static void main(String[] args) {
    Logger logger1=Logger.getInstance();
    Logger logger2=Logger.getInstance();
    System.out.println("Object
   references:\nlogger1="+logger1+"\nlogger2="+logger2);
  }
}
```



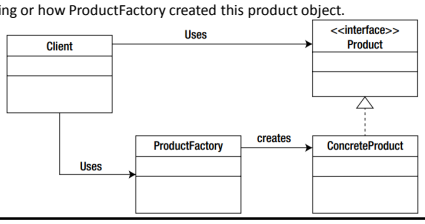
Factory Design Pattern

- It is creational design pattern.
- In real life, factories are manufacturing units that produce multiple instances of a product .
- The main responsibility of the factory is to keep producing cars of the required type and model.
- One important thing to observe is that a car may have different variants and the car factory should be able to manufacture on demand the required variants of the same car.
- Similarly, we can implement a factory that returns the required type of object(s) on demand in OOP.
- The factory decides which class(es) to instantiate to create the required object(s) and exactly how to create them.

7

Factory Design Pattern

- Client invokes ProductFactory to get an appropriate object from the product hierarchy.
- ProductFactory creates one of the Products from the Product hierarchy based on the provided information.
- Client uses the product object without knowing which actual product it is using or how ProductFactory created this product object.



8

Example: Factory Design Pattern

```
// Shape.java
public interface Shape {
    public void draw();
    public void fillColor();
}
// Circle.java
public class Circle implements Shape {
    private int xPos, yPos;
    private int radius;
    public Circle(int x, int y, int r) {
        xPos = x;
        yPos = y;
        radius = r;
        System.out.println("Circle constructor");
    }
    @Override
    public void draw() {
        System.out.println("Circle draw()");
        // draw() implementation
    }
    @Override
    public void fillColor() {
        // fillColor() implementation
    }
}
```

10

Example: Factory Design Pattern

- There are different types of shapes, such as Circle and Rectangle.
- The Canvas object might not want to know about the concrete shape that gets created.
- The Canvas object receives a shape identifier (command) from the front end based on which corresponding shape object needs to be created.
- We can use a ShapeFactory that can create the required shape object and return it to the Canvas object.

9

Example: Factory Design Pattern (contd.)

```
// Rectangle.java
public class Rectangle implements Shape
    {
    private int length, height;

    public Rectangle(int length, int
        height) {
        this.length = length;
        this.height = height;
        System.out.println("Rectangle
            constructor");
    }
    @Override
    public void draw() {
        System.out.println("Rectangle
            draw()");
        // draw() implementation
    }
    @Override
    public void fillColor() {
        // fillColor() implementation
    }
}
```

11

Example: Factory Design Pattern (contd.)

```
// ShapeFactory.java
public class ShapeFactory {
    public static Shape getShape(String sourceType) {
        switch(sourceType) {
            case "Circle":
                return new Circle(10, 10, 20);
            case "Rectangle":
                return new Rectangle(10, 20);
        }
        return null;
    }
}
```

12

Example: Factory Design Pattern (contd.)

```
public class Canvas {
    private ArrayList<Shape> shapeList = new ArrayList<Shape>();
    public void addNewShape(String shapeType) {
        Shape shape = ShapeFactory.getShape(shapeType);
        shapeList.add(shape);
    }
    public void redraw() {
        Iterator<Shape> itr = shapeList.iterator();
        while(itr.hasNext()) {
            Shape shape = itr.next();
            shape.draw();
        }
    }
}
```

13

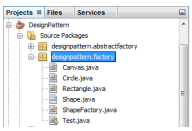
Example: Factory Design Pattern (contd.)

