Оглавление

16 Изменяемые объекты

В предыдущих главах в центре внимания были функциональные (неизменяемые) объекты. Дело в том, что идея использования объектов без какого-либо изменяемого состояния заслуживала более пристального рассмотрения. Но в Scala также вполне возможно определять объекты с изменяемым состоянием. Подобные изменяемые объекты зачастую появляются естественным образом, когда нужно смоделировать объекты из реального мира, которые со временем подвергаются изменениям.

В этой главе мы раскроем суть изменяемых объектов и рассмотрим синтаксические средства для их выражения, предлагаемые Scala. Кроме того, рассмотрим большой пример моделирования дискретных событий, в котором используются изменяемые объекты, а также описан внутренний предметно-ориентированный язык (domain-specific language, DSL), предназначенный для определения моделируемых цифровых электронных схем.

16.1. Что делает объект изменяемым

Принципиальную разницу между чисто функциональным и изменяемым объектами можно проследить, даже не изучая реализацию объектов. При вызове метода или получении значения поля по указателю в отношении функционального объекта вы всегда будете получать один и тот же результат.

Например, если есть следующий список символов:

```
val cs = List('a', 'b', 'c')
```

то применение `cs.head` всегда будет возвращать `'a'`. То же самое произойдет, даже если между местом определения `cs` и местом, где будет применено об-

ращение `cs.head`, над списком `cs` будет проделано произвольное количество других операций.

Что же касается изменяемого объекта, то результат вызова метода или обращения к полю может зависеть от того, какие операции были ранее выполнены в отношении объекта. Хороший пример изменяемого объекта — банковский счет. Его упрощенная реализация показана в листинге 16.1.

Листинг 16.1. Изменяемый класс банковского счета

```
class BankAccount:

  private var bal: Int = 0

  def balance: Int = bal

  def deposit(amount: Int): Unit =
    require(amount > 0)
    bal += amount

  def withdraw(amount: Int): Boolean =
    if amount > bal then false
    else
      bal -= amount
      true
```

В классе `BankAccount` определяются приватная переменная `bal` и три публичных метода: `balance` возвращает текущий баланс, `deposit` добавляет к `bal` заданную сумму, `withdraw` предпринимает попытку вывести из `bal` заданную сумму, гарантируя при этом, что баланс не станет отрицательным. Возвращаемое `withdraw` значение, имеющее тип `Boolean`, показывает, были ли запрошенные средства успешно выведены.

Даже если ничего не знать о внутренней работе класса `BankAccount`, все же можно сказать, что экземпляры `BankAccounts` являются изменяемыми объектами:

```
val account = new BankAccount
account.deposit(100)
account.withdraw(80) // true
account.withdraw(80) // false
```

Обратите внимание: в двух последних операциях вывода средств в ходе работы с программой были возвращены разные результаты. По выполнении первой операции было возвращено значение `true`, поскольку на банковском счете содержался достаточный объем, позволяющий вывести средства. Вторая операция вывода средств была такой же, как и первая, однако по ее выполнении было

возвращено значение `false`, поскольку баланс счета уменьшился настолько, что уже не мог покрыть запрошенные средства. Исходя из этого, мы понимаем, что банковским счетам присуще изменяемое состояние, так как при выполнении одной и той же операции в разное время получаются разные результаты.

Можно подумать, будто изменяемость `BankAccount` априори не вызывает сомнений, поскольку в нем содержится определение `var`-переменной. Изменяемость и `var`-переменные обычно идут рука об руку, но ситуация не всегда бывает столь очевидной. Например, класс может быть изменяемым и без определения или наследования каких-либо `var`-переменных, поскольку перенаправляет вызовы методов другим объектам, которые находятся в изменяемом состоянии. Может сложиться и обратная ситуация: класс содержит `var`-переменные и все же является чисто функциональным. Как образец, можно привести класс, кэширующий результаты дорогой операции в поле в целях оптимизации. Чтобы подобрать пример, предположим наличие неоптимизированного класса `Keyed` с дорогой операцией `computeKey`:

```
class Keyed:
  def computeKey: Int = ... // займет некоторое время
  ...
```

