Creational Design Patterns

“ *“I think computer viruses should count as life ... I think it says something about human nature that the only form of life we have created so far is purely destructive. We've created life in our own image.*” — Stephen Hawking

***Creational** patterns focus on abstracting the instantiation process, allowing systems to be independent of creation, composition, and representation details.*

So far in our coding examples, we have been creating concrete objects using the **new** keyword.

```
// Create an instance of PaymentProcessor
PaymentProcessor paymentProcessor = new PaymentProcessor();
```

However, at the core of all design principles we have studied so far, we are encouraged to program to an interface and not an implementation. And when we instantiate a class using **new** keyword, that is definitely programming to implementation.

Well, then one might ask, how are we going to instantiate a class or create objects?

We will still be using **new** keyword for object creation; however, we will encapsulate the process so that we have better flexibility and more code reuse.

Creational patterns help in designing a system independent of Object Creation process. They intend to give more flexibility in object creation by letting us configure at compile time or run time, what gets created, who creates it, how it gets created, and when it gets created.

1.1. Factory Pattern

Factory as the name suggests is used to produce goods. Similarly, in object-oriented world, we create a virtual factory to create objects and ask it to provide us with specific objects whenever required.

***Factory Pattern** is used to abstract the object creation logic from run time.*

Let us look at a relatable scenario to understand this better.

> *Say, in a manufacturing company, there is a need to create different types of vehicles. The company produces cars, bikes, and trucks. The process of creating these vehicles involves specific manufacturing steps for each type.*

Here is how would traditionally approach to this requirement would look like.

```
package Factory;

/**
 * Vehicle Class
 */
public abstract class Vehicle {
    abstract void manufacture();
    // Other methods can be defined with respect to Vehicles
}
```

```java
// Concrete Car class extending the Vehicle
public class Car extends Vehicle {
    @Override
    public void manufacture() {
        System.out.println("Car is being manufactured.");
    }
}

// Concrete Bike class extending the Vehicle
public class Bike extends Vehicle {
    @Override
    public void manufacture() {
        System.out.println("Bike is being manufactured.");
    }
}

// Concrete Truck class extending the Vehicle
public class Truck extends Vehicle {
    @Override
    public void manufacture() {
        System.out.println("Truck is being manufactured.");
    }
}

package Factory;
/**
 * Vehicle Customizer class is used to add features to vehicles and manufacture them
 */
public class VehicleCustomizer {
    public void customizeVehicle(String type) {
        // create specific vehicle instances
        Vehicle vehicle;
        if (type.equalsIgnoreCase("car")) {
            vehicle = new Car();
        } else if (type.equalsIgnoreCase("bike")) {
            vehicle = new Bike();
        } else if (type.equalsIgnoreCase("truck")) {
            vehicle = new Truck();
        }

        // implement customization logic
        vehicle.manufacture();
    }

}
```

 **Issues with above solution**

1. Whenever we have to introduce a new vehicle type or remove an existing vehicle type, based on the business requirement, we will have to change the code in Customizer class. It breaks OCP principle.
2. Customizer class is first creating objects and then customizing vehicle. It is currently sharing more than one responsibility, hence breaking SRP principle.
3. In current example Customizer is just one client using Vehicle objects, but in application there would be many such clients. Imagine code modification during requirement changes.

 So, the factory pattern can be leverage here and object creation logic can be encapsulated to make code more usable and loosely coupled and adhering to design principles, which in turn will help us in maintenance of the code.