```
// Test.java
public class Test {
    public static void main(String[] args) {
    Canvas canvas = new Canvas();
        canvas.addNewShape("Circle");
        canvas.addNewShape("Rectangle");
        canvas.redraw();
    }
}
```

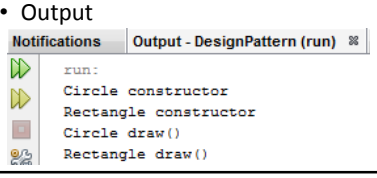
14

Running the Example: Factory Design Pattern

- Classes



- Output



15

Example: Factory Design Pattern (contd.)

- The Canvas class does not need to know how to create concrete shape objects.
- This transparency becomes very useful in case concrete object creation is expensive and complicated.
- The Canvas class does not need to know the exact concrete shape types.
- You can observe from the Canvas implementation that Canvas is only aware of the Shape interface.
- Therefore, if you add another concrete shape (say Square), you do not need to change the Canvas implementation.

16

Calendar uses Factory Design Pattern

```
import java.util.Calendar;
public class MainClass {
    public static void main(String[] args) {
        Calendar calendar = Calendar.getInstance();
        System.out.println(calendar);
    }
}
```

- This program prints the following:

java.util.GregorianCalendar [...]

- There are many other examples in Java SDK where the factory pattern is used, including the following:

createStatement() of java.sql.Connection interface, which creates a new Statement to communicate to database.

17

Factory and Abstract Factory Design Patterns

- It is creational design pattern.
- Both factory design patterns and abstract factory design patterns belong to the creational design pattern category.
- The abstract factory is basically a factory of factories.
- The abstract factory design pattern introduces one more indirection to create a specified object.
- A client of the abstract factory design pattern
  - First requests a proper factory from the abstract factory object, and
  - Then it requests an appropriate object from the factory object.

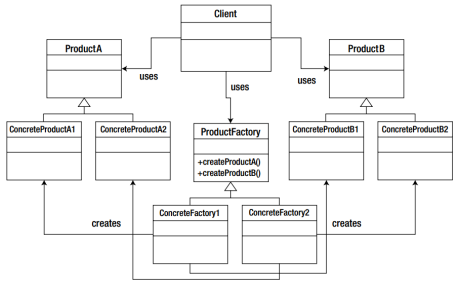
18

Factory and Abstract Factory Design Patterns

- When to use which?
  - When you have only one type of object to be created, you can use a factory design pattern;
  - When you have a family of objects to be created, you can use an abstract factory design pattern.

19

Class diagram: abstract factory



20

Factory and Abstract Factory Design Patterns

- There are two product hierarchies, ProductA and ProductB, along with their concrete product classes
  - (Concrete ProductA1, Concrete ProductA2, Concrete ProductB1, Concrete ProductB2).
- We want to create either
  - ConcreteProductA1 and ConcreteProductB1(as a group) or
  - ConcreteProductA2 and ConcreteProductB2.
- We define an abstract ProductFactory with two subclasses, ProductFactory1and ProductFactory2.
  - ProductFactory1 creates ConcreteProductA1and ConcreteProductB1
  - ProductFactory2creates ConcreteProductA2and ConcreteProductB2.

21

Example: Factory and Abstract Factory Design Patterns

- Let's assume that shapes can be of two types:
  - DisplayFriendly
  - PrinterFriendly
- Two flavors available for Circle class (as well as for your Rectangle class):
  - DisplayFriendlyCircle and
  - PrinterFriendlyCircle.
- We want to create only one type of objects: either display-friendly or printer-friendly.

22

Example: Factory and Abstract Factory Design Patterns

```
// Shape.java
public interface Shape {
    public void draw();
}

// PrinterFriendlyShape.java
public interface PrinterFriendlyShape extends Shape {
}

// DisplayFriendlyShape.java
public interface DisplayFriendlyShape extends Shape {
}
```

23

Example: Factory and Abstract Factory Design Patterns

```
// DisplayFriendlyCircle.java
public class DisplayFriendlyCircle implements
    DisplayFriendlyShape {
    private int xPos, yPos;
    private int radius;
    public DisplayFriendlyCircle(int x, int y, int r) {
        xPos = x;
        yPos = y;
        radius = r;
        System.out.println("DisplayFriendlyCircle constructor");
    }
```

24

Example: Factory and Abstract Factory Design Patterns

```
    @Override
    public void draw() {
        System.out.println("DisplayFriendlyCircle
          draw()");
        // draw() implementation
    }
}
```

25

Example: Factory and Abstract Factory Design Patterns

```
// DisplayFriendlyRectangle.java
public class DisplayFriendlyRectangle implements DisplayFriendlyShape {
    public DisplayFriendlyRectangle(int length, int height) {
        this.length = length;
        this.height = height;
        System.out.println("DisplayFriendlyRectangle constructor");
    }
    private int length, height;
    @Override
    public void draw() {
        System.out.println("DisplayFriendlyRectangle draw()");
        // draw() implementation
    }
}
```

26

Example: Factory and Abstract Factory Design Patterns

