Assignment 4: Abstract Factory Pattern

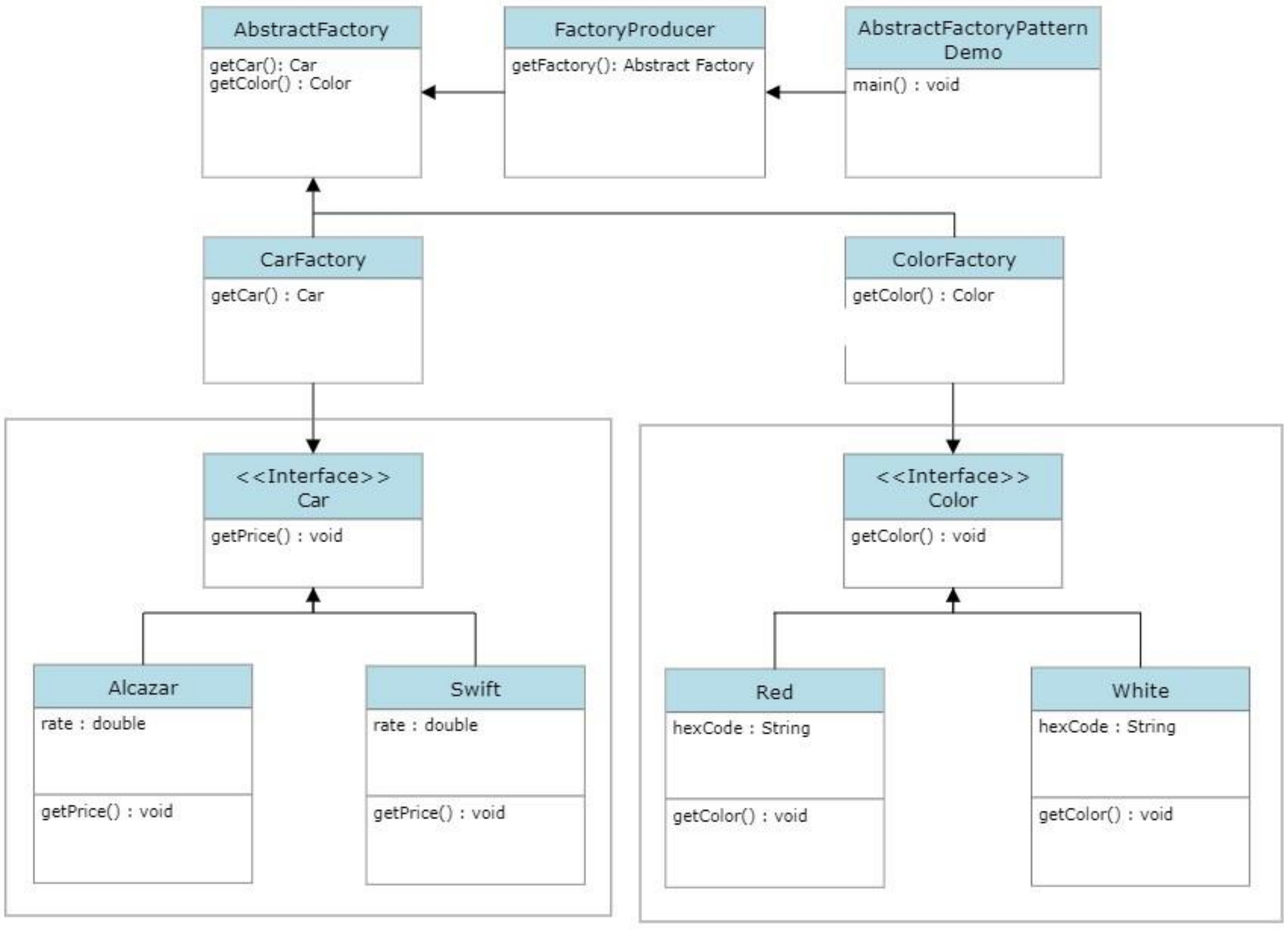
What is Abstract Factory Design Pattern?

Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

Intend

- ✓ 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



Implementation (Code)

- **Car.java**

```
abstract class Car {
    protected double rate;
    abstract void getPrice();
}
```

- **Alcazar.java**

```
public class Alcazar extends Car {
   public void getPrice() {
      rate = 1500000;
      System.out.println("Rate for Alcazar: " + rate);
   }
}
```

- **Swift.java**

```
public class Swift extends Car {
   public void getPrice() {
      rate = 500000;
      System.out.println("Rate for Swift: " + rate);
   }
}
```

- **CarFactory.java**

```
public class CarFactory extends AbstractFactory{
   public Car getCar(String carName){
      if (carName == null){
         return null;
      }
      if (carName.equalsIgnoreCase("Alcazar")){
         return new Alcazar();
      }
      else if (carName.equalsIgnoreCase("Swift")){
         return new Swift();
      }
      return null;
   }
   Color getColor(String color) {
      return null;
   }
}
```

- **Color.java**

```
abstract class Color {
   protected String hexCode;
   abstract void getColor();
}
```

- **Red.java**

```
public class Red extends Color {
   public void getColor(){
      hexCode = "#FF0000";
      System.out.println("Hex Code: "+hexCode);
   }
}
```

- **White.java**

```
public class White extends Color {
   public void getColor(){
      hexCode = "#FFFFFF";
      System.out.println("Hex Code: "+hexCode);
   }
}
```

- **ColourFactory.java**

```
public class ColourFactory extends AbstractFactory{
   public Color getColor(String color){
      if (color == null){
         return null;
      }
      if (color.equalsIgnoreCase("Red")){
         return new Red();
      }
      else if (color.equalsIgnoreCase("White")){
         return new White();
      }
      return null;
   }
   Car getCar(String carName) {
      return null;
   }
}
```

- **AbstractFactory.java**

```
public abstract class AbstractFactory {
   abstract Color getColor(String color);
   abstract Car getCar(String carName);
}
```

- **Producer.java**

```
public class Producer {
   public static AbstractFactory getFactory(String choice){
      if (choice.equals("Car")){
         return new CarFactory();
      }
      else if (choice.equals("Color")){
         return new ColourFactory();
      }
      return null;
   }
}
```

➢ **Demo.java**

```
import java.util.*;
public class Demo {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.print("Enter the Car name[Alcazar, Swift]: ");
      String carName = sc.nextLine();
      System.out.print("Enter the Color[White, Red]: ");
      String color = sc.nextLine();


      AbstractFactory carFactory = Producer.getFactory("Car");
      assert carFactory != null;
      Car car1 = carFactory.getCar(carName);
      car1.getPrice();

      AbstractFactory colorFactory = Producer.getFactory("Color");
      assert colorFactory != null;
      Color color1 = colorFactory.getColor(color);
      color1.getColor();
   }
}
```

Applicability

1. Use the Abstract Factory when your code needs to work with various families of related products, but you don't want it to depend on the concrete classes of those products—they might be unknown beforehand or you simply want to allow for future extensibility.
2. The Abstract Factory provides you with an interface for creating objects from each class of the product family. As long as your code creates objects via this interface, you don't have to worry about creating the wrong variant of a product which doesn't match the products already created by your app.