2 Чистые функции

В этой главе вы узнаете:

- зачем нужны чистые функции;
- как передавать копии данных;
- как повторно вычислять значения вместо их хранения;
- как передавать состояние;
- как тестировать чистые функции.

> “Иногда наиболее элегантная реализация — это просто функция. Не метод. Не класс. Не фреймворк. Просто функция.”
>
> Джон Кармак (John Carmack)

Зачем нужны чистые функции

В предыдущей главе вы познакомились с функциями, которые не лгут. Их сигнатуры в точности рассказывают, что делает их тело. Мы пришли к заключению, что этим функциям можно доверять: чем меньше сюрпризов возникает при разработке кода, тем меньше ошибок будет в создаваемых нами приложениях. В этой главе вы познакомитесь с самой надежной из всех функций, которые не лгут: *чистой функцией*.

ЭТО ВАЖНО!
Чистые функции — основа функционального программирования.

Скидки на покупки

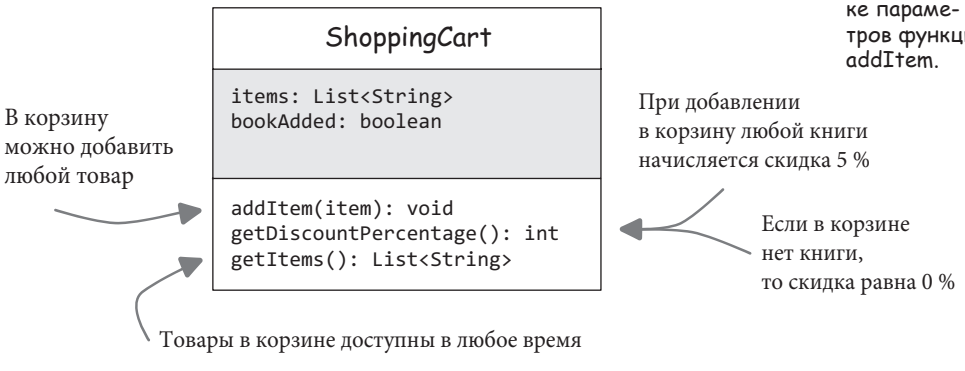
Начнем с примера, не использующего чистые функции. Мы посмотрим, какие проблемы свойственны этому решению, и попытаемся сначала решить их, используя интуитивное понимание. **Наша задача — реализовать «покупательскую корзину», способную вычислять скидки на основе ее текущего содержимого.**

Требования: покупательская корзина

1. В корзину можно добавить любой товар (смоделирован как значение типа `String`).
2. При добавлении в корзину любой книги начисляется скидка 5 %.
3. Если в корзине нет книги, то скидка равна 0 %.
4. Товары в корзине доступны в любое время.

Обратите внимание, что на диаграммах я иногда буду опускать типы и другие детали, чтобы сделать их как можно более понятными. Здесь я опустил тип String в списке параметров функции addItem.

Мы можем запрограммировать решение, прямо отобразив вышеперечисленные требования в код. Вот диаграмма класса `ShoppingCart`, представляющего реализацию:



Прежде чем углубиться в реализацию, кратко рассмотрим диаграмму, приведенную выше. Класс `ShoppingCart` имеет два поля, `items` и `bookAdded`, которые определяют внутреннее состояние. Каждое требование реализуется как отдельный метод. Эти методы играют роль общедоступного интерфейса с остальным миром (клиентами класса).

Императивное решение

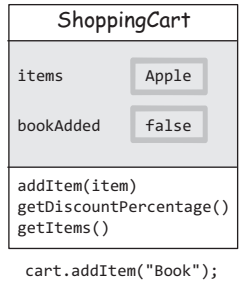
Мы разработали решение нашей проблемы, добавив несколько полей состояния и общедоступных методов. **Внимание!** Дизайн класса `ShoppingCart` имеет очень серьезные проблемы! Мы обсудим их ниже. Если вы уже заметили их, то поздравляю! Если нет, то подумайте о возможных способах неправильного использования этого класса и кода, представленного ниже.

Теперь пришло время написать код.