```
// PrinterFriendlyCircle.java
public class PrinterFriendlyCircle implements
   PrinterFriendlyShape{
    private int xPos, yPos;
    private int radius;
    public PrinterFriendlyCircle(int x, int y, int r) {
        xPos = x;
        yPos = y;
        radius = r;
        System.out.println("PrinterFriendlyCircle constructor");
    }
```

27

Example: Factory and Abstract Factory Design Patterns

```
    @Override
    public void draw() {
        System.out.println("PrinterFriendlyCircle draw()");
        // draw() implementation
    }
}
```

28

Example: Factory and Abstract Factory Design Patterns

```
// PrinterFriendlyRectangle.java
public class PrinterFriendlyRectangle implements
   PrinterFriendlyShape {
    public PrinterFriendlyRectangle(int length, int height) {
        this.length = length;
        this.height = height;
        System.out.println("PrinterFriendlyRectangle
          constructor");
    }
    private int length, height;
```

29

Example: Factory and Abstract Factory Design Patterns

```
    @Override
    public void draw() {
        System.out.println("PrinterFriendlyRectangle draw()");
        // draw() implementation
    }
}
```

30

Example: Factory and Abstract Factory Design Patterns

```
// ShapeFactory.java
public interface ShapeFactory {
    public Shape getShape(String sourceType);
}
```

31

Example: Factory and Abstract Factory Design Patterns

```
// DisplayFriendlyFactory.java
public class DisplayFriendlyFactory implements ShapeFactory {
    @Override
    public Shape getShape(String sourceType) {
        switch(sourceType){
            case "Circle":
                return new DisplayFriendlyCircle(10, 10, 20);
            case "Rectangle":
                return new DisplayFriendlyRectangle(10, 20);
        }
        return null;
    }
}
```

32

Example: Factory and Abstract Factory Design Patterns

```
// PrinterFriendlyFactory.java
public class PrinterFriendlyFactory implements ShapeFactory {
    @Override
    public Shape getShape(String sourceType) {
        switch(sourceType) {
            case "Circle":
                return new PrinterFriendlyCircle(10, 10, 20);
            case "Rectangle":
                return new PrinterFriendlyRectangle(10, 20);
        }
        return null;
    }
}
```

33

Example: Factory and Abstract Factory Design Patterns

```
public class Canvas {
  private ArrayList<Shape> shapeList = new ArrayList<Shape>();
  public void addNewShape(String shapeType, String objectType) {
    Shape shape=null;
    switch(objectType) {
      case "DisplayFriendly":
        shape=new DisplayFriendlyFactory().getShape(shapeType);
        break;
      case "PrinterFriendly":
        shape=new PrinterFriendlyFactory().getShape(shapeType);
        break;
    }
    shapeList.add(shape);
  }
```

34

Example: Factory and Abstract Factory Design Patterns

```
public void redraw() {
    Iterator<Shape> itr = shapeList.iterator();
    while(itr.hasNext()) {
      Shape shape = itr.next();
      shape.draw();
    }
  }
}
```

35

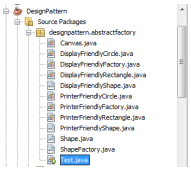
Example: Factory and Abstract Factory Design Patterns

```
// Test.java
public class Test {
    public static void main(String[] args) {
        Canvas canvas = new Canvas();
        canvas.addNewShape("Circle", "DisplayFriendly");
        canvas.addNewShape("Rectangle", "DisplayFriendly");
        canvas.redraw();
    }
}
```

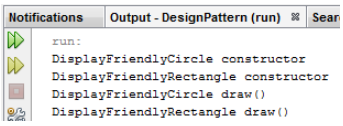
36

Running the Example: Factory and Abstract Factory Design Patterns

- Classes



- Output



37

References

- Oracle Certified Professional Java SE 7 Programmer Exams 1Z0-804 and 1Z0-804 (A Comprehension OCPJP 7 Certification Guide), by S G Ganesh and Tushar Sharma, publisher Apress
- Java Design Patterns, problem solving approaches, tutorials point, www.tutorialspoint.com

38