 Let us understand Factory Pattern implementation via code example below.

```java
package Factory;

/**
 * Vehicle Factory to produce objects of different vehicle types
 */
public class VehicleFactory {
    /**
     * This method returns objects of Different Vehicle subclasses
     * @param type : Vehicle type
     * @return objects of Different Vehicle subclasses
     */
    public Vehicle createVehicle(String type) {
        if (type.equalsIgnoreCase("car")) {
            return new Car();
        } else if (type.equalsIgnoreCase("bike")) {
            return new Bike();
        } else if (type.equalsIgnoreCase("truck")) {
            return new Truck();
        }
        return null;
    }
}
package Factory;
/**
 * Vehicle Customizer class is used to add features to vehicles and manufacture
 */
public class VehicleCustomizer {
    VehicleFactory vehicleFactory;
    /**
     * This method customizes vehicles based on specific types
     * @param type
     */
    public void customizeVehicle(String type) {
        // create specific vehicle instances
        Vehicle vehicle;
        vehicle = vehicleFactory.createVehicle(type);
        // implement customization logic
        vehicle.manufacture();
    }

}
```

 Using **Factory pattern**, we have encapsulated object creation logic and during requirement changes like addition of new vehicle types or removal of older one, changes need to be done only in *VehicleFactory* and not through entire code base.

1.2. Abstract Factory Pattern

Now, it is time to improve the code further. Even though, we have encapsulated object creation logic in the program above, we still have a **tight coupling in the factory** when it comes to create new objects. Yes, you identified it correctly, we are still doing a manual type check before creating object in the **factory**.

 *Design principles say that we should always **program to an interface, not an implementation**. We must identify the aspects of your application that vary and separate them from what stays the same.*

And we will follow the exact words. In our example, we understand that different vehicle types can be introduced or removed at any given instant. Hence, we would declare interfaces for each distinct vehicle of the vehicle family.

 Let us understand this implementation via below code example.

```
package Factory;
/**
 * Abstraction of Vehicle Factory
 */
public interface VehicleFactory {
    Vehicle createVehicle();
}

/**
 * Concrete CarFactory class implementing the VehicleFactory interface
 */
public class CarFactory implements VehicleFactory {
    @Override
    public Vehicle createVehicle() {
        return new Car();
    }
}

/**
 * Concrete BikeFactory class implementing the VehicleFactory interface
 */
public class BikeFactory implements VehicleFactory {
    @Override
    public Vehicle createVehicle() {
        return new Bike();
    }
}

/**
 * Concrete TruckFactory class implementing the VehicleFactory interface
 */
public class TruckFactory implements VehicleFactory {
    @Override
    public Vehicle createVehicle() {
        return new Truck();
    }
}
```

 *We use the **Abstract Factory Pattern** to create families of related objects without specifying their concrete classes, in this case, vehicles.*

 The *Vehicle* interface remains the same as before, defining the common functionality for all vehicles. The concrete vehicle classes (*Car*, *Bike*, *Truck*) also remain unchanged. However, instead of a single *VehicleFactory* class, we now have an abstract *VehicleFactory* interface. This interface declares a *createVehicle*() method that returns a *Vehicle* object. The concrete factories *(CarFactory*, *BikeFactory*, *TruckFactory)* implement the *VehicleFactory* interface. Each factory is responsible for creating a specific type of vehicle. For example, the *CarFactory* creates *Car* objects, the *BikeFactory* creates *Bike* objects, and so on.

Now, let us look at how this code would be consumed by *VehicleCustomizer* client.

```
package Factory;
```

```
/**
 * Vehicle Customizer class is used to add features to vehicles and manufacture
 */
public class VehicleCustomizer {
    VehicleFactory vehicleFactory;
    /**
     * This method customizes vehicles based on specific types
     * @param type
     */
    public void customizeVehicle(String type) {
        // create specific vehicle instances
        VehicleFactory carFactory = new CarFactory();
        Vehicle car = carFactory.createVehicle();
        car.manufacture();

        VehicleFactory bikeFactory = new BikeFactory();
        Vehicle bike = bikeFactory.createVehicle();
        bike.manufacture();

        VehicleFactory truckFactory = new TruckFactory();
        Vehicle truck = truckFactory.createVehicle();
        truck.manufacture();
    }

}
```

 Here in *VehicleCustomizer* client code, we create instances of the concrete factories (*carFactory*, *bikeFactory*, *truckFactory*). Then, we use the factory objects to create the corresponding vehicles by invoking the *createVehicle*() method. The factory internally handles the object creation process and returns the appropriate concrete vehicle object.

