

Оглавление

Отзывы на предыдущие издания книги	18
Предисловие	22
Благодарности	27
Введение	31
Целевая аудитория	31
Как пользоваться книгой	31
Как изучать Scala	32
Условные обозначения	33
Структура книги	33
Ресурсы	35
Исходный код	36
От издательства	36
Глава 1. Масштабируемый язык	37
1.1. Язык, который растет вместе с вами	38
Растут новые типы.	39
Растут новые управляющие конструкции	41
1.2. Почему язык Scala масштабируемый?	41
Scala — объектно-ориентированный язык.	42
Scala — функциональный язык.	43
1.3. Почему именно Scala	45
Scala — совместимый язык.	46
Scala — лаконичный язык	47
Scala — высокоуровневый язык	48
Scala — статически типизированный язык	50
1.4. Истоки Scala	53
Резюме	55

Глава 2. Первые шаги в Scala	56
Шаг 1. Осваиваем Scala REPL	57
Шаг 2. Объявляем переменные	58
Шаг 3. Определяем функции	60
Шаг 4. Пишем Scala-скрипты	62
Шаг 5. Организуем цикл с while и принимаем решение с if	64
Шаг 6. Перебираем элементы с foreach и for-do	66
Резюме	68
Глава 3. Дальнейшие шаги в Scala	69
Шаг 7. Параметризуем массивы типами	69
Шаг 8. Используем списки	73
Шаг 9. Используем кортежи	78
Шаг 10. Используем множества и отображения	79
Шаг 11. Учимся распознавать функциональный стиль	84
Шаг 12. Преобразование с отображениями и for-yield	87
Резюме	90
Глава 4. Классы и объекты	91
4.1. Классы, поля и методы	91
4.2. Когда подразумевается использование точки с запятой	96
4.3. Объекты-одиночки	96
4.4. Case-классы	99
4.5. Приложение на языке Scala	101
Резюме	103
Глава 5. Основные типы и операции	104
5.1. Некоторые основные типы	104
5.2. Литералы	105
Целочисленные литералы	106
Литералы чисел с плавающей точкой.	107
Большие числовые литералы	107
Символьные литералы	108
Строковые литералы	109
Булевы литералы.	110
5.3. Интерполяция строк	110
5.4. Все операторы являются методами	112
5.5. Арифметические операции	115
5.6. Отношения и логические операции	116

5.7. Поразрядные операции	117
5.8. Равенство объектов	119
5.9. Приоритет и ассоциативность операторов	120
5.10. Обогащающие операции	123
Резюме	124
Глава 6. Функциональные объекты	125
6.1. Спецификация класса Rational	125
6.2. Конструирование класса Rational	126
6.3. Переопределение метода toString	128
6.4. Проверка соблюдения предварительных условий	129
6.5. Добавление полей	130
6.6. Собственные ссылки	132
6.7. Вспомогательные конструкторы	132
6.8. Приватные поля и методы	134
6.9. Определение операторов	135
6.10. Идентификаторы в Scala	137
6.11. Перегрузка методов	140
6.12. Методы расширения	142
6.13. Предостережение	143
Резюме	143
Глава 7. Встроенные управляющие конструкции	145
7.1. Выражения if	146
7.2. Циклы while	147
7.3. Выражения for	150
Обход элементов коллекций	150
Фильтрация	152
Вложенные итерации	153
Привязки промежуточных переменных	153
Создание новой коллекции	154
7.4. Обработка исключений с помощью выражений try	155
Генерация исключений	155
Перехват исключений	156
Условие finally	157
Выдача значения	157
7.5. Выражения match	158
7.6. Программирование без break и continue	160

7.7. Область видимости переменных	162
7.8. Рефакторинг кода, написанного в императивном стиле	165
7.9. Резюме	167
Глава 8. Функции и замыкания	168
8.1. Методы	168
8.2. Локальные функции	169
8.3. Функции первого класса	171
8.4. Краткие формы функциональных литералов	173
8.5. Синтаксис заместителя	173
8.6. Частично примененные функции	174
8.7. Замыкания	177
8.8. Специальные формы вызова функций	180
Повторяющиеся параметры	180
Именованные аргументы	181
Значения параметров по умолчанию	182
8.9. Тип SAM	183
8.10. Хвостовая рекурсия	184
Трассировка функций с хвостовой рекурсией	185
Ограничения хвостовой рекурсии	187
Резюме	188
Глава 9. Управляющие абстракции	189
9.1. Сокращение повторяемости кода	189
9.2. Упрощение клиентского кода	193
9.3. Карринг	195
9.4. Создание новых управляющих конструкций	196
9.5. Передача параметров по имени	199
Резюме	202
Глава 10. Композиция и наследование	203
10.1. Библиотека двумерной разметки	203
10.2. Абстрактные классы	204
10.3. Определяем методы без параметров	205
10.4. Расширяем классы	208
10.5. Переопределяем методы и поля	210
10.6. Определяем параметрические поля	211
10.7. Вызываем конструктор суперкласса	212

10.8. Используем модификатор <code>override</code>	213
10.9. Полиморфизм и динамическое связывание	215
10.10. Объявляем финальные элементы	217
10.11. Используем композицию и наследование	218
10.12. Реализуем методы <code>above</code> , <code>beside</code> и <code>toString</code>	220
10.13. Определяем фабричный объект	223
10.14. Методы <code>heighten</code> и <code>widen</code>	224
10.15. Собираем все вместе	227
Резюме	228
Глава 11. Трейты	229
11.1. Как работают трейты	229
11.2. Сравнение «тонких» и «толстых» интерфейсов	232
11.3. Трейты как наращиваемые модификации	235
11.4. Почему не используется множественное наследование	239
11.5. Параметры трейтов	243
Резюме	245
Глава 12. Пакеты, импорты и экспорты	247
12.1. Помещение кода в пакеты	247
12.2. Краткая форма доступа к родственному коду	249
12.3. Импортирование кода	252
12.4. Неявное импортирование	255
12.5. Модификаторы доступа	256
Приватные члены	256
Защищенные члены	257
Публичные члены	258
Область защиты	258
Видимость и объекты-компаньоны	260
12.6. Определения верхнего уровня	261
12.7. Экспорты	262
Резюме	265
Глава 13. Сопоставление с образцом	266
13.1. Простой пример	266
case-классы	267
Сопоставление с образцом	268
Сравнение <code>match</code> со <code>switch</code>	270

13.2. Разновидности паттернов	271
Подстановочные паттерны	271
Паттерны-константы	272
Паттерны-переменные	272
Паттерны-конструкторы	274
Паттерны-последовательности	275
Паттерны-кортежи	276
Типизированные паттерны	276
Затирание типов	279
Привязка переменной	280
13.3. Ограждение образца	280
13.4. Наложение паттернов	281
13.5. Запечатанные классы	283
13.6. Сопоставление паттерна Options	285
13.7. Паттерны повсюду	286
Паттерны в определениях переменных	286
Последовательности вариантов в качестве частично примененных функций	287
Паттерны в выражениях for	290
13.8. Большой пример	291
Резюме	298
Глава 14. Работа со списками	299
14.1. Литералы списков	299
14.2. Тип List	300
14.3. Создание списков	300
14.4. Основные операции над списками	301
14.5. Паттерны-списки	302
14.6. Методы первого порядка класса List	304
Конкатенация двух списков	304
Принцип «разделяй и властвуй»	305
Получение длины списка: length	307
Обращение к концу списка: init и last	307
Реверсирование списков: reverse	308
Префиксы и суффиксы: drop, take и splitAt	309
Выбор элемента: apply и indices	310
Линеаризация списка списков: flatten	310
Объединение списков: zip и unzip	311
Отображение списков: toString и mkString	311

Преобразование списков: <code>iterator</code> , <code>toArray</code> , <code>copyToArray</code>	312
Пример: сортировка слиянием	313
14.7. Методы высшего порядка класса <code>List</code>	315
Отображения списков: <code>map</code> , <code>flatMap</code> и <code>foreach</code>	316
Фильтрация списков: <code>filter</code> , <code>partition</code> , <code>find</code> , <code>takeWhile</code> , <code>dropWhile</code> и <code>span</code>	317
Применение предикатов к спискам: <code>forall</code> и <code>exists</code>	318
Свертка списков: <code>foldLeft</code> и <code>foldRight</code>	319
Пример: реверсирование списков с помощью свертки	321
Сортировка списков: <code>sortWith</code>	322
14.8. Методы объекта <code>List</code>	323
Создание списков из их элементов: <code>List.apply</code>	323
Создание диапазона чисел: <code>List.range</code>	323
Создание единообразных списков: <code>List.fill</code>	324
Табулирование функции: <code>List.tabulate</code>	324
Конкатенация нескольких списков: <code>List.concat</code>	324
14.9. Совместная обработка нескольких списков	325
14.10. Понимание имеющегося в <code>Scala</code> алгоритма вывода типов	326
Резюме	330
Глава 15. Работа с другими коллекциями	331
15.1. Последовательности	331
Списки.	331
Массивы	332
Буферы списков.	333
Буферы массивов.	333
Строки (реализуемые через <code>StringOps</code>).	334
15.2. Множества и отображения	335
Использование множеств.	336
Применение отображений	338
Множества и отображения, используемые по умолчанию	340
Отсортированные множества и отображения	341
15.3. Выбор между изменяемыми или неизменяемыми коллекциями	342
15.4. Инициализация коллекций	344
Преобразование в массив или список.	346
Преобразования между изменяемыми и неизменяемыми множествами и отображениями	347
15.5. Кортежи	347
Резюме	349

Глава 16. Изменяемые объекты	350
16.1. Что делает объект изменяемым	350
16.2. Переназначаемые переменные и свойства	352
16.3. Практический пример: моделирование дискретных событий	356
16.4. Язык для цифровых схем	357
16.5. API моделирования	360
16.6. Моделирование электронной логической схемы	364
Класс Wire	365
Метод inverter	366
Методы andGate и orGate	367
Вывод симуляции.	368
Запуск симулятора.	368
Резюме	370
Глава 17. Иерархия Scala	372
17.1. Иерархия классов Scala	372
17.2. Как реализованы примитивы	376
17.3. Низшие типы	378
17.4. Определение собственных классов значений	379
Уход от монокультурности типов	380
17.5. Типы пересечений	382
17.6. Типы объединения	383
17.7. Прозрачные трейты	386
Резюме	387
Глава 18. Параметризация типов	388
18.1. Функциональные очереди	388
18.2. Соккрытие информации	392
Приватные конструкторы и фабричные методы	392
Альтернативный вариант: приватные классы	393
18.3. Аннотации вариантности	394
Вариантность и массивы	397
18.4. Проверка аннотаций вариантности	399
18.5. Нижние ограничители	402
18.6. Контравариантность	404
18.7. Верхние ограничители	407
Резюме	409

Глава 19. Перечисления	410
19.1. Перечисляемые типы данных	410
19.2. Алгебраические типы данных	414
19.3. Обобщенные ADT	416
19.4. Что делает типы ADT алгебраическими	417
Резюме	420
Глава 20. Абстрактные члены	421
20.1. Краткий обзор абстрактных членов	421
20.2. Члены-типы	422
20.3. Абстрактные val-переменные	423
20.4. Абстрактные var-переменные	424
20.5. Инициализация абстрактных val-переменных	425
Параметрические поля трейтов	427
Ленивые val-переменные	428
20.6. Абстрактные типы	431
20.7. Типы, зависящие от пути	434
20.8. Уточняющие типы	436
20.9. Практический пример: работа с валютой	437
Резюме	446
Глава 21. Гивены	447
21.1. Как это работает	448
21.2. Параметризованные given-типы	451
21.3. Анонимные given-экземпляры	455
21.4. Параметризованные given-экземпляры в виде классов типов	456
21.5. Импорт гивенов	459
21.6. Правила для контекстных параметров	461
21.7. Когда подходит сразу несколько гивенов	463
21.8. Отладка гивенов	465
Резюме	467
Глава 22. Методы расширения	468
22.1. Основы	468
22.2. Обобщенные расширения	471
22.3. Групповые расширения	472
22.4. Использование класса типов	474

22.5. Методы расширения для заданных экземпляров	477
22.6. Где Scala ищет методы расширения	480
Резюме	481
Глава 23. Классы типов	482
23.1. Зачем нужны классы типов	482
23.2. Границы контекста	487
23.3. Главные методы	490
23.4. Многостороннее равенство	494
23.5. Неявные преобразования	499
23.6. Пример использования класса типов: сериализация JSON	502
Резюме	510
Глава 24. Углубленное изучение коллекций	511
24.1. Изменяемые и неизменяемые коллекции	513
24.2. Согласованность коллекций	514
24.3. Трейт Iterable	516
Подкатегории Iterable	523
24.4. Трейты последовательностей Seq, IndexedSeq и LinearSeq	523
Буферы	528
24.5. Множества	530
24.6. Отображения	534
24.7. Конкретные классы неизменяемых коллекций	538
Списки.	539
Ленивые списки.	539
Неизменяемые ArraySeq.	540
Векторы.	541
Неизменяемые очереди	542
Диапазоны	543
Сжатые коллекции HAMT	543
Красно-черные деревья	544
Неизменяемые битовые множества	544
Векторные отображения	545
Списочные отображения	545
24.8. Конкретные классы изменяемых коллекций	546
Буферы массивов.	546
Буферы списков.	546
Построители строк.	547

ArrayDeque	547
Очереди	547
Стеки	548
Изменяемые ArraySeq	548
Хеш-таблицы	548
Слабые хеш-отображения	549
Совместно используемые отображения	550
Изменяемые битовые множества	550
24.9. Массивы	551
24.10. Строки	555
24.11. Характеристики производительности	556
24.12. Равенство	558
24.13. Представления	559
24.14. Итераторы	563
Буферизованные итераторы.	570
24.15. Создание коллекций с нуля	571
24.16. Преобразования между коллекциями Java и Scala	573
Резюме	575
Глава 25. Утверждения и тесты	576
25.1. Утверждения	576
25.2. Тестирование в Scala	578
25.3. Информативные отчеты об ошибках	579
25.4. Использование тестов в качестве спецификаций	581
25.5. Тестирование на основе свойств	584
25.6. Подготовка и проведение тестов	586
Резюме	587
Глоссарий	588
Библиография	604
Об авторах	607

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 сохранился 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 неprivатный член какого-либо объекта, в отношении которого в нем неявно определены методы `getter` и `setter`. Но названия таких методов отличаются от предписанных соглашениями `Java`. Метод получения значения (`getter`) `var`-переменной `x` называется просто `x`, а метод присваивания значения (`setter`) — `x_ =`.

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

```
var hour = 12
```

создает `getter` `hour` и `setter` `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 `getter` и `setter` обеспечивают исходной `var`-переменной некоторую видимость. Если `var`-переменная объявлена публичной (`public`), то таковыми же являются и `getter`, и `setter`. 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 `getter` и `setter`

```
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 некоей 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 переменной¹.

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.