**Design Patterns**

Design patterns are programming language independent strategies for solving a common problem. That means a design pattern represents an idea, not a particular implementation. By using design patterns, you can make your code more flexible, reusable, and maintainable.

In many real-world situations, we want to create only one instance of a class. For example, there can be only one active president of a country at any given time. This pattern is called a Singleton pattern. Other software examples could be a single DB connection shared by multiple objects as creating a separate DB connection for every object is costly. Similarly, there can be a single configuration manager or error manager in an application that handles all problems instead of creating multiple managers.

**What Are Design Patterns?**

We have been building object-oriented software for over 40 years now, starting with Smalltalk, which was the first object-oriented language.

The programming world has encountered a large number of problems, and a variety of solution have been proposed to tackle them.

An attempt was made by a group of four people, famously called the “Gang-Of-Four” or GoF, to come up with a set of common problems and solutions for them, in the given context.

This catalog of common problems and their solutions is labeled as GOF (Gang of Four) Design Patterns.

**Why Design Patterns?**

The advantages of design patterns are:

* To provide standard terminology that everybody understands
* Not to repeat the same mistakes over and over again

**What Are the Types of Design Patterns?**

The design patterns we talk about here, are from the perspective of an object-oriented world. There are mainly three different kinds of design patterns:

* Creational Patterns
* Structural Patterns
* Behavioral Patterns

**Creational**

* Creational patterns deal with the creation of objects.

**Structural**

* Structural patterns deal with the composition of objects.

It deals with questions such as:

* What does a class contain?
* What are the relationships of a class with other classes? Is it inheritance or composition?

**Behavioral**

* Behavioral patterns focus more on the behavior of objects, or more precisely, interactions between objects.

How does an object communicate with another object?

**List of the Original 23 Patterns**

|  |  |  |
| --- | --- | --- |
| Purpose | Design Pattern | Aspect(s) that can vary |
| Creational | Abstract Factory | families of product objects |
| Builder | how a composite object gets created |
| Factory Method | subclass of object that is instantiated |
| Prototype | class of object that is instantiated |
| Singleton | the sole instance of a class |
| Structural | Adapter | interface to an object |
| Bridge | implementation of an object |
| Composite | structure and composition of an object |
| Decorator | responsibilities of an object without subclassing |
| Facade | interface to a subsystem |
| Flyweight | storage costs of objects |
| Proxy | how an object is accessed; its location |
| Behavioral | Chain of Responsibility | object that can fulfill a request |
| Command | when and how a request is fulfilled |
| Interpreter | grammar and interpretation of a language |
| Iterator | how an aggregate's elements are accessed, traversed |
| Mediator | how and which objects interact with each other |
| Memento | what private information is stored outside an object, and when |
| Observer | number of objects that depend on another object; how the dependent objects stay up to date |
| State | states of an object |
| Strategy | an algorithm |
| Template Method | steps of an algorithm |
| Visitor | operations that can be applied to object(s) without changing their class(es) |

**Singleton pattern**

* one of the simplest design patterns in Java.
* This pattern involves a single class which is responsible to create an object while making sure that only single object gets created.
* ensure that a class has only one instance, while providing a global access point to this instance.

Follow the instruction to create Singleton class:

* **Static member:** It gets memory only once because of static, it contains the instance of the Singleton class.
* **Private constructor:** It will prevent to instantiate the Singleton class from outside the class.
* **Static factory method:** This provides the global point of access to the Singleton object and returns the instance to the caller.

**Initialization Types of Singleton**

Singleton class can be instantiated by two methods:

1. **Early initialization :** In this method, class is initialized whether it is to be used or not. The main advantage of this method is its simplicity. You initiate the class at the time of class loading. Its drawback is that class is always initialized whether it is being used or not.
2. **Lazy initialization :** In this method, class in initialized only when it is required. It can save you from instantiating the class when you don’t need it. Generally, lazy initialization is used when we create a singleton class.

### Understanding early Instantiation of Singleton Pattern

In such case, we create the instance of the class at the time of declaring the static data member, so instance of the class is created at the time of class loading.

**class** Singleton

{

// Early, instance will be created at load time

**private** **static** Singleton obj = new Singleton ();

     // private constructor to force use of

    // getInstance() to create Singleton object

**private** Singleton() {}

**public** **static** Singleton getInstance()

    {

**return** obj;

    }

}

### Understanding lazy Instantiation of Singleton Pattern

In such case, we create the instance of the class in synchronized method or synchronized block, so instance of the class is created when required.