 By utilizing the **Abstract Factory Pattern**, the client code is decoupled from the concrete vehicle classes and specific factory implementations. Instead, the client code interacts with the abstract factory interface, allowing for flexibility and easy substitution of different factory implementations. This pattern enables the creation of families of related objects and provides a higher level of abstraction in the code structure.

1.3. Builder Pattern

So far in our [example of vehicle application], we have been creating objects of different types of vehicles, however, vehicles had zero or limited attributes assigned. Code in constructor has been clean. Now, let us increase the complexity to some degree so that our simple program resembles more to classes or objects in real world.

Currently, we have three vehicle types into our system: Car, Bike and Truck. Now, let us just focus on one vehicle type, say Car. Business heads want to add more details of a car to be captured by the application, such as engine, wheel, brand, etc. These attributes may have their own attributes being captured by the system. Engine may have attributes like engine type and horsepower. Now, we have case of nested attributes and object creation logic gets complex inside constructor.

```
     /**
     * Constructs a new Car object with the provided parameters.
     *
     * @param wheel  The wheel of the car.
     * @param brand  The brand of the car.
     * @param color  The color of the car.
     */
    private Car(String wheel, String brand, String color) {
        this.wheel = wheel;
        this.brand = brand;
```

```
        this.color = color;
        Engine engine = new Engine("V8", 500);
        this.engine = engine;
    }
```

 Now, *Engine* object creation is also happening inside constructor of *Car*. One might argue that this one statement of engine creation can be pulled out of constructor code and constructor be supplied with Engine object via arguments. Agreed, this can be a possible approach. However, imagine a situation, when you are creating a Car object and engine information is not available. Now, class Car is tightly coupled with class Engine. Repercussions you are smart enough to figure out.

 **Builder Pattern** comes to our rescue!

 ***Builder Pattern*** *lets us construct complex objects step by step. It allows us to produce different types and representations of an object using the same construction code.*

 Let us look at the modified code leveraging this pattern and decoupling the dependencies.

```
package VehicleExample;

/**
 * Concrete Car Class
 */
public class Car extends Vehicle {
    private String wheel;
    private String brand;
    private String color;
    private Engine engine;

    /**
     * Retrieves the wheel of the vehicle.
     *
     * @return The wheel of the vehicle.
     */
    public String getWheel() {
        return wheel;
    }

    /**
     * Retrieves the brand of the vehicle.
     *
     * @return The brand of the vehicle.
     */
    public String getBrand() {
        return brand;
    }

    /**
     * Retrieves the color of the vehicle.
     *
     * @return The color of the vehicle.
     */
    public String getColor() {
        return color;
    }

    /**
     * Retrieves the engine of the vehicle.
     *
     * @return The engine of the vehicle.
     */
```

```java
    public Engine getEngine() {
        return engine;
    }

    /**
     * Constructs a new Car object with the provided builder.
     *
     * @param builder The CarBuilder object used to construct the Car.
     */
    private Car(CarBuilder builder) {
        this.wheel = builder.wheel;
        this.brand = builder.brand;
        this.color = builder.color;
        this.engine = builder.engine;
    }

    @Override
    public void manufacture() {
        System.out.println("Car with " + this.getWheel() + " wheel is being manufactured.");
    }

    // CarBuilder class for constructing Car objects
    public static class CarBuilder {
        private String wheel;
        private String brand;
        private String color;
        private Engine engine;

        /**
         * Sets the wheel of the car.
         *
         * @param wheel The wheel to set.
         * @return The CarBuilder instance.
         */
        public CarBuilder setWheel(String wheel) {
            this.wheel = wheel;
            return this;
        }

        /**
         * Sets the brand of the car.
         *
         * @param brand The brand to set.
         * @return The CarBuilder instance.
         */
        public CarBuilder setBrand(String brand) {
            this.brand = brand;
            return this;
        }

        /**
         * Sets the color of the car.
         *
         * @param color The color to set.
         * @return The CarBuilder instance.
         */
        public CarBuilder setColor(String color) {
            this.color = color;
            return this;
        }

        /**
         * Sets the engine of the car.
         *
         * @param engine The engine to set.
         * @return The CarBuilder instance.
```

```
         */
        public CarBuilder setEngine(Engine engine) {
            this.engine = engine;
            return this;
        }

        /**
         * Builds and returns a new Car instance with the configured properties.
         *
         * @return A new Car instance.
         */
        public Car build() {
            return new Car(this);
        }
    }
}
```

 In this modified version of the code, The *CarBuilder* class is a nested static class within the *Car* class. By making the *CarBuilder* class static and nested within the *Car* class, it is encapsulated within the scope of the *Car* class. This encapsulation allows the builder class to access the private constructor of the *Car* class, which enforces object creation through the builder. Since the *CarBuilder* class is static, it can be accessed without requiring an instance of the *Car* class. This means that the client code can create a builder object and invoke its methods directly without needing to instantiate the *Car* class first.

Irony: To remove the tight coupling during object creation with nested properties, we have introduced a tight coupling of our own. A necessary trade off in this case.

```
package VehicleExample;

public class Client {
    public static void main(String[] args) {
        Engine engine = new Engine("V8", 500);
        Car car = new Car.CarBuilder()
                .setWheel("MRF")
                .setBrand("Toyota")
                .setColor("Red")
                .setEngine(engine)
                .build();

        car.manufacture(); // output: Car with MRF wheel is being manufactured.
        System.out.println("Engine of Car: "+ car.getEngine()); // output: Engine of Car:
2LTurbo
    }
}
```

 In the client code, we create an *Engine* object with the desired engine details. Then, using the *Car.Builder*, we set the car properties including the *Engine* object. Finally, we build the *Car* object and utilize it by invoking the manufacture method and retrieving the car and engine details.

 The **Builder Pattern** provides a way to construct complex objects step-by-step, allowing for flexible and readable object creation. It separates the construction logic from the object's representation, enabling the construction of objects with different configurations using the same building process. It simplifies the object creation process, supports optional and default property values, and ensures the resulting objects are in a consistent state.

1.4. Prototype Pattern

Different requirements in application arise need for different patterns in the solution. One pattern might be suitable for a specific scenario but won't fit at all for some other. With time and experience, we gain enough expertise to get an instinct about when to use a particular pattern.

In the above three creational patterns, we learnt how we can encapsulate object creation logic to suit our business needs. Carrying forward the discussion on similar lines, let us entertain another business scenario.

In the automotive industry, there are often standard configurations or models of vehicles that have slight variations based on customer preferences or market demands. Customers often have specific preferences for their vehicles, such as color, interior features, or additional accessories. In the vehicle development process, it is common to create prototypes for testing and validation purposes.

 Now, to replicate this scenario in our system, which enables to vary type and configuration of vehicles easily, we use **Prototype Pattern.**

 ***Prototype pattern*** *that lets us specify the kind of objects we want to create using a prototypical instance and cloning existing objects (prototypes) without making our code dependent on their classes.*

 Applying this pattern in our code, we add a *clone()* method in our *Car* class, when called via instance of the class, returns a new object with configuration similar to the instance of the object which has called the function.

```
    public Vehicle clone() {
        return new Car(this.type, this.brand, this.color);
    }
```

 Client code uses the returned original object as a prototype, clones and creates a new object and adds necessary configuration as per requirement.

```
Car carPrototype = new Car("Car", "Toyota", "Red");
carPrototype.manufacture();

Car clonedCar = (Car) carPrototype.clone();
clonedCar.manufacture();

// Modify the properties of the cloned car
clonedCar.setBrand("Honda");
clonedCar.setColor("Blue");
clonedCar.manufacture();
```

 **The Prototype pattern** delegates the cloning process to the actual objects that are being cloned. The clone method creates an object of the current class and carries over all of the field values of the old object into the new one. This pattern can be used when a system needs to be independent of how objects are created, composed or represented and class instantiation is specified at run time. It helps to avoid building a class hierarchy of factories that is parallel to hierarchy of products. Cloning avoids the costly process of initializing objects from scratch, as it creates a replica of an existing object with pre-initialized properties. The Prototype pattern can greatly reduce the number of classes a system needs.