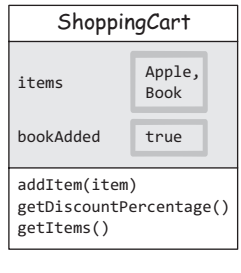
```
public class ShoppingCart {
  private List<String> items = new ArrayList<>();
  private boolean bookAdded = false;


  public void addItem(String item) {
    items.add(item);
    if(item.equals("Book")) {
      bookAdded = true;
    }
  }


  public int getDiscountPercentage() {
    if(bookAdded) {
      return 5;
    } else {
      return 0;
    }
  }


  public List<String> getItems() {
    return items;
  }
}
```

Эта диаграмма представляет фрагмент кода в нижней части страницы. Серая область представляет состояние (то есть переменные, которые будут менять значения с течением времени)

`cart.addItem("Apple");`



`cart.addItem("Book");`



Не забывайте, что это лишь небольшой пример, цель которого — показать некоторые неочевидные проблемы, существующие в реальном коде и сложно выявляемые.

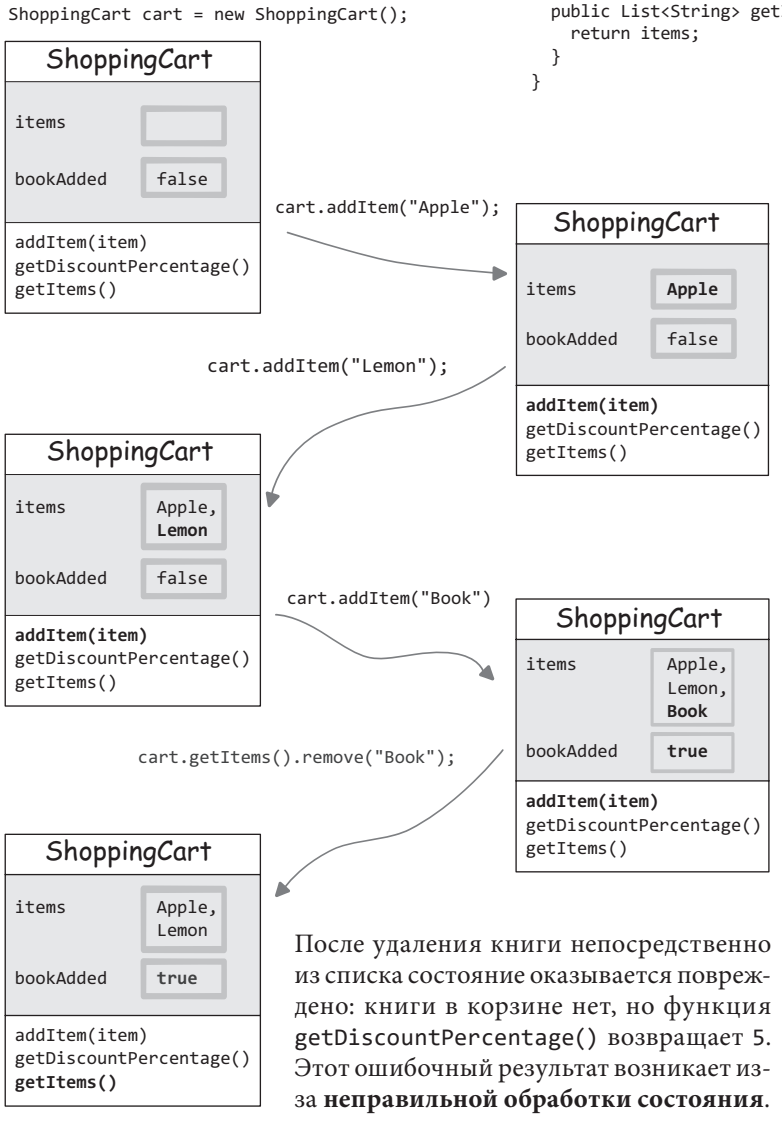
Выглядит разумно, правда? При добавлении книги в корзину мы присваиваем флагу `bookAdded` значение `true`. Этот флаг используется методом `getDiscountPercentage` для вычисления процента скидки. *Оба поля*, **`items`** и **`bookAdded`**, *называются состоянием*, поскольку их значения меняются со временем. Теперь посмотрим, как можно использовать этот класс.

```
ShoppingCart cart = new ShoppingCart();
cart.addItem("Apple");
System.out.println(cart.getDiscountPercentage());
вывод в консоли:  0
cart.addItem("Book");
System.out.println(cart.getDiscountPercentage());
вывод в консоли:  5
```

Ошибка в коде

Весь код, который мы видели до сих пор, выглядит неплохо. Тем не менее реализация класса ShoppingCart содержит ошибку, во многом связанную с *состоянием*: полями items и bookAdded. Посмотрим на один возможный поток выполнения программы, чтобы увидеть проблему.

```
class ShoppingCart {
  private List<String> items = new ArrayList<>();
  private boolean bookAdded = false;
  public void addItem(String item) {
    items.add(item);
    if(item.equals("Book")) {
      bookAdded = true;
    }
  }
  public int getDiscountPercentage() {
    if(bookAdded) {
      return 5;
    } else {
      return 0;
    }
  }
  public List<String> getItems() {
    return items;
  }
}
```



После удаления книги непосредственно из списка состояние оказывается повреждено: книги в корзине нет, но функция getDiscountPercentage() возвращает 5. Этот ошибочный результат возникает из-за **неправильной обработки состояния**.