**class** Singleton

{

**private** **static** Singleton obj;

    // private constructor to force use of

    // getInstance() to create Singleton object

**private** Singleton() {}

**public** **static** Singleton getInstance()

    {

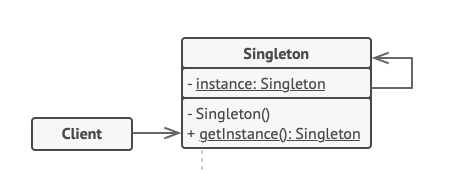
**if** (obj==**null**)

            obj = **new** Singleton();

**return** obj;

    }

}



The **Singleton** class declares the static method getInstance that returns the same instance of its own class.

The Singleton’s constructor should be hidden from the client code. Calling the getInstance method should be the only way of getting the Singleton object.

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!");

}

}

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

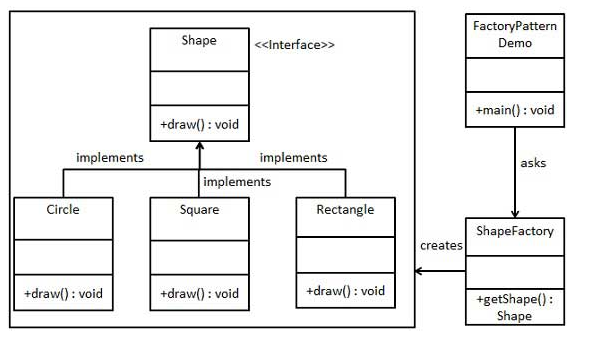
object.showMessage();

}

}

# **Factory Pattern**

* In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.
* just **define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.** In other words, subclasses are responsible to create the instance of the class.



Create an interface.

public interface Shape

{

void draw();

}

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.");

}

}

Create a Factory to generate object of concrete class based on given information.

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;

}

}

Use the Factory to get object of concrete class by passing an information such as type.

public class FactoryPatternDemo

{

public static void main(String[] args)

{

ShapeFactory shapeFactory = new ShapeFactory();

//get an object of Circle and call its draw method.

Shape shape1 = shapeFactory.getShape("CIRCLE");

//call draw method of Circle

shape1.draw();

//get an object of Rectangle and call its draw method.

Shape shape2 = shapeFactory.getShape("RECTANGLE");

//call draw method of Rectangle

shape2.draw();

//get an object of Square and call its draw method.

Shape shape3 = shapeFactory.getShape("SQUARE");

//call draw method of square

shape3.draw();

}

}

**Example 2:**

### Calculate Electricity Bill : A Real World Example of Factory Method

**Step 1:**Create a Plan abstract class.

|  |
| --- |
| **import** java.io.\*;  **abstract** **class** Plan  {  **protected** **double** rate;  **abstract** **void** getRate();    **public** **void** calculateBill(**int** units)  {                System.out.println(units\*rate);            }  }//end of Plan class. |

**Step 2:**Create the concrete classes that extends Plan abstract class.

|  |
| --- |
| **class**  DomesticPlan **extends** Plan  {          //@override  **public** **void** getRate()  {               rate=3.50;          }     }//end of DomesticPlan class.  **class**  CommercialPlan **extends** Plan  {     //@override  **public** **void** getRate()  {          rate=7.50;     }  }//end of CommercialPlan **class**.  **class**  InstitutionalPlan **extends** Plan  {     //@override  **public** **void** getRate()  {          rate=5.50;     }  }//end of InstitutionalPlan **class**. |

**Step 3:**Create a GetPlanFactory to generate object of concrete classes based on given information..

|  |
| --- |
| **class** GetPlanFactory  {       //use getPlan method to get object of type Plan  **public** Plan getPlan(String planType)  {  **if**(planType == **null**)  {  **return** **null**;              }  **if**(planType.equalsIgnoreCase("DOMESTICPLAN"))   {  **return** **new** DomesticPlan();                 }  **else** **if**(planType.equalsIgnoreCase("COMMERCIALPLAN"))  {  **return** **new** CommercialPlan();              }  **else** **if**(planType.equalsIgnoreCase("INSTITUTIONALPLAN"))   {  **return** **new** InstitutionalPlan();            }  **return** **null**;     }  }//end of GetPlanFactory class. |

**Step 4:**Generate Bill by using the GetPlanFactory to get the object of concrete classes by passing an information such as type of plan DOMESTICPLAN or COMMERCIALPLAN or INSTITUTIONALPLAN.

