Factory

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

Question: ***Which creational pattern is used in the following code?***
We're going to create a *Shape* interface and concrete classes implementing the *Shape* interface. A pattern class *Pattern* is defined as a next step.
PatternDemo, our demo class will use *Pattern* to get a *Shape* object. It will pass information (CIRCLE / RECTANGLE / SQUARE) to *Pattern* to get the type of object it needs.

1. Create an interface.

Shape.java

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

2. 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.");
  }
}
```

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

Pattern.java

```
public class Pattern {

  //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;
  }
}
```

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

PatternDemo.java

```
public class PatternDemo {

  public static void main(String[] args) {
    Pattern Pattern = new Pattern ();

    Shape shape1 = Pattern.getShape("CIRCLE");
    shape1.draw();

    Shape shape2 = Pattern.getShape("RECTANGLE");
    shape2.draw();

    Shape shape3 = Pattern.getShape("SQUARE");
    shape3.draw();
  }
}
```

5. Verify the output.

(Output)

```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

Reference: https://www.tutorialspoint.com/design_pattern/factory_pattern.htm

Singleton

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.

Question: ***Which creational pattern is used in the following code?***
We're going to create a *Pattern* class. *Pattern* class has its constructor as private and has a static instance. *Pattern* class provides a static method to get its static instance to outside world. *PatternDemo*, our demo class will use *Pattern* class to get a *Pattern* object.

1. Create a Pattern Class.

Pattern.java

```
public class Pattern {

  private static Pattern instance = new Pattern ();

  //make the constructor private so that this class cannot be instantiated
  private Pattern (){}

  //Get the only object available
  public static Pattern getInstance(){
    return instance;
  }

  public void showMessage(){
    System.out.println("Hello World!");
  }
}
```

2. Get the only object from the Pattern class.

PatternDemo.java

```
public class PatternDemo {
  public static void main(String[] args) {

    Pattern object = new Pattern ();

    Pattern object = Pattern.getInstance();

    object.showMessage();
  }
}
```

3. Verify the output.

(Output)

```
Hello World!
```

Reference: https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm

Builder

A Builder class builds the final object step by step. This builder is independent of other objects.

Question: ***Which creational pattern is used in the following code?***
We have considered a business case of fast-food restaurant where a typical meal could be a burger and a cold drink. Burger could be either a Veg Burger or Chicken Burger and will be packed by a wrapper. Cold drink could be either a coke or pepsi and will be packed in a bottle.
We are going to create an *Item* interface representing food items such as burgers and cold drinks and concrete classes implementing the *Item* interface and a *Packing* interface representing packaging of food items and concrete classes implementing the *Packing* interface as burger would be packed in wrapper and cold drink would be packed as bottle.
We then create a *Meal* class having *ArrayList* of *Item* and a *Pattern* to build different types of *Meal* objects by combining *Item*. *PatternDemo*, our demo class will use *Pattern* to build a *Meal*.

1. Create an interface Item representing food item and packing.

Item.java

```
public interface Item {
  public String name();
  public Packing packing();
  public float price();
}
```

Packing.java

```
public interface Packing {
  public String pack();
}
```

2. Create concrete classes implementing the Packing interface.

Wrapper.java

```
public class Wrapper implements Packing {

  @Override
  public String pack() {
    return "Wrapper";
  }
}
```

Bottle.java

```
public class Bottle implements Packing {

  @Override
  public String pack() {
    return "Bottle";
  }
}
```

3. Create abstract classes implementing the item interface providing default functionalities.

Burger.java

```
public abstract class Burger implements Item {

  @Override
  public Packing packing() {
    return new Wrapper();
  }

  @Override
  public abstract float price();
}
```

ColdDrink.java

```
public abstract class ColdDrink implements Item {

   @Override
   public Packing packing() {
      return new Bottle();
   }

   @Override
   public abstract float price();
}
```

4. Create concrete classes extending Burger and ColdDrink classes

VegBurger.java

```
public class VegBurger extends Burger {

  @Override
  public float price() {
    return 25.0f;
  }

  @Override
  public String name() {
    return "Veg Burger";
  }
}
```

ChickenBurger.java

```
public class ChickenBurger extends Burger {

  @Override
  public float price() {
    return 50.5f;
  }

  @Override
  public String name() {
    return "Chicken Burger";
  }
}
```

Coke.java

```
public class Coke extends ColdDrink {

  @Override
  public float price() {
    return 30.0f;
  }

  @Override
  public String name() {
    return "Coke";
  }
}
```

