**Зачем нужен паттерн Factory (Фабрика)?**

Давайте представим, что нам нужен класс, который создает объекты разного типа. Иногда это очень удобно. Почему?

Во-первых, представим, что нам нужна "фабрика", которая выпускает разные пончики:

* пончик с вишней
* пончик с белым шоколадом
* пончик с миндалем

Соответственно, под **каждый пончик** должен быть **свой класс.**

Однако все эти классы очень похожи. Все пончики будут иметь одинаковый набор параметров:

* вес
* калорийность
* дату изготовления
* начинку
* и т.д.

Поскольку эти классы пончиков очень похожи, создавать каждый экземпляр класса вручную будет достаточно громоздко и к тому же они будут практически одинаковыми. Поэтому мы можем создать специальный объект, который будет уметь делать пончик, **зная его начинку**.

То есть мы сможем просто сказать:

* "Создай пончик **с вишней**"
* "Создай пончик **с белым шоколадом**"
* "Создай пончик **с миндалем**"

Для этого нам необходимо использовать **паттерн Factory.**Ну, что ж, давайте попробуем создать нашу первую "фабрику" (Factory). Давайте назовем ее "Фабрика пончиков" (DoughnutFactory):

|  |  |
| --- | --- |
| 1  2  3 | public class DoughnutFactory {    } |

Для того, чтобы мы могли их производить на одной фабрике, нужно "указать, что все эти пончики имеют **один тип**" - а именно, "пончик". Давайте объединим их, создав общий для всех интерфейс **Doughnut**:

|  |  |
| --- | --- |
| 1  2  3 | public interface Doughnut {      void eat();  } |

Отлично! Теперь давайте создадим три класса пончиков, которые имплементиуют интерфейс **Doughnut**.

Вишневый:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class CherryDoughnut implements Doughnut {      @Override      public void eat() {          System.out.println("You are eating Cherry doughnut!");      }  } |

Шоколадный:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class ChocolateDoughnut implements Doughnut {      @Override      public void eat() {          System.out.println("You are eating Chocolate doughnut!");      }  } |

И миндальный:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class AlmondDoughnut implements Doughnut {      @Override      public void eat() {          System.out.println("You are eating Almond doughnut!");      }  } |

Отлично - классы есть, интерфейс есть. Давайте производить!

Итак, суть фабрики сводится к тому, чтобы производить объекты **разного типа**. Но как сделать так, чтоб Фабрика понимала какой именно тип объекта(пончика) мы хотим получить: с вишней, с белым шоколадом или с миндалем?

Для этого давайте создадим ENUM, в котором запишем все возможные типы пончиков:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public enum DoughnutTypes {      CHERRY,      CHOCOLATE,      ALMOND  } |

Конечно, мы могли бы использовать вместо ENUM строку String с названием, но тогда ошибок гораздо сложнее избежать. Если исключить опечатки, то пришлось бы запоминать, в каком формате вводить строку - "Chocolate", "CHOCOLATE" или "ChocolateDoughnut"? В общем, все эти проблемы нам не нужны, поэтому мы создали ENUM.

Таким образом, наша фабрика будет принимать название объекта (в нашем случае ENUM), и возвращать объект нужного типа. Прототип такой функции будет выглядеть следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class DoughnutFactory {        public Doughnut getDoughnut(DoughnutTypes type) {        }  } |

Как видите, принимается ENUM **DoughnutTypes**, а возвращается любой объект, имплементирующий интерфейс  **Doughnut**. Таким образом, мы теперь можем "конструировать" любой пончик и возвращать его пользователю.

Итак, перейдем к сути. Определяем тип с помощью обычного**switch... case**:

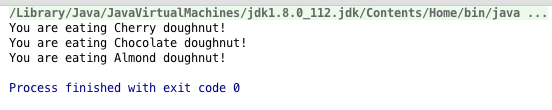
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class DoughnutFactory {        public Doughnut getDoughnut(DoughnutTypes type) {          Doughnut toReturn = null;          switch (type) {              case CHERRY:                  toReturn = new CherryDoughnut();                  break;              case CHOCOLATE:                  toReturn = new ChocolateDoughnut();                  break;              case ALMOND:                  toReturn = new AlmondDoughnut();                  break;              default:                  throw new IllegalArgumentException("Wrong doughnut type:" + type);          }          return toReturn;      }  } |

Как Вы видите, мы создаем и возвращаем новый объект нужного типа. Если будет введено не CHERRY, не CHOCOLATE и не ALMOND (а это может быть только **null**), тогда мы показываем ошибку.

Хорошо, теперь протестируем наш код. Создадим **main()**, в котором создадим все типы поисков по очереди и вызовем на них метод **eat()**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class Main {      public static void main(String[] args) {          DoughnutFactory factory = new DoughnutFactory();            Doughnut cherry = factory.getDoughnut(DoughnutTypes.CHERRY);          Doughnut chocolate = factory.getDoughnut(DoughnutTypes.CHOCOLATE);          Doughnut almond = factory.getDoughnut(DoughnutTypes.ALMOND);            cherry.eat();          chocolate.eat();          almond.eat();      }  } |

В консоли получим:



Отлично! Вот мы и создали свою первую фабрику.

**Какие преимущества дает паттерн Фабрика (Factory)**

Благодаря паттерну Factory наша жизнь облегчается:

1. Во-первых, мы можем создавать объекты разных типов с помощью **одного и того же метода**. Это очень удобно, когда возникает ситуация, в которой мы не знаем, какой тип нам понадобится.

Например, создадим метод "скушать случайный пончик" - **eatRandomDoughnut**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.util.Random;        public static void eatRandomDoughnut(DoughnutFactory factory){          Doughnut randomDougnut = getRandomDougnut(factory);          System.out.printf("What a surprise! ");          randomDougnut.eat();     }        public static Doughnut getRandomDougnut(DoughnutFactory factory){          Random random = new Random();          DoughnutTypes type = DoughnutTypes.values()[random.nextInt(DoughnutTypes.values().length)];            return(factory.getDoughnut(type));      } |

Обратите внимание - nextInt() будет генерировать целые числа в диапазоне [0; 3). То есть возможны такие опции: 0, 1 и 2. А это и есть 3 вида начинок пончиков.

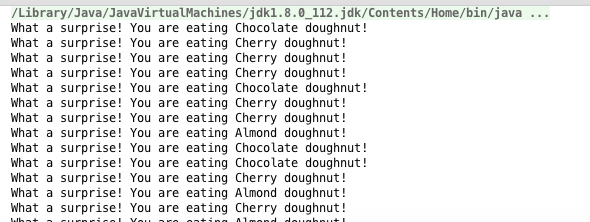
Давайте запустим:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | import java.util.Random;    public class Main {        public static void main(String[] args) {          DoughnutFactory factory = new DoughnutFactory();          eatRandomDoughnut(factory);     }        public static void eatRandomDoughnut(DoughnutFactory factory){          Doughnut randomDougnut = getRandomDougnut(factory);          System.out.printf("What a surprise! ");          randomDougnut.eat();      }        public static Doughnut getRandomDougnut(DoughnutFactory factory){          Random random = new Random();          DoughnutTypes type = DoughnutTypes.values()[random.nextInt(DoughnutTypes.values().length)];            return(factory.getDoughnut(type));      }  } |

Для убедительности, можно запустить метод **eatRandomDoughnut()** сто раз:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | import java.util.Random;    public class Main {        public static void main(String[] args) {          DoughnutFactory factory = new DoughnutFactory();          for(int i = 0; i < 100; i++) {              eatRandomDoughnut(factory);          }      }        public static void eatRandomDoughnut(DoughnutFactory factory) {          Doughnut randomDougnut = getRandomDougnut(factory);          System.out.printf("What a surprise! ");          randomDougnut.eat();      }        public static Doughnut getRandomDougnut(DoughnutFactory factory){          Random random = new Random();          DoughnutTypes type = DoughnutTypes.values()[random.nextInt(DoughnutTypes.values().length)];            return(factory.getDoughnut(type));      }    } |

В итоге получим что-то наподобие:



2. Во-вторых, мы можем "запаковать" дополнительный функционал в нашу Фабрику. Например, посчитаем, сколько пончиков каждого типа было создано конкретной фабрикой.

Сначала создаем новые переменные, которые будут хранить результат:

|  |  |
| --- | --- |
| 1  2  3 | private Integer cherryDoughnutCount;  private Integer chocolateDoughnutCount;  private Integer almondDoughnutCount; |

Нам желательно задать начальное значение для этих переменных - ноль.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class DoughnutFactory {      private Integer cherryDoughnutCount = 0;      private Integer chocolateDoughnutCount = 0;      private Integer almondDoughnutCount = 0;        //...    } |

Теперь, как мы будем изменять эти значения? Очень просто -дополним наш метод **getDoughnut()**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public Doughnut getDoughnut(DoughnutTypes type) {      Doughnut toReturn = null;      switch (type) {          case CHERRY:              cherryDoughnutCount++;              toReturn = new CherryDoughnut();              break;          case CHOCOLATE:              chocolateDoughnutCount++;              toReturn = new ChocolateDoughnut();              break;          case ALMOND:              almondDoughnutCount++;              toReturn = new AlmondDoughnut();              break;          default:              throw new IllegalArgumentException("Wrong doughnut type:" + type);      }      return toReturn;  } |

Отлично! Для вывода результатов в консоль напишем следущий метод:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public void printCount() {      System.out.println("Number of doughnuts produced (by type):");      System.out.println("Cherry doughnuts: " + cherryDoughnutCount);      System.out.println("Chocolate doughnuts: " + chocolateDoughnutCount);      System.out.println("Almond doughnuts: " + almondDoughnutCount);      System.out.println("Total: " + (cherryDoughnutCount + chocolateDoughnutCount + almondDoughnutCount));  } |

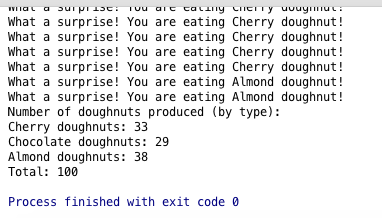
Итого, наш класс **DoughnutFactory** целиком, со всеми дополнениями, будет выглядеть так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | public class DoughnutFactory {      private Integer cherryDoughnutCount = 0;      private Integer chocolateDoughnutCount = 0;      private Integer almondDoughnutCount = 0;        public Doughnut getDoughnut(DoughnutTypes type) {          Doughnut toReturn = null;          switch (type) {              case CHERRY:                  cherryDoughnutCount++;                  toReturn = new CherryDoughnut();                  break;              case CHOCOLATE:                  chocolateDoughnutCount++;                  toReturn = new ChocolateDoughnut();                  break;              case ALMOND:                  almondDoughnutCount++;                  toReturn = new AlmondDoughnut();                  break;              default:                  throw new IllegalArgumentException("Wrong doughnut type:" + type);          }          return toReturn;      }        public void printCount() {          System.out.println("Number of doughnuts produced (by type):");          System.out.println("Cherry doughnuts: " + cherryDoughnutCount);          System.out.println("Chocolate doughnuts: " + chocolateDoughnutCount);          System.out.println("Almond doughnuts: " + almondDoughnutCount);          System.out.println("Total: " + (cherryDoughnutCount + chocolateDoughnutCount + almondDoughnutCount));      }  } |

Отлично! Теперь запустим с помощью такого кода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import java.util.Random;    public class Main {        public static void main(String[] args) {          DoughnutFactory factory = new DoughnutFactory();          for(int i = 0; i < 100; i++) {              eatRandomDoughnut(factory);          }          factory.printCount();      }        public static void eatRandomDoughnut(DoughnutFactory factory) {          Doughnut randomDougnut = getRandomDougnut(factory);          System.out.printf("What a surprise! ");          randomDougnut.eat();      }        public static Doughnut getRandomDougnut(DoughnutFactory factory){          Random random = new Random();          DoughnutTypes type = DoughnutTypes.values()[random.nextInt(DoughnutTypes.values().length)];            return(factory.getDoughnut(type));      }    } |

В консоли получим что-то похожее на это:



Согласитесь, считать с помощью Фабрики гораздо удобнее чем, например, создавать статические переменные в каждом классе, а помочь печатать их с помощью отдельного метода. Это и читабельнее, и будет легче править в будущем.

Конечно, это очень простой пример. Вы можете добавлять любой другой функционал.

3. И в-третьих, с помощью паттерна Factory мы можем генерировать сложные объекты намного проще и с меньшим количеством ошибок.

        Например, у нас есть объект с огромным количеством полей:

* б**О**льшая часть этих полей заполняются **типично** (например, тесто "вариант 1"  для пончика с вишней и тесто "вариант 2" для пончика с белым шоколадом);
* а часть полей будет заполняться разными значениями.

Таким образом, при создании сложных объектов, мы можем "прятать" весь этот процесс в фабрику. Например, если для создания объекта требуются какие-нибудь сложные вычисления, или если у объекта очень много параметров, а Вам пока нужен только объект с параметрами по умолчанию.

Как видите, у паттерну Фабрика очень много полезных применений. И теперь Вы знаете, как создать свою собственную.