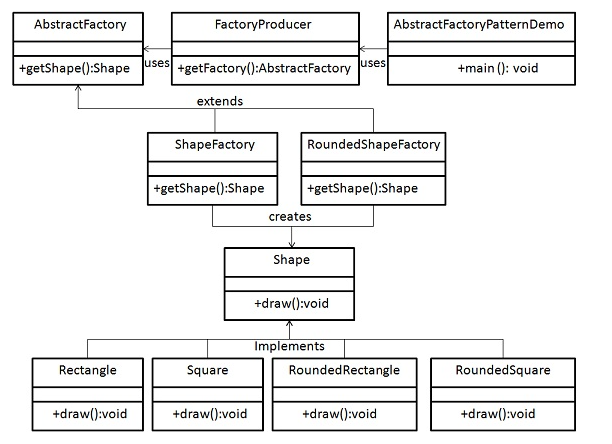
|  |
| --- |
| **import** java.io.\*;  **class** GenerateBill  {  **public** **static** **void** main(String args[])**throws** IOException  {        GetPlanFactory planFactory = **new** GetPlanFactory();          System.out.print("Enter the name of plan for which the bill will be generated: ");        BufferedReader br=**new** BufferedReader(**new** InputStreamReader(System.in));          String planName=br.readLine();        System.out.print("Enter the number of units for bill will be calculated: ");  **int** units=Integer.parseInt(br.readLine());          Plan p = planFactory.getPlan(planName);        //call getRate() method and calculateBill()method of DomesticPaln.         System.out.print("Bill amount for "+planName+" of  "+units+" units is: ");             p.getRate();             p.calculateBill(units);         }      }//end of GenerateBill class. |

# **Abstract Factory Pattern**

* It work around a super-factory which creates other factories.
* This factory is also called as factory of factories.
* This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
* Abstract Factory Pattern says that just **define an interface or abstract class for creating families of related (or dependent) objects but without specifying their concrete sub-classes.**That means Abstract Factory lets a class returns a factory of classes. So, this is the reason that Abstract Factory Pattern is one level higher than the Factory Pattern.

#### **Advantage of Abstract Factory Pattern**

* Abstract Factory Pattern isolates the client code from concrete (implementation) classes.
* It eases the exchanging of object families.
* It promotes consistency among objects.



## Step 1

Create an interface for Shapes.

*Shape.java*

public interface Shape

{

void draw();

}

## Step 2

Create concrete classes implementing the same interface.

*RoundedRectangle.java*

public class RoundedRectangle implements Shape

{

@Override

public void draw()

{

System.out.println("Inside RoundedRectangle::draw() method.");

}

}

*RoundedSquare.java*

public class RoundedSquare implements Shape

{

@Override

public void draw()

{

System.out.println("Inside RoundedSquare::draw() method.");

}

}

*Rectangle.java*

public class Rectangle implements Shape

{

@Override

public void draw()

{

System.out.println("Inside Rectangle::draw() method.");

}

}

## Step 3

Create an Abstract class to get factories for Normal and Rounded Shape Objects.

*AbstractFactory.java*

public abstract class AbstractFactory

{

abstract Shape getShape(String shapeType) ;

}

## Step 4

Create Factory classes extending AbstractFactory to generate object of concrete class based on given information.

*ShapeFactory.java*

public class ShapeFactory extends AbstractFactory

{

@Override

public Shape getShape(String shapeType)

{

if(shapeType.equalsIgnoreCase("RECTANGLE"))

{

return new Rectangle();

}else if(shapeType.equalsIgnoreCase("SQUARE"))

{

return new Square();

}

return null;

}

}

*RoundedShapeFactory.java*

public class RoundedShapeFactory extends AbstractFactory

{

@Override

public Shape getShape(String shapeType)

{

if(shapeType.equalsIgnoreCase("RECTANGLE"))

{

return new RoundedRectangle();

}else if(shapeType.equalsIgnoreCase("SQUARE"))

{

return new RoundedSquare();

}

return null;

}

}

## Step 5

Create a Factory generator/producer class to get factories by passing an information such as Shape

*FactoryProducer.java*

public class FactoryProducer

{

public static AbstractFactory getFactory(boolean rounded)

{

if(rounded)

{

return new RoundedShapeFactory();

}else

{

return new ShapeFactory();

}

}

}

## Step 6

Use the FactoryProducer to get AbstractFactory in order to get factories of concrete classes by passing an information such as type.

*AbstractFactoryPatternDemo.java*

public class AbstractFactoryPatternDemo

{

public static void main(String[] args)

{

//get shape factory

AbstractFactory shapeFactory = FactoryProducer.getFactory(false);

//get an object of Shape Rectangle

Shape shape1 = shapeFactory.getShape("RECTANGLE");

//call draw method of Shape Rectangle

shape1.draw();

//get an object of Shape Square

Shape shape2 = shapeFactory.getShape("SQUARE");

//call draw method of Shape Square

shape2.draw();

//get shape factory

AbstractFactory shapeFactory1 = FactoryProducer.getFactory(true);

//get an object of Shape Rectangle

Shape shape3 = shapeFactory1.getShape("RECTANGLE");

//call draw method of Shape Rectangle

shape3.draw();

//get an object of Shape Square

Shape shape4 = shapeFactory1.getShape("SQUARE");

//call draw method of Shape Square

shape4.draw();

}

}