При условии, что `computeKey` не читает и не записывает никаких `var`-переменных, эффективность `Keyed` можно увеличить, добавив кэш:

```
class MemoKeyed extends Keyed:
  private var keyCache: Option[Int] = None
  override def computeKey: Int =
    if !keyCache.isDefined then
      keyCache = Some(super.computeKey)
    keyCache.get
```

Использование `MemoKeyed` вместо `Keyed` может ускорить работу: когда результат выполнения операции `computeKey` будет запрошен повторно, вместо еще одного запуска `computeKey` может быть возвращено значение, сохраненное в поле `keyCache`. Но за исключением такого ускорения поведение классов `Keyed` и `MemoKeyed` абсолютно одинаково. Следовательно, если `Keyed` является чисто функциональным классом, то таковым будет и класс `MemoKeyed`, даже притом что содержит переназначаемую переменную.

16.2. Переназначаемые переменные и свойства

В отношении переназначаемой переменной допускается выполнение двух основных операций: получения ее значения или присваивания ей нового.

В таких библиотеках, как JavaBeans, эти операции часто инкапсулированы в отдельные методы считывания (getter) и записи значения (setter), которые необходимо объявлять явно.

В Scala каждая `var`-переменная представляет собой неприватный член какого-либо объекта, в отношении которого в нем неявно определены методы геттер и сеттер. Но названия таких методов отличаются от предписанных соглашениями Java. Метод получения значения (геттер) `var`-переменной `x` называется просто `x`, а метод присваивания значения (сеттер) — `x_=`.

Например, появляясь в классе, определение `var`-переменной

```
var hour = 12
```

создает геттер `hour` и сеттер `hour_=` вдобавок к переназначаемому полю, у которого всегда имеется внутренняя пометка `"object private"`. Она означает, что доступ к полю устанавливается только из объекта, который его содержит. В то же время геттер и сеттер обеспечивают исходной `var`-переменной некоторую видимость. Если `var`-переменная объявлена публичной (public), то таковыми же являются и геттер, и сеттер. Если она является защищенной (protected), то и они тоже, и т. д.

Рассмотрим, к примеру, класс `Time`, показанный в листинге 16.2, в котором определены две публичные `var`-переменные с именами `hour` и `minute`.

Листинг 16.2. Класс с публичными var-переменными

```
class Time:
  var hour = 12
  var minute = 0
```

Эта реализация в точности соответствует определению класса, показанного в листинге 16.3. В данном определении имена локальных полей `h` и `m` были выбраны произвольно, чтобы не конфликтовали с уже используемыми именами.

Листинг 16.3. Как публичные var-переменные расширяются в геттер и сеттер

```
class Time:

  private var h = 12
  private var m = 0

  def hour: Int = h
  def hour_=(x: Int) =
    h = x

  def minute: Int = m
  def minute_=(x: Int) =
    m = x
```

Интересным аспектом такого расширения var-переменных в геттер и сеттер является то, что вместо определения var-переменной можно также выбрать вариант непосредственного определения этих методов доступа. Он позволяет как угодно интерпретировать операции доступа к переменной и присваивания ей значения. Например, вариант класса Time, показанный в листинге 16.4, содержит необходимые условия, благодаря которым перехватываются все присваивания недопустимых значений часам и минутам, хранящимся в переменных hour и minute.

Листинг 16.4. Непосредственное определение геттера и сеттера

```
class Time:

  private var h = 12
  private var m = 0

  def hour: Int = h
  def hour_=(x: Int) =
    require(0 <= x && x < 24)
    h = x

  def minute = m
  def minute_=(x: Int) =
    require(0 <= x && x < 60)
    m = x
```

В некоторых языках для этих похожих на переменные величин, не являющихся простыми переменными из-за того, что их геттер и сеттер могут быть переопределены, имеются специальные синтаксические конструкции. Например, в C# эту роль играют свойства. По сути, принятое в Scala соглашение о постоянной интерпретации переменной как имеющей пару геттер и сеттер предоставляет вам такие же возможности, что и свойства C#, но при этом не требует какого-то специального синтаксиса.