Да, мы не планировали такой способ применения getItems, но помните, что любая возможность рано или поздно будет кем-то использована. При программировании важно подумать обо всех возможных способах использования, чтобы как можно лучше защитить внутреннее состояние.

Кстати, у getItems().add тоже есть проблемы! Опытные разработчики могут заметить, что она довольно очевидна, но будьте уверены, что такие проблемы встречаются довольно часто!

Передача копий данных

Проблему, с которой мы столкнулись в предыдущем примере, легко решить путем **возврата копии списка** из вызова `getItems`.

```
public class ShoppingCart {
  private List<String> items = new ArrayList<>();
  private boolean bookAdded = false;

  public void addItem(String item) {
    items.add(item);
    if(item.equals("Book")) {
      bookAdded = true;
    }
  }

  public int getDiscountPercentage() {
    if(bookAdded) {
      return 5;
    } else {
      return 0;
    }
  }
  public List<String> getItems() {
   return items;
  }
}
```

```
public List<String> getItems() {
  return new ArrayList<>(items);
}
```

Мы возвращаем не текущее состояние `items`, а создаем и возвращаем копию. В таком случае никто не сможет испортить `items`.

Почему используется копия вместо Collections.unmodifiableList, вы узнаете в главе 3

Это изменение может показаться не таким уж важным, но **передача копий данных — один из фундаментальных аспектов функционального программирования**! Очень скоро мы подробно рассмотрим этот прием. Но сначала убедимся, что класс `ShoppingCart` работает верно, независимо от того, как он используется.

Удаление товара из корзины

Допустим, клиенту нашего класса неожиданно понадобилась дополнительная возможность, не оговоренная вначале. Мы осознали это, получив горький опыт неправильной работы нашего кода. Вот требование № 5.

> 5. Любой товар, ранее добавленный в корзину, можно удалить.

ЭТО ВАЖНО!

В ФП мы передаем копии данных и не позволяем изменять их на месте.

Поскольку теперь вызывающей стороне возвращается копия `items`, для удовлетворения этого требования нужно добавить еще один общедоступный метод:

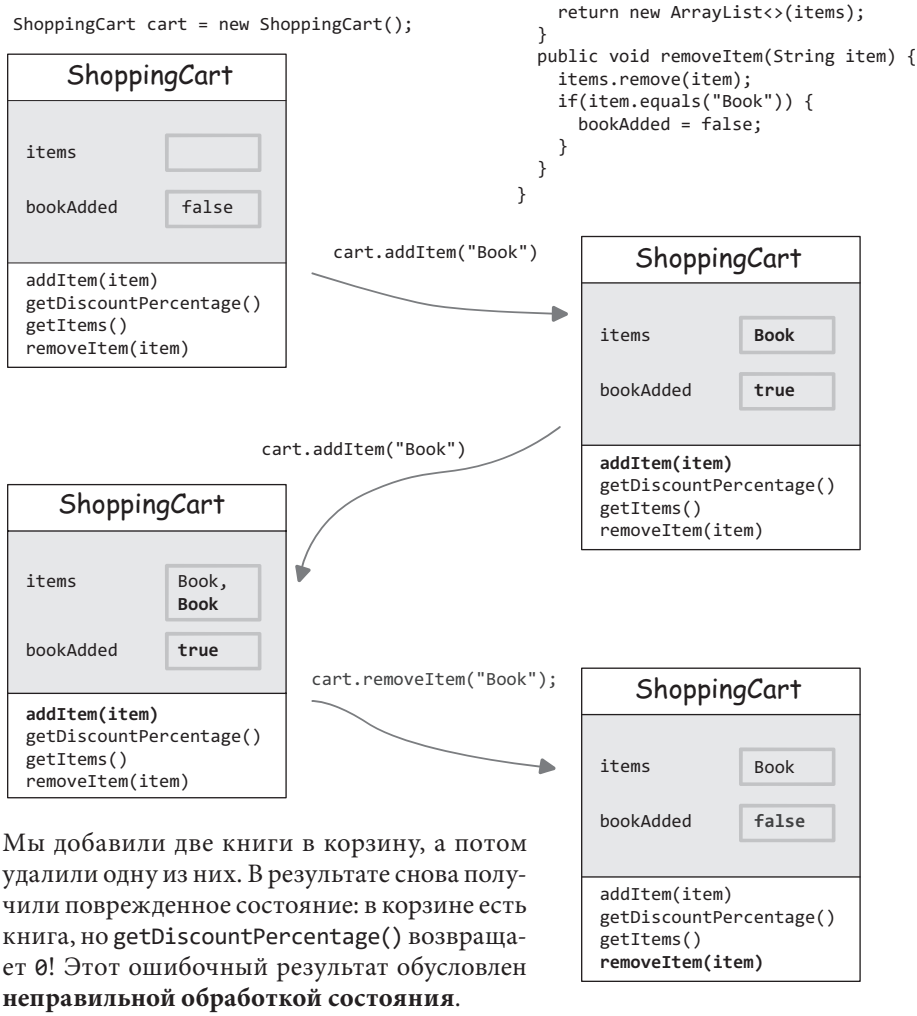
```
public void removeItem(String item) {
  items.remove(item);
  if(item.equals("Book")) {
    bookAdded = false;
  }
}
```

Это конец нашим проблемам? Теперь код работает *правильно*?

Ошибка в коде... снова

Теперь мы возвращаем копию `items` и добавили метод `removeItem`, что значительно улучшило наше решение. Можем ли мы сказать, что с проблемой покончено? Оказывается, нет. Удивительно, но проблем с `ShoppingCart` и его внутренним состоянием даже больше, чем можно было ожидать. Посмотрим на другой возможный поток выполнения программы, чтобы увидеть новую проблему.

```
class ShoppingCart {
  private List<String> items = new ArrayList<>();
  private boolean bookAdded = false;

  public void addItem(String item) {
    items.add(item);
    if(item.equals("Book")) {
      bookAdded = true;
    }
  }

  public int getDiscountPercentage() {
    if(bookAdded) {
      return 5;
    } else {
      return 0;
    }
  }

  public List<String> getItems() {
    return new ArrayList<>(items);
  }
  public void removeItem(String item) {
    items.remove(item);
    if(item.equals("Book")) {
      bookAdded = false;
    }
  }
}
```



Мы добавили две книги в корзину, а потом удалили одну из них. В результате снова получили поврежденное состояние: в корзине есть книга, но `getDiscountPercentage()` возвращает `0`! Этот ошибочный результат обусловлен **неправильной обработкой состояния**.

Повторные вычисления вместо сохранения

Проблему, с которой мы столкнулись в предыдущем примере, можно решить, если отступить на шаг назад и переосмыслить нашу основную цель.

Перед нами стояла задача реализовать функционал покупательской корзины, способной вычислять скидку. Мы попали в ловушку, пытаясь отслеживать все операции добавления и удаления и императивно определяя, была ли добавлена книга. Вместо этого можно просто **вычислять скидку заново каждый раз**, когда это необходимо, просматривая весь список.

```
public class ShoppingCart {
  private List<String> items = new ArrayList<>();
  private boolean bookAdded = false;

  public void addItem(String item) {
    items.add(item);
    if(item.equals("Book")) {
      bookAdded = true;
    }
  }

  public int getDiscountPercentage() {
    if(bookAdded) {
      return 5;
    } else {
      return 0;
    }
  }

  public List<String> getItems() {
    return new ArrayList<>(items);
  }

  public void removeItem(String item) {
    items.remove(item);
    if(item.equals("Book")) {
      bookAdded = false;
    }
  }
}
```

Мы удалили состояние bookAdded и перенесли логику вычисления скидки из addItem/removeItem в getDiscountPercentage.

```
public int getDiscountPercentage() {
  if(items.contains("Book")) {
    return 5;
  } else {
    return 0;
  }
}
```

getDiscountPercentage вычисляет скидку, когда она необходима, просматривая весь список.

Какая перемена! Код стал намного безопаснее и менее проблематичным. Вся логика, связанная со скидками, теперь находится в `getDiscountPercentage`. Мы убрали состояние `bookAdded`, доставившее нам столько проблем. Единственный недостаток новой версии — для очень больших списков покупок может потребоваться много времени, чтобы вычислить скидку. **В крайних случаях мы можем пожертвовать производительностью ради удобочитаемости и простоты сопровождения.**

Мы вернемся к этой теме в главе 3