Cloning can be a difficult process to implement if a class internally includes objects that do not support copying or have circular reference.

1.5. Singleton Pattern

Continuing with our vehicle context, manufacturers are outsourced by vehicle makers for different parts. There can be a scenario that a particular manufacturer is utilized to supply car with certain basic configuration, say a specific engine type. Other properties of the model may vary and can be supplied, which the manufacturer would utilize in addition with its own specification of engine, to produce a new car model. Since, this is a unique manufacturer, system requires to have only one global instance of it throughout the application.

This is where **Singleton Pattern** might come handy. We can limit resource allocation to just one instance of manufacturer throughout the application, and it will help ensure synchronization and adherence to predefined protocols. The singleton instance can act as a central control point for the manufacturing system. It allows for centralized configuration and management of various aspects, such as production rules, quality standards, or system-wide settings. This simplifies the maintenance and updates of these configurations.

Singleton Pattern *makes sure that a class has only one instance throughout application and a global access point is provided to access functionality of the class*.

Let us look the code snippet below.

```
package VehicleExample;

public class CarManufacturer {
    private static CarManufacturer instance;

    private CarManufacturer() {
        // Perform initialization here
    }

    public static CarManufacturer getInstance() {
        if (instance == null) {
            instance = new CarManufacturer();
        }
        return instance;
    }

    public Car manufactureCar(String wheel, String brand, String color) {
        // Manufacturing logic here
        Engine engine = new Engine("V8", 500);
        Car car = new Car.CarBuilder()
                .setWheel(wheel)
                .setBrand(brand)
                .setColor(color)
                .setEngine(engine)
                .build();
        return car;
    }
}
```

Here in the above code example, I have incorporated the code of Builder pattern for *Car* object creation inside *CarManufacturer,* which has a *private static instance* variable instance to hold the single instance of the class. The constructor is made private to prevent direct instantiation from outside the class. The *getInstance()* method provides access to the singleton instance. If the instance

variable is *null*, it creates a new instance of *CarManufacturer*. If the instance variable is already set, it simply returns the existing instance.

The *CarManufacturer* class also includes a *manufactureCar()* method, which represents the manufacturing process for cars. It takes parameters such as the wheel, color, and brand, and adds the specific engine type, for which this manufacturer specializes and returns the new created *Car* object.

 **Singleton Pattern** helps in limiting the instance of a class to one, which ensures efficient allocation and utilization of available resources. Having multiple instances could lead to resource wastage or conflicts in resource allocation. Since the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it. Use of this pattern is strictly in adherence to business requirements. Here, we are limiting the accessibility of a class to one global instance but in other cases, where multiple instances of a class is desired, we must avoid Singleton Pattern.

1.6. Summary

In this chapter, we discussed different creational patterns at length and gained an insight into when it suitable to use them and when we must avoid their use.

- **Factory Pattern:** The Factory Pattern provides an interface for creating objects, but the specific class to instantiate is determined by the factory class. It encapsulates object creation and decouples the client code from the specific object classes. It allows for flexible object creation based on different criteria.
- **Abstract Factory Pattern:** The Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. It encapsulates a group of factory methods for creating different types of objects. It promotes the creation of objects that are compatible and work together.
- **Builder Pattern:** The Builder Pattern separates the construction of a complex object from its representation. It allows step-by-step creation of an object by providing a builder class that handles the construction process. It provides flexibility in constructing objects with optional or default property values.
- **Prototype Pattern:** The Prototype Pattern involves creating new objects by cloning existing objects, known as prototypes. It allows objects to be copied or cloned to create new instances with the same properties. This pattern is useful when creating new objects is costly or complex, and when objects need to be created dynamically at runtime.
- **Singleton Pattern:** The Singleton Pattern ensures that there is only one instance of a class throughout the application. It provides a global access point to the same instance, allowing centralized control and coordination. It is useful when we need to manage a shared resource or when having multiple instances would be redundant or inefficient.