Pepsi.java

```
public class Pepsi extends ColdDrink {

  @Override
  public float price() {
    return 35.0f;
  }

  @Override
  public String name() {
    return "Pepsi";
  }
}
```

5. Create a Meal class having Item objects defined above.

Meal.java

```
import java.util.ArrayList;
import java.util.List;

public class Meal {
  private List<Item> items = new ArrayList<Item>();

  public void addItem(Item item){
    items.add(item);
  }

  public float getCost(){
    float cost = 0.0f;

    for (Item item : items) {
      cost += item.price();
    }
    return cost;
  }

  public void showItems(){

    for (Item item : items) {
      System.out.print("Item : " + item.name());
      System.out.print(", Packing : " + item.packing().pack());
      System.out.println(", Price : " + item.price());
    }
  }
}
```

6. Create a Pattern class, the actual builder class responsible to create Meal objects.

Pattern.java

```
public class Pattern {

  public Meal prepareVegMeal (){
    Meal meal = new Meal();
    meal.addItem(new VegBurger());
    meal.addItem(new Coke());
    return meal;
  }

  public Meal prepareNonVegMeal (){
    Meal meal = new Meal();
    meal.addItem(new ChickenBurger());
    meal.addItem(new Pepsi());
    return meal;
  }
}
```

7. PatternDemo uses Pattern.

PatternDemo.java

```
public class PatternDemo {
  public static void main(String[] args) {

    Pattern pattern = new Pattern ();

    Meal vegMeal = pattern.prepareVegMeal();
    System.out.println("Veg Meal");
    vegMeal.showItems();
    System.out.println("Total Cost: " + vegMeal.getCost());

    Meal nonVegMeal = pattern.prepareNonVegMeal();
    System.out.println("\n\nNon-Veg Meal");
    nonVegMeal.showItems();
    System.out.println("Total Cost: " + nonVegMeal.getCost());
  }
}
```

8. Verify the output.

(Output)

```
Veg Meal
Item : Veg Burger, Packing : Wrapper, Price : 25.0
Item : Coke, Packing : Bottle, Price : 30.0
Total Cost: 55.0

Non-Veg Meal
Item : Chicken Burger, Packing : Wrapper, Price : 50.5
Item : Pepsi, Packing : Bottle, Price : 35.0
Total Cost: 85.5
```

Reference: https://www.tutorialspoint.com/design_pattern/builder_pattern.htm

Prototype

Prototype pattern refers to creating duplicate object while keeping performance in mind. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
This pattern involves implementing a prototype interface, which tells to create a clone of the current object. This pattern is used when creation of object directly is costly. For example, an object is to be created after a costly database operation. We can cache the object, returns its clone on next request and update the database as and when needed thus reducing database calls.

Question: ***Which creational pattern is used in the following code?***
We're going to create an abstract class *Shape* and concrete classes extending the *Shape* class. A class *Pattern* is defined as a next step, which stores shape objects in a *Hashtable* and returns their clone when requested.
PatternDemo, our demo class will use *Pattern* class to get a *Shape* object.

1. Create an abstract class implementing Clonable interface.

Shape.java

```
public abstract class Shape implements Cloneable {

  private String id;
  protected String type;

  abstract void draw();

  public String getType(){
    return type;
  }

  public String getId() {
    return id;
  }

  public void setId(String id) {
    this.id = id;
  }

  public Object clone() {
    Object clone = null;

    try {
      clone = super.clone();

    } catch (CloneNotSupportedException e) {
      e.printStackTrace();
    }

    return clone;
  }
}
```

2. Create concrete classes extending the above class.

Rectangle.java

```
public class Rectangle extends Shape {

   public Rectangle(){
     type = "Rectangle";
   }

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

Square.java

```
public class Square extends Shape {

   public Square(){
     type = "Square";
   }

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Circle.java

```
public class Circle extends Shape {

   public Circle(){
     type = "Circle";
   }

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```

3. Create a class to get concrete classes from database and store them in a Hashtable.

Pattern.java

```
import java.util.Hashtable;

public class Pattern {

   private static Hashtable<String, Shape> shapeMap  = new Hashtable<String,
Shape>();
```

```
  public static Shape getShape(String shapeId) {
    Shape cachedShape = shapeMap.get(shapeId);
    return (Shape) cachedShape.clone();
  }

