Creational Design Patterns in Java

stackabuse.com/creational-design-patterns-in-java/

Overview

This is the first article in a short series dedicated to Design Patterns in Java.

Creational Patterns

The Creational Patterns in Java that are covered in this article are:

Factory Method

The Factory Method, also often called the Factory Pattern is a widely used design pattern that commands object creation.

In this pattern, a *Factory* class is created as the parent class of all sub-classes belonging to a certain logical segment of related classes.

Just like a `SessionFactory` is used to create, update, delete and manipulate all `Session` objects, so is any other factory responsible for their set of child classes.

It's important to note that the sub-classes can't be reached without using their respective factory. This way, their creation is both hidden from the client and is dependent on the factory.

Implementation:

Let's build a small, simple project to demonstrate this.

We're going to define a few classes belonging to a logic segment, each of them implementing the same interface. Then we're going to create a factory for these objects.

```
public interface Animal {
   void eat();
}
```

The interface only has one method for the convenience of presenting the point.

Now, let's define a few classes that implement this interface, each in their own way:

```
 public class Dog implements Animal {
    @Override
    public void eat() {
        System.out.println("Dog is eating, woof!");
    }
}

public class Cat implements Animal {
    @Override
    public void eat() {
        System.out.println("Cat is eating, meow!");
    }
}

public class Rabbit implements Animal {
    @Override
    public void eat() {
        System.out.println("Rabbit is eating, squeak!");
    }
}
```

Note: These classes are separate *.java* files, they're grouped together like this for readability.

Now that we have a group of classes, we can designate a factory for them:

```
public class AnimalFactory {

    public Animal getAnimal(String animal) {
        if(animal.equals(null)) return null;

        if(animal.equalsIgnoreCase("Dog")) {
            return new Dog();
        } else if(animal.equalsIgnoreCase("Cat")) {
            return new Cat();
        } else if(animal.equalsIgnoreCase("Rabbit")) {
            return new Rabbit();
        }
        return null;
    }
}
```

This way, we have a factory to instantiate our objects in a predefined way by the factory, without direct contact with the objects themselves.

Now, let's observe the result.

```
public class Main {
   public static void main(String[] args) {
      AnimalFactory animalFactory = new AnimalFactory();

      Animal animal = animalFactory.getAnimal("dOg");
      animal.eat();

      Animal animal2 = animalFactory.getAnimal("CAT");
      animal2.eat();

      Animal animal3 = animalFactory.getAnimal("raBbIt");
      animal3.eat();
   }
}
```

Running this piece of code will yield:

```
Dog is eating, woof!
Cat is eating, meow!
Rabbit is eating, squeak!
```

If you'd like to read a standalone detailed article on The Factory Method Design Pattern, we've got you covered!

Abstract Factory

The **Abstract Factory** design pattern builds upon the **Factory Pattern** and acts as the highest factory in the hierarchy. It represents the practice of creating a **factory of factories**.

This pattern is responsible for creating all other factories as its sub-classes, exactly like how factories are responsible for creating all of their own sub-classes.

Implementation:

The previous example can be used as a good base for this implementation.

The `Animal` interface is renamed to the `Pet` interface and each implementation is changed:

```
public class Dog implements Pet {
    @Override
    public void eat() {
        System.out.println("Dog is eating, woof!");
    }
}

public class Cat implements Pet {
    @Override
    public void eat() {
        System.out.println("Cat is eating, meow!");
    }
}

public class Rabbit implements Pet {
    @Override
    public void eat() {
        System.out.println("Rabbit is eating, squeak!");
    }
}
```

A new interface is defined:

```
public interface Human {
    public void feedPet();
}
```

And as usual, a few concrete classes implement this interface:

```
public class Child implements Human {
    @Override
    public void feedPet() {
        System.out.println("Child is feeding pet irresponsibly.");
    }
}

public class Adult implements Human {
    @Override
    public void feedPet() {
        System.out.println("Adult is feeding pet responsibly.");
    }
}

public class Elder implements Human {
    @Override
    public void feedPet() {
        System.out.println("Elder is overfeeding the pet.");
    }
}
```

At this point, we have the adequate classes to create an `AbstractFactory` as well as the respective *Factory* classes for these two groups: `PetFactory` and `HumanFactory`.

The `AbstractFactory`'s concern is the ability to provide these objects to the `FactoryProducer`, not to instantiate them:

```
public abstract class AbstractFactory {
   public abstract Pet getPet(String pet);
   public abstract Human getHuman(String human);
}
```

Before we define the class that instantiates these objects using the `AbstractFactory`, we need to create our two factories.

```
public class HumanFactory extends AbstractFactory {

   @Override
   Human getHuman(String human) {
      if(human.equals(null)) return null;

      if(human.equalsIgnoreCase("chILd")) {
         return new Child();
      } else if(human.equalsIgnoreCase("adult")) {
         return new Adult();
      } else if(human.equalsIgnoreCase("elDeR")) {
         return new Elder();
      }
      return null;
   }

   @Override
   Pet getPet(String pet) {
      // don't implement
      return null;
   }
```

```
public class PetFactory extends AbstractFactory {

    @Override
    public Pet getPet(String pet) {
        if(pet.equals(null)) return null;

        if(pet.equalsIgnoreCase("Dog")) {
            return new Dog();
        } else if(pet.equalsIgnoreCase("Cat")) {
            return new Cat();
        } else if(pet.equalsIgnoreCase("Rabbit")) {
            return new Rabbit();
        }
        return null;
    }

    @Override
    Human getHuman(String human) {
        //don't implement
        return null;
    }
}
```

And now, with these, we can create the `FactoryProducer` which is charged with the responsibility to instantiate the adequate factories, with the help of the `AbstractFactory`:

```
public class FactoryProducer {
    public static AbstractFactory getFactory(String factory) {
        if(factory.equalsIgnoreCase("Human")) {
            return new HumanFactory();
        } else if(factory.equalsIgnoreCase("Pet")) {
            return new PetFactory();
        }
        return null;
    }
}
```

By passing a `String`, the `FactoryProducer` returns the `AbstractFactory` with their requested child factory.

Now, let's observe the result:

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

      AbstractFactory humanFactory = FactoryProducer.getFactory("Human");
      AbstractFactory petFactory = FactoryProducer.getFactory("Pet");

      Human human = humanFactory.getHuman("Child");
      human.feedPet();

      Pet pet = petFactory.getPet("Dog");
      pet.eat();

      Human human2 = humanFactory.getHuman("Elder");
      human2.feedPet();

      Pet pet2 = petFactory.getPet("Rabbit");
      pet2.eat();
   }
}
```

By running this piece of code, we are greeted with:

```
Child is feeding pet irresponsibly.
Dog is eating, woof!
Elder is overfeeding the pet.
Rabbit is eating, squeak!
```

Builder

The Builder pattern is used to help build final objects, for classes with a huge amount of fields or parameters in a step-by-step manner. It's not very useful in small, simple classes that don't have many fields, but complex objects are both hard to read and maintain by themselves.

Initializing an object with more than a few fields using a constructor is messy and susceptible to human error.

Implementation:

Let's define a class with a few fields:

```
public class Computer {
   private String computerCase;
   private String CPU;
   private String motherboard;
   private String GPU;
   private String HDD;
   private String operatingSystem;
   private int powerSupply;
   private int amountOfRAM;

   public Computer(String computerCase, String CPU, String motherboard, String GPU,
   String HDD, String operatingSystem, int powerSupply, int amountOfRAM) {
      this.computerCase = computerCase;
      this.CPU = CPU;
      this.motherboard = motherboard;
      this.GPU = GPU;
      this.HDD = HDD;
      this.operatingSystem = operatingSystem;
      this.powerSupply = powerSupply;
      this.amountOfRAM = amountOfRAM;
  }

   //getters and setters
}
```

The problem is evident - Even a small, simple class like this requires a big and messy constructor.

Classes can easily have considerably more fields than this, which gave birth to the Builder design pattern.

To apply it, we'll nest a `static Builder` class within the `Computer` class.

This builder will be used to build our objects in a clean and readable way, unlike the example above:

```
public class Computer {

  public static class Builder {
     private String computerCase;
     private String CPU;
     private String motherboard;
     private String GPU;
     private String HDD;
     private String operatingSystem;
     private int powerSupply;
     private int amountOfRAM;

     public Builder withCase(String computerCase) {
        this.computerCase = computerCase;
        return this;
      }
```

```
    public Builder withCPU(String CPU) {
        this.CPU = CPU;
        return this;
    }

    public Builder withMotherboard(String motherboard) {
        this.motherboard = motherboard;
        return this;
    }

    public Builder withGPU(String GPU) {
        this.GPU = GPU;
        return this;
    }

    public Builder withHDD(String HDD) {
        this.HDD = HDD;
        return this;
    }

    public Builder withOperatingSystem(String operatingSystem) {
        this.operatingSystem = operatingSystem;
        return this;
    }

    public Builder withPowerSupply(int powerSupply) {
        this.powerSupply = powerSupply;
        return this;
    }

    public Builder withAmountOfRam(int amountOfRAM) {
        this.amountOfRAM = amountOfRAM;
        return this;
    }

    public Computer build() {
        Computer computer = new Computer();
        computer.computerCase = this.computerCase;
        computer.CPU = this.CPU;
        computer.motherboard = this.motherboard;
        computer.GPU = this.GPU;
        computer.HDD = this.HDD;
        computer.operatingSystem = this.operatingSystem;
        computer.powerSupply = this.powerSupply;
        computer.amountOfRAM = this.amountOfRAM;

        return computer;
    }
}

private Computer() {
   //nothing here
}
```

```
    //fields
    //getters and setters
}
```

This nested class has the same fields as the `Computer` class and uses them to build the object itself.

The `Computer` constructor is made private so that the only way to initialize it is via the `Builder` class.

With the `Builder` all set-up, we can initialize `Computer` objects:

```
public class Main {
    public static void main(String[] args) {
        Computer computer = new Computer.Builder()
                .withCase("Tower")
                .withCPU("Intel i5")
                .withMotherboard("MSI B360M-MORTAR")
                .withGPU("nVidia Geforce GTX 750ti")
                .withHDD("Toshiba 1TB")
                .withOperatingSystem("Windows 10")
                .withPowerSupply(500)
                .withAmountOfRam(8)
                .build();
    }
}
```

This is a much cleaner and more verbose way than writing:

```
public class Main {
    public static void main(String[] args) {
        Computer computer = new Computer("Tower", "Intel i5", "MSI B360M-MORTAR",
        "nVidia GeForce GTX 750ti, "Toshiba 1TB", "Windows 10", 500, 8);
    }
}
```

If you'd like to read a standalone, detailed article on The Builder Design Pattern, we've got you covered!

Prototype

The Prototype pattern is used mainly to minimize the cost of object creation, usually when large-scale applications create, update or retrieve objects which cost a lot of resources.

This is done by copying the object, once it's created, and reusing the copy of the object in later requests, to avoid performing another resource-heavy operation. It depends on the decision of the developer whether this will be a full or shallow copy of the object, though the goal is the same.

Implementation:

Since this pattern clones objects, it would be fitting to define a class for them:

```
// to clone the object, the class needs to implement Cloneable
public abstract class Employee implements Cloneable {

    private String id;
    protected String position;
    private String name;
    private String address;
    private double wage;

    abstract void work();

    public Object clone() {
        Object clone = null;
        try {
            clone = super.clone();
        } catch(CloneNotSupportedException ex) {
            ex.printStackTrace();
        }
        return clone;
    }
   //getters and setters
}
```

Now, as usual, let's define a few classes that extend `Employee`:

```
public class Programmer extends Employee {
    public Programmer() {
        position = "Senior";
    }
    @Override
    void work() {
        System.out.println("Writing code!");
    }
}

public class Janitor extends Employee {
    public Janitor() {
        position = "Part-time";
    }
    @Override
    void work() {
        System.out.println("Cleaning the hallway!");
    }
}

public class Manager extends Employee {
    public Manager() {
        position = "Intern";
    }
    @Override
    void work() {
        System.out.println("Writing a schedule for the project!");
    }
}
```

At this point, we have everything we need for a class from a data layer to save, update and retrieve these employees for us.

A `Hashtable` will be used to simulate a database, and predefined objects will simulate objects retrieved via queries:

```
public class EmployeesHashtable {

   private static Hashtable<String, Employee> employeeMap = new Hashtable<String,
Employee>();

   public static Employee getEmployee(String id) {
      Employee cacheEmployee = employeeMap.get(id);
      // a cast is needed because the clone() method returns an Object
      return (Employee) cacheEmployee.clone();
   }

   public static void loadCache() {
      // predefined objects to simulate retrieved objects from the database
      Programmer programmer = new Programmer();
      programmer.setId("ETPN1");
      employeeMap.put(programmer.getId(), programmer);

      Janitor janitor = new Janitor();
      janitor.setId("ETJN1");
      employeeMap.put(janitor.getId(), janitor);

      Manager manager = new Manager();
      manager.setId("ETMN1");
      employeeMap.put(manager.getId(), manager);
   }
}
```

To observe the result:

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

      Employee cloned1 = (Employee) EmployeesHashtable.getEmployee("ETPN1");
      Employee cloned2 = (Employee) EmployeesHashtable.getEmployee("ETJN1");
      Employee cloned3 = (Employee) EmployeesHashtable.getEmployee("ETMN1");

      System.out.println("Employee: " + cloned1.getPosition() + " ID:"
         + cloned1.getId());
      System.out.println("Employee: " + cloned2.getPosition() + " ID:"
         + cloned2.getId());
      System.out.println("Employee: " + cloned3.getPosition() + " ID:"
         + cloned3.getId());
   }
}
```

Running this piece of code will yield:

```
Employee: Senior ID:ETPN1
Employee: Part-time ID:ETJN1
Employee: Intern ID:ETMN1
```

Singleton

The Singleton pattern ensures the existence of only one object instance in the whole JVM.

This is a rather simple pattern and it provides the ability to access this object even without instantiating it. Other design patterns use this pattern, like the Abstract Factory, Builder, and Prototype patterns we've already covered.

Implementation:

This is a fairly simple implementation of a *Singleton* class:

```
public class SingletonClass {

    private static SingletonClass instance = new SingletonClass();

    private SingletonClass() {}

    public static SingletonClass getInstance() {
        return instance;
    }

    public void showMessage() {
        System.out.println("I'm a singleton object!");
    }
}
```

This class is creating a static object of itself, which represents the global instance.

By providing a private constructor, the class cannot be instantiated.

A static method `getInstance()` is used as a global access point for the rest of the application.

Any number of public methods can be added to this class, but there's no need to do so for this tutorial.

With this, our class fulfills all requirements to become a *Singleton*.

Let's define some code that retrieves this object and runs a method:

```
public class Main {
    public static void main(String[] args) {
        SingletonClass singletonClass = SingletonClass.getInstance();
        singletonClass.showMessage();
    }
}
```

Running this code will result in:

```
I'm a singleton object!
```

Conclusion

With this, all **Creational Design Patterns** in Java are fully covered, with working examples.

If you'd like to continue reading about Design Patterns in Java, the following article covers Structural Design Patterns.

About David Landup
Serbia Twitter Website

Get occassional tutorials, guides, and jobs in your inbox. No spam ever. Unsubscribe at any time.