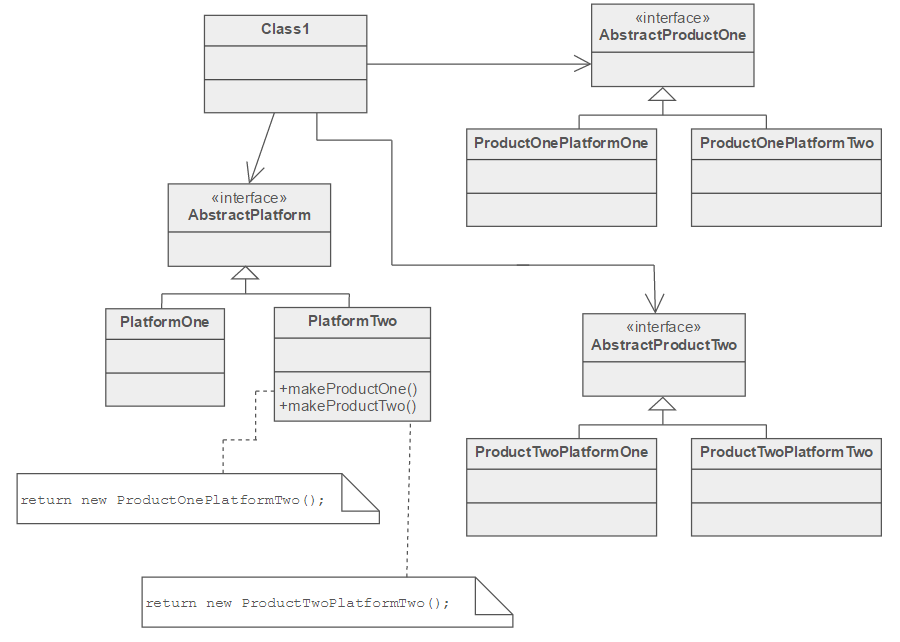
***Abstract Factory Pattern***

**Purpose**

* Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
* A hierarchy that encapsulates: many possible "platforms", and the construction of a suite of "products".
* The new operator considered harmful.

**Structure**

The Abstract Factory defines a Factory Method per product. Each Factory Method encapsulates the new operator and the concrete, platform-specific, product classes. Each "platform" is then modeled with a Factory derived class.



**Real Life Example**

The purpose of the Abstract Factory is to provide an interface for creating families of related objects, without specifying concrete classes. This pattern is found in the sheet metal stamping equipment used in the manufacture of Japanese automobiles. The stamping equipment is an Abstract Factory which creates auto body parts. The same machinery is used to stamp right hand doors, left hand doors, right front fenders, left front fenders, hoods, etc. for different models of cars. Through the use of rollers to change the stamping dies, the concrete classes produced by the machinery can be changed within three minutes.

**Relation between other patterns**

* Sometimes creational patterns are competitors: there are cases when either Prototype or Abstract Factory could be used profitably. At other times they are complementary: Abstract Factory might store a set of Prototypes from which to clone and return product objects, Builder can use one of the other patterns to implement which components get built. Abstract Factory, Builder, and Prototype can use Singleton in their implementation.
* Abstract Factory, Builder, and Prototype define a factory object that's responsible for knowing and creating the class of product objects, and make it a parameter of the system. Abstract Factory has the factory object producing objects of several classes. Builder has the factory object building a complex product incrementally using a correspondingly complex protocol. Prototype has the factory object (aka prototype) building a product by copying a prototype object.
* Abstract Factory classes are often implemented with Factory Methods, but they can also be implemented using Prototype.
* Abstract Factory can be used as an alternative to Facade to hide platform-specific classes.
* Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately.
* Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed.

**Coding Example**

Create an interface for Shapes.

*Shape.java*

public interface Shape {

void draw();

}

Create concrete classes implementing the same interface.

*Rectangle.java*

public class Rectangle implements Shape {

@Override

public void draw() {

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

}

}

*Square.java*

public class Square implements Shape {

@Override

public void draw() {

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

}

}

*Circle.java*

public class Circle implements Shape {

@Override

public void draw() {

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

}

}

Create an interface for Colors.

*Color.java*

public interface Color {

void fill();

}

Create concrete classes implementing the same interface.

*Red.java*

public class Red implements Color {

@Override

public void fill() {

System.out.println("Inside Red::fill() method.");

}

}

*Green.java*

public class Green implements Color {

@Override

public void fill() {

System.out.println("Inside Green::fill() method.");

}

}

*Blue.java*

public class Blue implements Color {

@Override

public void fill() {

System.out.println("Inside Blue::fill() method.");

}

}

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

*AbstractFactory.java*

public abstract class AbstractFactory {

abstract Color getColor(String color);

abstract Shape getShape(String shape) ;

}

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 == 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;

}

@Override

Color getColor(String color) {

return null;

}

}

*ColorFactory.java*

public class ColorFactory extends AbstractFactory {

@Override

public Shape getShape(String shapeType){

return null;

}

@Override

Color getColor(String color) {

if(color == null){

return null;

}

if(color.equalsIgnoreCase("RED")){

return new Red();

}else if(color.equalsIgnoreCase("GREEN")){

return new Green();

}else if(color.equalsIgnoreCase("BLUE")){

return new Blue();

}

return null;

}

}

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

*FactoryProducer.java*

public class FactoryProducer {

public static AbstractFactory getFactory(String choice){

if(choice.equalsIgnoreCase("SHAPE")){

return new ShapeFactory();

}else if(choice.equalsIgnoreCase("COLOR")){

return new ColorFactory();

}

return null;

}

}

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("SHAPE");

//get an object of Shape Circle

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

//call draw method of Shape Circle

shape1.draw();

//get an object of Shape Rectangle

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

//call draw method of Shape Rectangle

shape2.draw();

//get an object of Shape Square

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

//call draw method of Shape Square

shape3.draw();

//get color factory

AbstractFactory colorFactory = FactoryProducer.getFactory("COLOR");

//get an object of Color Red

Color color1 = colorFactory.getColor("RED");

//call fill method of Red

color1.fill();

//get an object of Color Green

Color color2 = colorFactory.getColor("Green");

//call fill method of Green

color2.fill();

//get an object of Color Blue

Color color3 = colorFactory.getColor("BLUE");

//call fill method of Color Blue

color3.fill();

}

}

Output

Inside Circle::draw() method.

Inside Rectangle::draw() method.

Inside Square::draw() method.

Inside Red::fill() method.

Inside Green::fill() method.

Inside Blue::fill() method.