  // for each shape run database query and create shape
  // shapeMap.put(shapeKey, shape);
  // for example, we are adding three shapes

  public static void loadCache() {
    Circle circle = new Circle();
    circle.setId("1");
    shapeMap.put(circle.getId(),circle);

    Square square = new Square();
    square.setId("2");
    shapeMap.put(square.getId(),square);

    Rectangle rectangle = new Rectangle();
    rectangle.setId("3");
    shapeMap.put(rectangle.getId(), rectangle);
  }
}
```

4. PatternDemo uses Pattrn class to get clones of shapes stored in a Hashtable.

PatternDemo.java

```
public class PatternDemo {
  public static void main(String[] args) {
    Pattern.loadCache();

    Shape clonedShape = (Shape) Pattern.getShape("1");
    System.out.println("Shape : " + clonedShape.getType());

    Shape clonedShape2 = (Shape) Pattern.getShape("2");
    System.out.println("Shape : " + clonedShape2.getType());

    Shape clonedShape3 = (Shape) Pattern.getShape("3");
    System.out.println("Shape : " + clonedShape3.getType());
  }
}
```

5. Verify the output.

(Output)

```
Shape : Circle
Shape : Square
Shape : Rectangle
```

Reference: https://www.tutorialspoint.com/design_pattern/prototype_pattern.htm

Abstract Factory

Abstract Factory patterns 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.
In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.

Question: ***Which creational pattern is used in the following code?***
We are going to create a *Shape* and *Color* interfaces and concrete classes implementing these interfaces. We create an abstract factory class *Pattern* as next step. Creator classes *ShapeCreator* and *ColorCreator* are defined where each creator extends *Pattern*. A factory creator/generator class *CreatorProducer* is created.
PatternDemo, our demo class uses *CreatorProducer* to get a *Pattern* object. It will pass information (*CIRCLE / RECTANGLE / SQUARE* for *Shape*) to *AbstractFactory* to get the type of object it needs. It also passes information (*RED / GREEN / BLUE* for *Color*) to *Pattern* to get the type of object it needs.

1. Create an interface for Shapes.

Shape.java

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

2. 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.");
  }
}
```

3. Create an interface for Colors.

Color.java

```
public interface Color {
  void fill();
}
```

4. 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.");
  }
}
```

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

Pattern.java

```
public abstract class Pattern {
  abstract Color getColor(String color);
  abstract Shape getShape(String shape) ;
}
```

6. Create creator classes extending Pattern to generate object of concrete class based on given information.

ShapeCreator.java

```
public class ShapeCreator extends Pattern {

  @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;
  }
}
```

ColorCreator.java

```
public class ColorCreator extends Pattern {

  @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;
    }
  }
```

7. Create a creator generator/producer class to get creators by passing an information such as Shape or Color

CreatorProducer.java

```
public class CreatorProducer {
  public static Pattern getCreator(String choice){

    if(choice.equalsIgnoreCase("SHAPE")){
      return new ShapeCreator();

    }else if(choice.equalsIgnoreCase("COLOR")){
      return new ColorCreator();
    }

    return null;
  }
}
```

8. Use the CreatorProducer to get Pattern in order to get factories of concrete classes by passing information such as type.

PatternDemo.java

```
public class PatternDemo {
  public static void main(String[] args) {

    //get shape creator
    Pattern shapeCreator = CreatorProducer.getCreator("SHAPE");

    //get an object of Shape Circle
    Shape shape1 = shapeCreator.getShape("CIRCLE");

    //call draw method of Shape Circle
    shape1.draw();

    //get an object of Shape Rectangle
```

```
      Shape shape2 = shapeCreator.getShape("RECTANGLE");

      //call draw method of Shape Rectangle
      shape2.draw();

      //get an object of Shape Square
      Shape shape3 = shapeCreator.getShape("SQUARE");

      //call draw method of Shape Square
      shape3.draw();

      //get color factory
      Pattern colorCreator = CreatorProducer.getCreator("COLOR");

      //get an object of Color Red
      Color color1 = colorCreator.getColor("RED");

      //call fill method of Red
      color1.fill();

      //get an object of Color Green
      Color color2 = colorCreator.getColor("Green");

      //call fill method of Green
      color2.fill();

      //get an object of Color Blue
      Color color3 = colorCreator.getColor("BLUE");

      //call fill method of Color Blue
      color3.fill();
   }
}
```

9. Verify the output.

(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.
```

Reference: https://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm