3.2. Не используйте статические методы

Обсуждение на http://goo.gl/8ql2ov.

Ах, статические методы... Одна из моих любимых тем. Мне понадобилось несколько лет, чтобы осознать, насколько важна эта проблема. Теперь я сожалею обо всем том времени, которое потратил на написание процедурного, а не объектно-ориентированного программного обеспечения. Я был слеп, но теперь прозрел. Статические методы — настолько же большая, если не еще бóльшая проблема в ООП, чем наличие константы `NULL`. Статических методов в принципе не должно было быть в Java, да и в других объектно-ориентированных языках, но, увы, они там есть. Мы не должны знать о таких вещах, как ключевое слово `static` в Java, но, увы, вынуждены. Я не знаю, кто именно привнес их в Java, но они — чистейшее зло. Статические методы, а не авторы этой возможности. Я надеюсь.

Посмотрим, что такое статические методы и почему мы до сих пор создаем их. Скажем, мне нужна функциональность загрузки веб-страницы посредством HTTP-запросов. Я создаю такой «класс»:

```
class WebPage {
  public static String read(String uri) {
    // выполнить HTTP-запрос
    // и конвертировать ответ в UTF8-строку
  }
}
```

Пользоваться им очень удобно:

```
String html = WebPage.read("http://www.java.com");
```

Метод `read()` относится к тому классу методов, против которого я выступаю. Предлагаю вместо этого использовать объект

(также я поменял имя метода в соответствии с рекомендациями из раздела 2.4):

```
class WebPage {
  private final String uri;
  public String content() {
    // выполнить HTTP-запрос
    // и конвертировать ответ в UTF8-строку
  }
}
```

Вот как им пользоваться:

```
String html = new WebPage("http://www.java.com")
  .content();
```

Вы можете сказать, что между ними нет особой разницы. Статические методы работают даже быстрее, потому что нам нет необходимости создавать новый объект каждый раз, когда нужно скачать веб-страницу. Просто вызовите статический метод, он сделает дело, вы получите результат и будете работать дальше. Нет необходимости возиться с объектами и сборщиком мусора. Кроме того, мы можем сгруппировать несколько статических методов в класс-утилиту и назвать его, скажем, `WebUtils`.

Эти методы помогут загружать веб-страницы, получать статистическую информацию, определять время отклика и т. п. В них будет много методов, а использовать их просто и интуитивно понятно. Кроме того, как применять статические методы, тоже интуитивно понятно. Все понимают, как они работают. Просто напишите `WebPage.read()`, и — вы догадались! — будет прочитана страница. Мы дали компьютеру инструкцию, и он ее выполняет. Просто и понятно, так ведь? А вот и нет!

Статические методы в любом контексте — безошибочный индикатор плохого программиста, понятия не имеющего об ООП. Для применения статических методов нет ни единого оправда-

ния ни в одной ситуации. Забота о производительности не считается. Статические методы — издевательство над объектно-ориентированной парадигмой. Они существуют в Java, Ruby, C++, PHP и других языках. К несчастью. Мы не можем их оттуда выбросить, не можем переписать все библиотеки с открытым исходным кодом, полные статических методов, но можем прекратить использовать их в своем коде.

Мы должны прекратить применять статические методы.

Теперь посмотрим на них с нескольких разных позиций и обсудим их практические недостатки. Я могу заранее обобщить их для вас: статические методы ухудшают сопровождаемость программного обеспечения. Это не должно вас удивлять. Все сводится к сопровождаемости.

Объектное мышление против компьютерного

Изначально я назвал этот подраздел «Объектное мышление против процедурного», но потом переименовал. «Процедурное мышление» означает почти то же самое, но словосочетание «мыслить как компьютер» лучше описывает проблему. Мы унаследовали этот образ мышления из ранних языков программирования, таких как Assembly, C, COBOL, Basic, Pascal, и многих других. Основа парадигмы в том, что компьютер работает на нас, а мы указываем ему, что делать, давая ему явные инструкции, например:

```
  CMP AX, BX
  JNAE greater
  MOV CX, BX
  RET
greater:
  MOV CX, AX
  RET
```

Это ассемблерная «подпрограмма» для процессора Intel 8086. Она находит и возвращает большее из двух чисел. Мы помещаем их в регистры AX и BX соответственно, а результат попадает в регистр CX. Вот точно такой же код на языке C:

```
int max(int a, int b) {
  if (a > b) {
    return a;
  }
  return b;
}
```

«Что же с этим настолько не так?» — спросите вы. Ничего. Все с этим кодом в порядке — он работает, как и положено. Именно так работают все компьютеры. Они ожидают, что мы дадим им инструкции, которые они будут исполнять одну за другой. Многие годы мы писали программы именно так. Преимущество данного подхода в том, что мы остаемся вблизи процессора, направляя его дальнейшее движение. Мы у руля, а компьютер следует нашим инструкциям. Мы указываем компьютеру, как найти большее из двух чисел. Мы принимаем решения, он им следует. Поток исполнения всегда последователен, от начала сценария до его конца.

Такой *линейный* тип мышления называется «думать как компьютер». Компьютер в какой-то момент начнет исполнять инструкции и в какой-то момент закончит делать это. При написании кода на языке C мы вынуждены думать таким образом. Операторы, разделенные точками с запятыми, идут сверху вниз. Такой стиль унаследован из ассемблера.

Хотя языки более высокого уровня, чем ассемблер, имеют процедуры, подпрограммы и другие механизмы абстракции, они не устраняют последовательный образ мышления. Программа все равно проходится сверху вниз. В таком подходе нет ничего зазорного при написании небольших программ, но в более крупных масштабах так мыслить трудно.

Взглянем на тот же код, записанный на *функциональном* языке программирования Lisp:

```
(defun max (a b)
   (if (> a b) a b))
```

Можете ли вы сказать, где начинается и заканчивается исполнение этого кода? Нет. Мы не знаем, ни каким образом процессор получит результат, ни то, как конкретно будет работать функция `if`. Мы очень отстранены от процессора. Мы мыслим как функция, а не как компьютер. Когда нам нужна новая вещь, мы определяем ее:

```
(def x (max 5 9))
```

Мы *определяем*, а не даем инструкции процессору. Этой строчкой мы привязываем `x` к `(max 5 9)`. Мы не просим компьютер вычислить большее из двух чисел. Мы просто говорим, что `x` есть большее из двух чисел. Мы не управляем тем, как и когда это будет вычислено. Обратите внимание, это важно: `x` есть большее из чисел. Отношение «есть» («быть», «являться») — то, чем отличается функциональная, логическая и объектно-ориентированная парадигма программирования от процедурной.

При компьютерном образе мышления мы находимся у руля и контролируем поток исполнения инструкций. При объектно-ориентированном образе мышления мы просто определяем, кто есть кто, и пусть они взаимодействуют, когда это им понадобится. Вот как вычисление большего из двух чисел должно выглядеть в ООП:

```
class Max implements Number {
  private final Number a;
  private final Number b;
  public Max(Number left, Number right) {
    this.a = left;
    this.b = right;
  }
}
```

А так я буду его использовать:

```
Number x = new Max(5, 9);
```

Смотрите, я не вычисляю большее из двух чисел. Я определяю, что `x` есть большее из двух чисел. Меня не особо беспокоит, что находится внутри объекта класса `Max` и как именно он реализует интерфейс `Number`. Я не даю процессору инструкции относительно этого вычисления. Я просто инстанцирую объект. Это очень похоже на `def` в Lisp. В этом смысле ООП очень похоже на функциональное программирование.

Напротив, статические методы в ООП — то же самое, что подпрограммы в C или ассемблере. Они не имеют отношения к ООП и заставляют нас писать процедурный код в объектно-ориентированном синтаксисе. Вот код на Java:

```
int x = Math.max(5, 9);
```

Это совершенно неправильно и не должно использоваться в настоящем объектно-ориентированном проектировании.

Декларативный стиль против императивного

Императивное программирование «описывает вычисления в терминах операторов, изменяющих состояние программы». *Декларативное* программирование, с другой стороны, «выражает логику вычисления, не описывая поток его выполнения» (я цитирую «Википедию»). Об этом мы, по сути, говорили на протяжении нескольких предыдущих страниц. Императивное программирование похоже на то, что делают компьютеры, — последовательное выполнение инструкций. Декларативное программирование ближе к естественному образу мышления, в котором у нас есть сущности и отношения между ними. Очевидно, что декларативное программирование — более мощный подход, но императивный подход понятнее процедурным программи-

стам. Почему декларативный подход более мощный? Не переключайтесь, и через несколько страниц мы доберемся до сути.

Какое отношение все это имеет к статическим методам? Неважно, статический это метод или объект, мы все еще должны где-то написать `if (a > b)`, так ведь? Да, именно так. Как статический метод, так и объект — всего лишь обертка над оператором `if`, который выполняет задачу сравнения `a` с `b`. Разница в том, как эта функциональность *используется* другими классами, объектами и методами. И это существенная разница. Рассмотрим ее на примере.

Скажем, у меня есть интервал, ограниченный двумя целыми числами, и целое число, которое должно в него попадать. Я должен убедиться, что это так. Вот что мне *придется* сделать, если метод `max()` — статический:

```
public static int between(int l, int r, int x) {
  return Math.min(Math.max(l, x), r);
}
```

Нужно создать еще один статический метод, `between()`, который использует два имеющихся статических метода, `Math.min()` и `Math.max()`. Есть только один способ это сделать — императивный подход, поскольку значение вычисляется сразу же. Когда я делаю вызов, я немедленно получаю результат:

```
int y = Math.between(5, 9, 13); // возвращает 9
```

Я получаю число `9` сразу же после вызова `between()`. Когда будет сделан вызов, мой процессор тут же начнет работать над этим вычислением. Это *императивный* подход. А как тогда выглядит декларативный подход?

Вот, взгляните:

```
class Between implements Number {
  private final Number num;
  Between(Number left, Number right, Number x) {
    this.num = new Min(new Max(left, x), right);
```