

Professor Hans Noodles

41 уровень

07.11.2019 41865 28

Паттерн проектирования Factory

Статья из группы Java Developer

43656 участников

Вы в группе

Привет, друг! Сегодня мы продолжим изучать с тобой паттерны проектирования. В этой лекции будем говорить о Фабрике. Обсудим с тобой, какую проблему решают с помощью данного шаблона, посмотрим на примере, как фабрика помогает открывать кофейню. А еще я я дам тебе 5 простых шагов для создания фабрики.



Чтобы быть со всеми на одной волне и легко улавливать суть, тебе должны быть знакомы такие темы:

- Наследование в Java
- Сужение и расширение ссылочных типов в Java
- Взаимодействие между различными классами и объектами

Что такое Фабрика?

Шаблон проектирования Фабрика позволяет управлять созданием объектов.

Процесс создания нового объекта не то чтобы прост, но и не слишком сложен. Все мы знаем, что для создания нового объекта необходимо использовать оператор `new`. И может показаться, что здесь нечем управлять, однако это не так.

Сложности могут возникнуть, когда в нашем приложении есть некоторый класс, у которого есть множество наследников, и необходимо создавать экземпляр определенного класса в зависимости от некоторых условий.

Фабрика — это шаблон проектирования, который помогает решить проблему создания различных объектов в зависимости от

НАЧАТЬ ОБУЧЕНИЕ

Абстрактно, не правда ли? Больше конкретики и ясности появится, когда мы рассмотрим пример ниже.

Создаем различные виды кофе

Предположим, мы хотим автоматизировать кофейню. Нам необходимо научиться готовить различные виды кофе. Для этого в нашем приложении мы создадим класс кофе и его производные: американо, капучино, эспрессо, латте — те виды кофе, которые мы будем готовить.

Начнем с общего кофейного класса:

```
1 public class Coffee {
2     public void grindCoffee(){
3         // перемалываем кофе
4     }
5     public void makeCoffee(){
6         // делаем кофе
7     }
8     public void pourIntoCup(){
9         // наливаем в чашку
10    }
11 }
```

Далее создадим его наследников:

```
1 public class Americano extends Coffee {}
2 public class Cappuccino extends Coffee {}
3 public class CaffeLatte extends Coffee {}
4 public class Espresso extends Coffee {}
```

Наши клиенты будут заказывать какой-либо вид кофе, и эту информацию нужно передавать программе. Это можно сделать разными способами, например использовать `String`. Но лучше всего для этих целей подойдет `enum`. Создадим `enum` и определим в нем типы кофе, на которые мы принимаем заказы:

```
1 public enum CoffeeType {
2     ESPRESSO,
3     AMERICANO,
4     CAFFE_LATTE,
5     CAPPUCCINO
6 }
```

Отлично, теперь напомним код нашей кофейни:

```
1 public class CoffeeShop {
2
3     public Coffee orderCoffee(CoffeeType type) {
4         Coffee coffee = null;
5
6         switch (type) {
7             case AMERICANO:
8                 coffee = new Americano();
9                 break;
```

```
12         break;
13     case CAPPUCCINO:
14         coffee = new Cappuccino();
15         break;
16     case CAFFE_LATTE:
17         coffee = new CaffeLatte();
18         break;
19     }
20
21     coffee.grindCoffee();
22     coffee.makeCoffee();
23     coffee.pourIntoCup();
24
25     System.out.println("Вот ваш кофе! Спасибо, приходите еще!");
26     return coffee;
27 }
28 }
```

Метод `orderCoffee` можно разделить на две составляющие:

1. Создание конкретного экземпляра кофе в блоке `switch-case`. Именно здесь происходит то, что делает Фабрика — создание конкретного типа в зависимости от условий.
2. Само приготовление — перемолка, приготовление и разлитие в чашку.

Что важно знать, если нужно будет вносить в метод изменения в будущем:

1. Сам алгоритм приготовления (перемолка, приготовление и разлитие в чашку) останется неизменным (по крайней мере мы на это рассчитываем).
2. А вот ассортимент кофе может измениться. Возможно, мы начнем готовить мока.. Мокка.. Моккачи... Господь с ним, новый вид кофе.

Мы уже сейчас можем предположить, что в будущем, с определенной долей вероятности, нам придется вносить изменения в метод, в блок `switch-case`.

Также возможно, в нашей кофейне метод `orderCoffee` будет не единственным местом, в котором мы будем создавать различные виды кофе. Следовательно, вносить изменения придется в нескольких местах.

Тебе уже наверняка понятно, к чему я клоню. Нам нужно рефакторить. Вынести блок, отвечающий за создание кофе, в отдельный класс по двум причинам:

1. Мы сможем переиспользовать логику создания кофе в других местах.
2. Если ассортимент изменится, нам не придется править код везде, где будет использоваться создание кофе. Достаточно будет изменить код только в одном месте.

Иными словами, пришло время запилить фабрику.

Пилим нашу первую фабрику

Для этого создадим новый класс, который будет отвечать только за создание нужных экземпляров классов кофе:

```
1 public class SimpleCoffeeFactory {
2     public Coffee createCoffee (CoffeeType type) {
3         Coffee coffee = null;
4
5         switch (type) {
6             case AMERICANO:
7                 coffee = new Americano();
8                 break;
9             case CAPPUCCINO:
10                 coffee = new Cappuccino();
11                 break;
12             case CAFFE_LATTE:
13                 coffee = new CaffeLatte();
14                 break;
15             default:
16                 throw new IllegalArgumentException("Неизвестный тип кофе");
17         }
18         return coffee;
19     }
20 }
```

```
9         case ESPRESSO:
10             coffee = new Espresso();
11             break;
12         case CAPPUCCINO:
13             coffee = new Cappuccino();
14             break;
15         case CAFFE_LATTE:
16             coffee = new CaffeLatte();
17             break;
18     }
19
20     return coffee;
21 }
22 }
```

Поздравляю тебя! Мы только что реализовали шаблон проектирования Фабрика в самом его простейшем виде.

Хотя все могло быть еще проще, если сделать метод `createCoffee` статичным. Но тогда мы потеряли бы две возможности:

- 1. Наследоваться от `SimpleCoffeeFactory` и переопределять метод `createCoffee`.
- 2. Внедрять нужную реализацию фабрики в наши классы.

Кстати о внедрении. Нам нужно вернуться в кофейню и внедрить нашу фабрику по созданию кофе.

Внедрение фабрики в кофейню

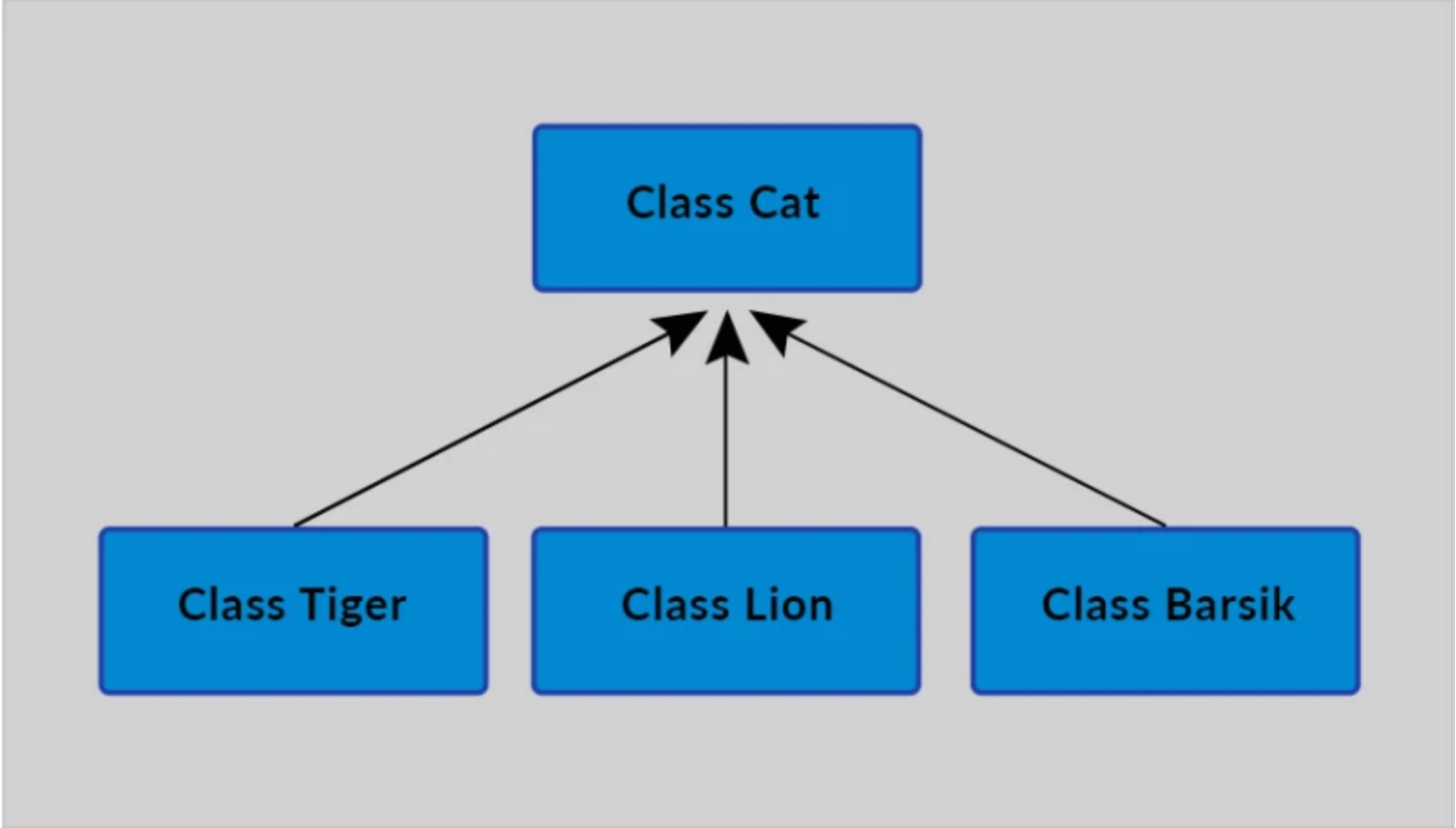
Перепишем класс нашей кофейни с использованием фабрики:

```
1 public class CoffeeShop {
2
3     private final SimpleCoffeeFactory coffeeFactory;
4
5     public CoffeeShop(SimpleCoffeeFactory coffeeFactory) {
6         this.coffeeFactory = coffeeFactory;
7     }
8
9     public Coffee orderCoffee(CoffeeType type) {
10         Coffee coffee = coffeeFactory.createCoffee(type);
11         coffee.grindCoffee();
12         coffee.makeCoffee();
13         coffee.pourIntoCup();
14
15         System.out.println("Вот ваш кофе! Спасибо, приходите еще!");
16         return coffee;
17     }
18 }
```

Отлично. Теперь схематично и лаконично попробуем описать структуру шаблона проектирования Фабрика.

5 шагов к открытию собственной фабрики

Шаг 1. У тебя в программе класс с несколькими потомками, как на картинке ниже:



Шаг 2. Ты создаешь `enum`, в котором определяешь enum-переменную для каждого класса-наследника:

```
1  enum CatType {
2      LION,
3      TIGER,
4      BARSIK
5  }
```

Шаг 3. Ты строишь свою фабрику. Называешь её `MyClassFactory`, код ниже:

```
1  class CatFactory {}
```

Шаг 4. Ты создаешь в своей фабрике метод `createMyClass`, который принимает в себя переменную-`enum` `MyClassType`. Код ниже:

```
1  class CatFactory {
2      public Cat createCat(CatType type) {
3
4      }
5  }
```

Шаг 5. Ты пишешь в теле метода блок `switch-case`, в котором перебираешь все enum значения и создаешь экземпляр класса, соответствующий `enum` значению:

```
1  class CatFactory {
2      public Cat createCat(CatType type) {
3          Cat cat = null;
4
5          switch (type) {
6              case LION:
7                  cat = new Barsik();
8                  break;
9              case TIGER:
10                 cat = new Tiger();
```

```

13         cat = new Lion();
14         break;
15     }
16
17     return cat;
18 }
19 }

```

Like a boss.

Научитесь программировать с нуля с JavaRush:
1200 задач, автопроверка решения и стиля кода

[НАЧАТЬ ОБУЧЕНИЕ](#)

Как тренироваться

Читать — хорошо, писать код — еще лучше. Если в твоём имени чётное количество букв, попробуй создать свою виртуальную пиццерию.

Если в твоём имени нечетное количество букв, попробуй создать виртуальный суши-бар.

+169

Комментарии (28)

популярные

новые

старые

JavaCoder

Введите текст комментария

Алексей Куренков

Java Developer в InnoTech

26 августа, 17:04

Я понимаю что в 1м примере с кофе нужно разделить создание экземпляров от приготовления кофе. Мне не понятно, а не проще ли просто было создать метод createCoffee в этом же классе CoffeeShop, а не городить дополнительный класс? Я не настаиваю что я прав, но хочу понять что бы разобраться.

```
public class CoffeeShop {  
  
    public Coffee createCoffee (CoffeeType type) {  
        Coffee coffee = null;  
  
        switch (type) {  
            case AMERICANO:  
                coffee = new Americano();  
                break;  
            case ESPRESSO:  
                coffee = new Espresso();  
                break;  
            case CAPPUCCINO:  
                coffee = new Cappuccino();  
                break;  
            case CAFFE_LATTE:  
                coffee = new CaffeLatte();  
                break;  
        }  
  
        return coffee;  
    }  
  
    public Coffee orderCoffee(CoffeeType type) {  
        /*Coffee coffee = null;  
  
        switch (type) {
```

НАЧАТЬ ОБУЧЕНИЕ

```
        case ESPRESSO:
            coffee = new Espresso();
            break;
        case CAPPUCCINO:
            coffee = new Cappuccino();
            break;
        case CAFFE_LATTE:
            coffee = new CaffeLatte();
            break;
    }*/
    Coffee coffee = createCoffee(type);

    coffee.grindCoffee();
    coffee.makeCoffee();
    coffee.pourIntoCup();

    System.out.println("Вот ваш кофе! Спасибо, приходите еще!");
    return coffee;
}
}
```

