Design Pattern: Factory

The Factory design pattern solves the problem of creating objects without knowing the concrete implementation or needing to change it when your program is running. This is done by adding a wrapper class in the "new" keyword. The Factory pattern solves this problem by encapsulating the creation of objects in a separate class (the Factory class), which knows how to create objects and/or different types. This way, the client code can request an object from the Factory without knowing which specific class is being instantiated, and the Factory takes care of creating the object which also requires business logic to know which object to create. Returning it to the client. This is a creation pattern.

Factory lets the class defer instantiation to subclasses.

To be able to use the Factory pattern, the objects that the Factory creates **must** implement a common interface.

This also allows you to hide the creation of the object from the client. The client only has to call the factory class and the factory class will create the object for you. If the constructor of your object is protected or private, the client will not be able to create the object directly. The factory class will be the only one that can create the object.

For example if you have an animal interface and you have a dog implementation, a cat implementation, and a duck implementation and you want the animals to randomly be created. This can be done by having a factory that generates a random number and based on that number it will create a dog, cat, or duck, in runtime.

Without Factory Pattern

e.g.

```
package org.example;

public abstract class Animals {
   private String name;
   private int age;

   public int getAge() {
       return age;
   }

   public String getName() {
       return name;
   }

   public void setAge(int age) {
       this.age = age;
   }

   public void setName(String name) {
```

```
        this.name = name;
    }

    public void makeSound() {
        System.out.println( getName() + " is making a sound");
    }
}
```

```
package org.example;

public class Dog extends Animals{
    public Dog() {
        setName("Dog");
        setAge(22);
    }
}
```

```
package org.example;

public class Duck extends Animals{
    public Duck(){
        setName("Duck");
        setAge(2);
    }
}
```

```
package org.example;

public class Cat extends Animals{
    public Cat() {
        setName("Cat");
        setAge(21);
    }
}
```

```
package org.example;

import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Animals  animal;

        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter the name of the animal: ");
```

```
        if (scanner.hasNextLine()){
            String name = scanner.nextLine();
            if (name.equals("Duck")){
                animal = new Duck();
                System.out.println("Name: " + animal.getName());
                System.out.println("Age: " + animal.getAge());
            } else if (name.equals("Dog")){
                animal = new Dog();
                System.out.println("Name: " + animal.getName());
                System.out.println("Age: " + animal.getAge());
            } else {
                System.out.println("Animal not found");
            }
        }
    }
}
```

With Factory Pattern

```
public interface Factory {
    Animals createAnimal( String animalName );
}
```

```
package org.example;

public class AnimalFactory implements Factory{
    @Override
    public Animals createAnimal(String animalName) {
        if (animalName.equals("Duck")){
            return new Duck();
        } else if (animalName.equals("Dog")){
            return new Dog();
        } else {
            return null;
        }
    }
}
```

```
package org.example;

import java.util.Scanner;

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

```
        Animals animal;

        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter the name of the animal: ");