Свойства могут иметь множество назначений. В примере, показанном в листинге 16.4, методы присваивания значений навязывают соблюдение конкретных условий, защищая таким образом переменную от присваивания ей недопустимых значений. Кроме того, свойства позволяют регистрировать все обращения к переменной со стороны геттера и сеттера. Или же можно объединять переменные с событиями, например уведомляя с помощью методов-подписчиков о каждом изменении переменной.

Вдобавок возможно, а иногда и полезно определять геттер и сеттер без связанных с ними полей. Например, в листинге 16.5 показан класс Thermometer,

в котором инкапсулирована переменная `temperature`, позволяющая читать и обновлять ее значение. Температурные значения могут выражаться в градусах Цельсия или Фаренгейта. Этот класс позволяет получать и устанавливать значение температуры в любых единицах измерения.

Листинг 16.5. Определение геттера и сеттера без связанного с ними поля

```
import scala.compiletime.uninitialized

class Thermometer:

  var celsius: Float = uninitialized

  def fahrenheit = celsius * 9 / 5 + 32

  def fahrenheit_=(f: Float) =
    celsius = (f - 32) * 5 / 9

  override def toString = s"${fahrenheit}F/${celsius}C"
```

В первой строке тела этого класса определяется `var`-переменная `celsius`, в которой будет храниться значение температуры в градусах Цельсия. Для переменной `celsius` изначально устанавливается значение по умолчанию: в качестве инициализирующего значения для нее устанавливается знак `= uninitialized`. Точнее, инициализатором поля данному полю присваивается нулевое значение. Суть нулевого значения зависит от типа поля. Для числовых типов это `0`, для булевых — `false`, а для ссылочных — `null`. Получается то же самое, что и при определении в Java некой переменной без инициализатора.

Учтите, что в Scala просто отбросить инициализатор `= uninitialized` нельзя. Если использовать код

```
var celsius: Float
```

то получится объявление абстрактной, а не инициализированной переменной[1].

За определением переменной `celsius` следуют геттер по имени `fahrenheit` и сеттер `fahrenheit_=`, которые обращаются к той же температуре, но в градусах Фаренгейта. В листинге нет отдельного поля, содержащего значение текущей температуры в таких градусах. Вместо этого геттер и сеттер для

[1] Абстрактные переменные будут рассматриваться в главе 20.

значений в градусах Фаренгейта выполняют автоматическое преобразование из градусов Цельсия и в них же соответственно. Пример взаимодействия с объектом `Thermometer` выглядит следующим образом:

```
val t = new Thermometer
t // 32.0F/0.0C

t.celsius = 100
t // 212.0F/100.0C

t.fahrenheit = -40
t // -40.0F/-40.0C
```

16.3. Практический пример: моделирование дискретных событий

Далее в главе на расширенном примере будут показаны интересные способы возможного сочетания изменяемых объектов с функциями, являющимися значениями первого класса. Речь идет о проектировании и реализации симулятора цифровых схем. Эта задача разбита на несколько подзадач, каждая из которых интересна сама по себе.

Сначала мы покажем весьма лаконичный язык для цифровых схем. Его определение подчеркнет общий метод встраивания предметно-ориентированных языков (domain-specific languages, DSL) в язык их реализации, подобный Scala. Затем представим простую, но всеобъемлющую среду для моделирования дискретных событий. Ее основной задачей будет являться отслеживание действий, выполняемых в ходе моделирования. И наконец, мы покажем, как структурировать и создавать программы дискретного моделирования. Цели создания таких программ — моделирование физических объектов объектами-симуляторами и использование среды для моделирования физического времени.

Этот пример взят из классического учебного пособия Абельсона и Суссмана [Abe96]. Наша ситуация отличается тем, что языком реализации является Scala, а не Scheme, и тем, что различные аспекты примера структурно выделены в четыре программных уровня. Первый относится к среде моделирования, второй — к основному пакету моделирования схем, третий касается библиотеки определяемых пользователем электронных схем, а четвертый, последний уровень предназначен для каждой моделируемой схемы как таковой. Каждый уровень выражен в виде класса, и более конкретные уровни являются наследниками более общих.