Ответить

0

Anonymous #3090867 Уровень 1, Минск, Belarus

2 августа, 15:05

Вы на приколе

```
case BARSIK:
    cat = new Lion();
    break;
}
```

Ответить

0

Art09 Уровень 35, Пермь, Россия

25 мая, 07:32

У данного примера с кофе, мы все также не решаем проблему масштабирования. Если у нас появляется новый вид кофе, то нам нужно вносить изменения в 2 класса SimpleCoffeeFactory и CoffeeType. "Если ассортимент изменится, нам не придется править код везде, где будет использоваться создание кофе. Достаточно будет изменить код только в одном месте." - остается проблемой!!!

Ответить

0

Руслан Уровень 22, Уральск

23 мая, 08:04

А как это дело запустить то? В мэйне мы же не можем создать экземпляр кофешоп... Статик сделать тоже не получается.
Как вывести то в консоль что кофе готово?

Ответить

0

Art09 Уровень 35, Пермь, Россия

25 мая, 07:25

```
Создаем Магазин в кот. мы продаем кофе, также есть у нас фабрика по созданию разного кофе.
Далее помещаем в ссылку созданный кофе и отдаем клиенту:
CoffeeShop coffeeShop = new CoffeeShop(new SimpleCoffeeFactory());
Coffee coffee = coffeeShop.orderCoffee(CoffeeType.AMERICANO);
```

Ответить

0

Ян Уровень 41, Лида, Беларусь

8 января, 19:28

1

Хотя все могло быть еще проще, если сделать метод createCoffee статичным.

2

Но тогда мы потеряли бы две возможности:

3

1. Наследоваться от SimpleCoffeeFactory и переопределять метод createCoffee

4

.....

Почему не будет возможности наследоваться и тем более переопределять метод, если createCoffe будет с модификатором static?

Ответить

0

Максим Караваев Java Developer в Deutsche Telekom IT

14 января, 13:00

Потому что статические методы не переопределяются. Т.к. это метод класса, а не экземпляра. Можно создать в наследнике статический метод с такой же сигнатурой, но это будет не переопределённый метод родителя, а просто метод другого класса с такой же сигнатурой.

Ответить

+1

Ян Уровень 41, Лида, Беларусь

14 января, 19:21

благодарю, стало понятнее. раньше думал, что переопределять методы нельзя только с модификатором final...

Ответить

0

alex Уровень 41, Россия

2 июня 2021, 11:21

Использовать switch тоже не совсем хорошая практика. Это претензия не к автору, почему то во всех

Maks Panteleev

Java Developer в Bell Integrator

26 июля 2021, 14:08

⋮

а что используют? иф?

Ответить

−

+2

+

Денис Гарбуз

Уровень 1, Ukraine

4 июня, 14:46

⋮

Автор коммента не встречал в проектах свич, потому что он нее работал ни на каких проектах

Ответить

−

0

+

Romanya

Java Developer в Региональный Информа

30 мая 2021, 12:20

⋮

Статья класс! Спасибо автору.

Ответить

−

0

+

Valua Sinicyn

Уровень 41, Харьков, Украина

23 февраля 2021, 20:41

⋮

[Фабрика](#) на человеческом.

Ответить

−

+4

+

Azat

Уровень 41, Казань

15 сентября 2020, 19:11

⋮

Интересно, как реализовать фабрику без свитча. Использовать свитч, насколько могу судить, не лучшая практика

Ответить

−

+2

+

Yaroslav Katrushka

Уровень 18

7 августа 2020, 15:28

⋮

Хорошая статья. Доступно и полезно

Ответить

−

+4

+

↺

Показать еще комментарии

ОБУЧЕНИЕ

- Курсы программирования
- Курс Java
- Помощь по задачам
- Подписки
- Задачи-игры

СООБЩЕСТВО

- Пользователи
- Статьи
- Форум
- Чат
- Истории успеха
- Активности

КОМПАНИЯ

- О нас
- Контакты
- Отзывы
- FAQ
- Поддержка



JavaRush — это интерактивный онлайн-курс по изучению Java-программирования с нуля. Он содержит 1200 практических задач с проверкой решения в один клик, необходимый минимум теории по основам Java и мотивирующие фишки, которые помогут пройти курс до конца: игры, опросы, интересные проекты и статьи об эффективном обучении и карьере Java-девелопера.

ПОДПИСЫВАЙТЕСЬ

ЯЗЫК ИНТЕРФЕЙСА

Русский

▼

НАЧАТЬ ОБУЧЕНИЕ