        if (scanner.hasNextLine()){
            String name = scanner.nextLine();
            animal = factory.createAnimal(name);
            animal.makeSound();
        }
    }
}
```

OOP Principles

The Factory design pattern conforms to the four pillars of Object-Oriented Programming (OOP) in the following ways:

Encapsulation: The Factory pattern encapsulates the object creation process in a separate class or method, which provides a clear interface for creating objects. This helps to hide the implementation details of object creation from the client code and promotes information hiding.

Abstraction: The Factory pattern provides an abstract interface for creating objects, which allows the client code to work with objects at a higher level of abstraction. The client code does not need to know the details of how objects are created, only that they can be created using the Factory.

Inheritance: The Factory pattern can take advantage of inheritance by using subclassing to provide different implementations of the object creation process. This allows for greater flexibility in the object creation process and promotes code reuse, which is a key aspect of inheritance.

Polymorphism: The Factory pattern uses polymorphism to allow the client code to work with objects of different types through a common interface. The client code does not need to know the specific type of object being created, only that it conforms to a common interface provided by the Factory.

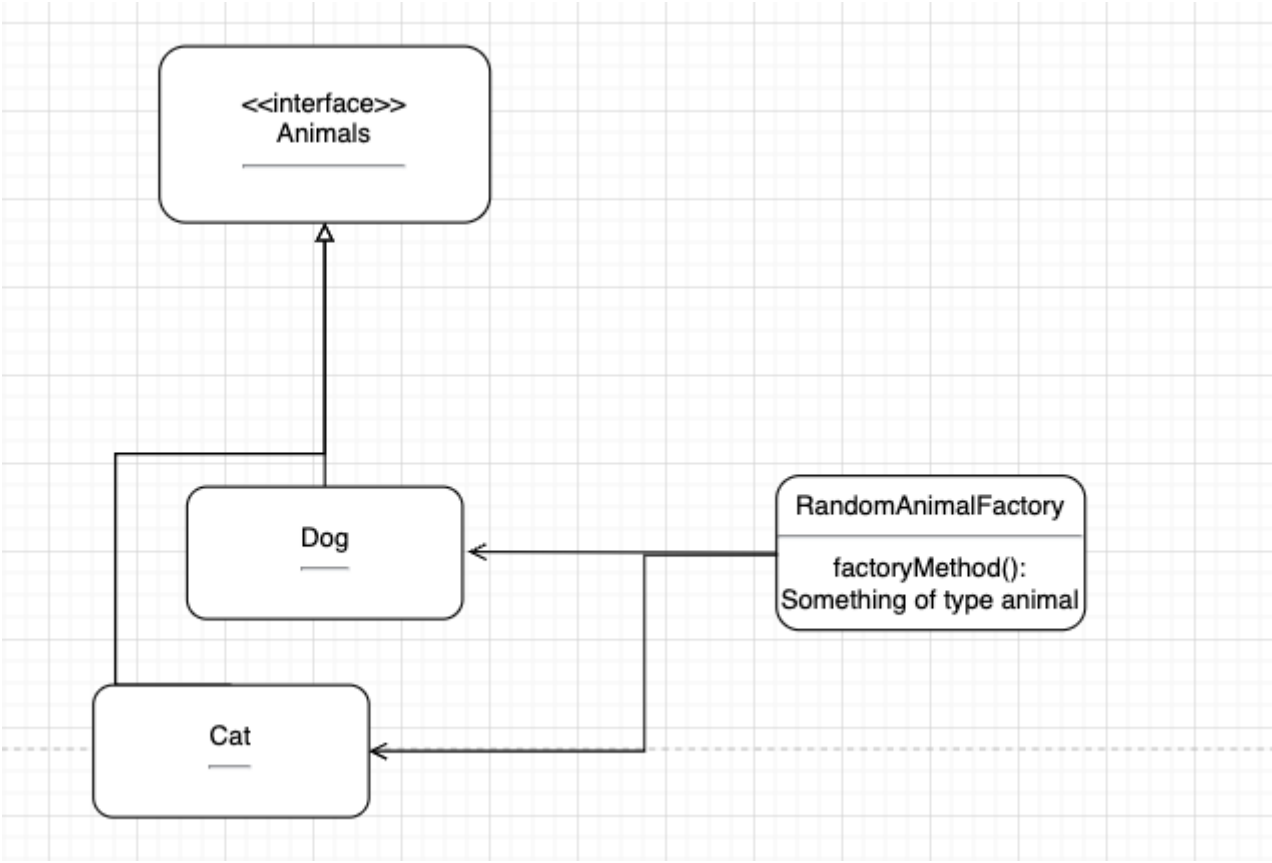
Types of Factory

There are three types of Factory patterns:

1. Simple Factory - The Simple Factory pattern is the most basic type of Factory pattern. It encapsulates the object creation process in a separate class or method, which provides a clear interface for creating objects. This has a single concrete factory. Some programmers consider this not to be a design pattern because it does fit some characteristics of a design pattern. For example, it is not reusable because there is only one way to make an object.

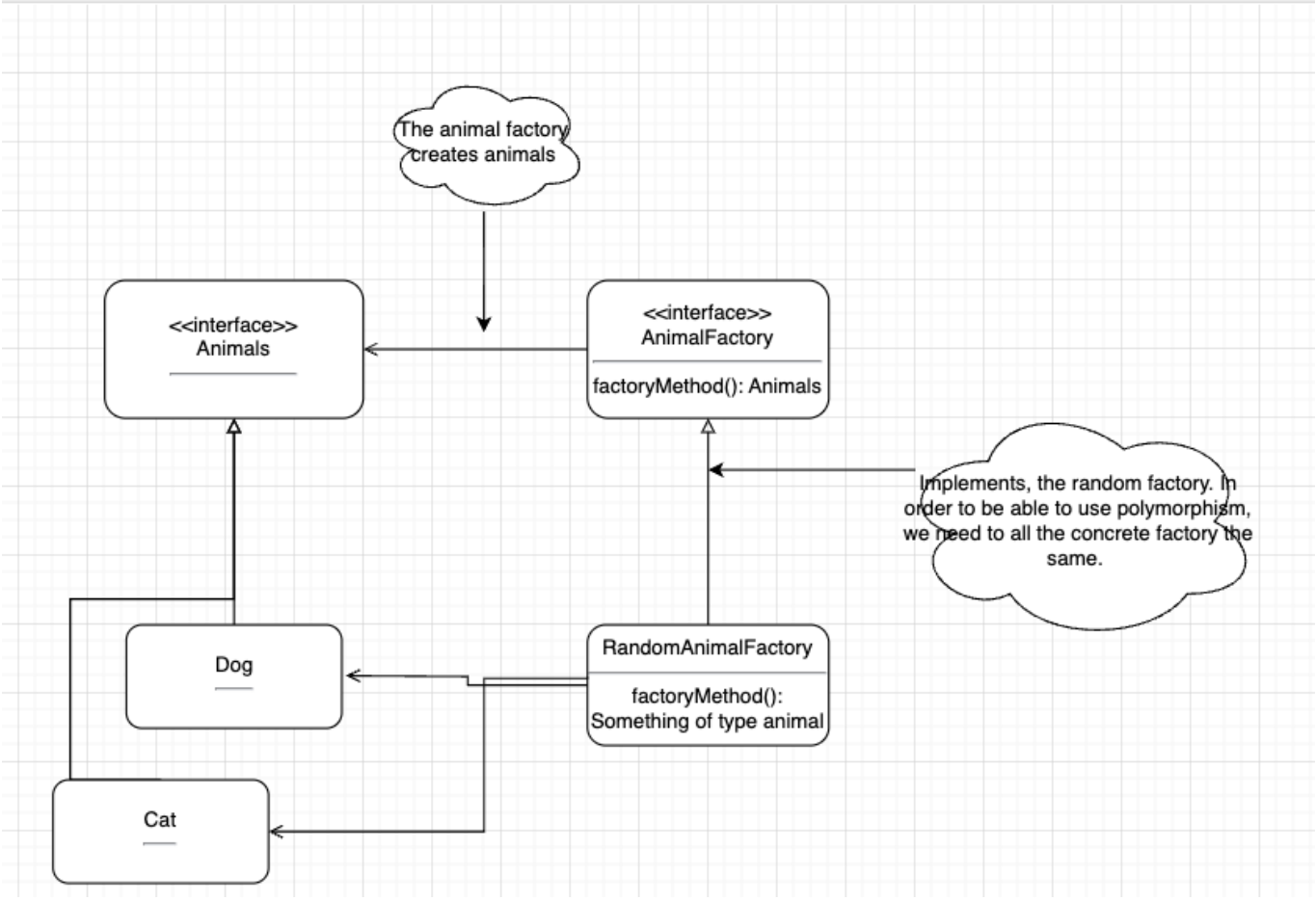
```
public class ShapeFactory {
    //use getShape method to get object of type shape
    public Shape getShape(String shapeType){
        if(shapeType == null){
```

```
            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;
    }
}
```



---

2. Factory Method - This is the same as the simple factory but it has multiple concrete factories by having a shared 'factory' interface, this is the most common type of factory, and each factory is used to create the same type of object, but with a different business logic.



3. Abstract Factory - This is the same as the factory method but has the capability to create different types of objects, this is the most complex type of factory. It leaves the type of object to create to the concrete factories. So you can have a different type that a certain factory makes.

The Abstract factory pattern is most used for UI, for example, if you have a button, you can have a button factory that creates a button for each operating system, (Windows, Mac, Linux) and each button will have a different look and feel.