Функциональные языки заставляют применять неизменяемые структуры данных и ссылочно прозрачные методы. В некоторых функциональных языках это выражено в виде категоричных требований. Scala же дает возможность выбрать. При желании можно писать программы в *императивном* стиле — так называется программирование с изменяемыми данными и побочными эффектами. Но при необходимости в большинстве случаев Scala позволяет с легкостью избежать использования императивных конструкций благодаря существованию хороших функциональных альтернатив.

1.3. Почему именно Scala

Подойдет ли вам язык Scala? Разбираться и принимать решение придется самостоятельно. Мы считаем, что, помимо хорошей масштабируемости, существует еще множество причин, по которым вам может понравиться программирование на Scala. В этом разделе будут рассмотрены четыре наиболее важных аспекта: совместимость, лаконичность, абстракции высокого уровня и расширенная статическая типизация.

Scala — совместимый язык

Scala не требует резко отходить от платформы Java, чтобы опередить на шаг этот язык. Scala позволяет повышать ценность уже существующего кода, то есть опираться на то, что у вас уже есть, поскольку он был разработан для достижения беспрепятственной совместимости с Java[1]. Программы на Scala компилируются в байт-коды виртуальной машины Java (JVM). Производительность при выполнении этих кодов находится на одном уровне с производительностью программ на Java. Код Scala может вызывать методы Java, обращаться к полям этого языка, поддерживать наследование от его классов и реализовывать его интерфейсы. Для всего перечисленного не требуются ни специальный синтаксис, ни явные описания интерфейса, ни какой-либо связующий код. По сути, весь код Scala интенсивно использует библиотеки Java, зачастую даже без ведома программистов.

Еще один аспект полной совместимости — интенсивное заимствование в Scala типов данных Java. Данные типа `Int` в Scala представлены в виде имеющегося в Java примитивного целочисленного типа `int`, соответственно `Float` представлен как `float`, `Boolean` — как `boolean` и т. д. Массивы Scala отображаются на массивы Java. В Scala из Java позаимствованы и многие стандартные библиотечные типы. Например, тип строкового литерала `"abc"` в Scala фактически представлен классом `java.lang.String`, а исключение должно быть подклассом `java.lang.Throwable`.

[1] Изначально существовала реализация Scala, запускаемая на платформе .NET, но она больше не используется. В последнее время все большую популярность набирает реализация Scala под названием Scala.js, запускаемая на JavaScript.

Java-типы в Scala не только заимствованы, но и «принаряжены» для придания им привлекательности. Например, строки в Scala поддерживают такие методы, как `toInt` или `toFloat`, которые преобразуют строки в целое число или число с плавающей точкой. То есть вместо `Integer.parseInt(str)` вы можете написать `str.toInt`. Как такое возможно без нарушения совместимости? Ведь в Java-классе `String` нет метода `toInt`! По сути, в Scala имеется универсальное решение для устранения противоречия между сложной структурой библиотеки и совместимостью. Scala позволяет определять *неявные преобразования*, которые всегда применяются при несовпадении типов или выборе несуществующих элементов. В рассматриваемом случае при поиске метода `toInt` для работы со строковым значением компилятор Scala не найдет такого элемента в классе `String`. Однако он найдет неявное преобразование, превращающее Java-класс `String` в экземпляр Scala-класса `StringOps`, в котором такой элемент определен. Затем преобразование будет автоматически применено, прежде чем будет выполнена операция `toInt`.

Код Scala также может быть вызван из кода Java. Иногда при этом следует учитывать некоторые нюансы. Scala — более богатый язык, чем Java, и потому некоторые расширенные функции Scala должны быть закодированы, прежде чем могут быть отображены на код Java. Подробности объясняются в главе 31.

Scala — лаконичный язык

Программы на Scala, как правило, отличаются краткостью. Программисты, работающие с данным языком, отмечают сокращение количества строк почти на порядок по сравнению с Java. Но это можно считать крайним случаем. Более консервативные оценки свидетельствуют о том, что обычная программа на Scala должна умещаться в половину тех строк, которые используются для аналогичной программы на Java. Меньшее количество строк означает не только сокращение объема набираемого текста, но и экономию сил при чтении и осмыслении программ, а также уменьшение количества возможных недочетов. Свой вклад в сокращение количества строк кода вносят сразу несколько факторов.

В синтаксисе Scala не используются некоторые шаблонные элементы, отягощающие программы на Java. Например, в Scala не обязательно применять точки с запятыми. Есть и несколько других областей, где синтаксис Scala менее зашумлен. В качестве примера можно сравнить, как записывается код классов и конструкторов в Java и Scala. В Java класс с конструктором зачастую выглядит следующим образом:

```
// Это Java
class MyClass {

    private int index;
    private String name;

    public MyClass(int index, String name) {
        this.index = index;
        this.name = name;
    }
}
```

А в Scala, скорее всего, будет использована такая запись:

```
class MyClass(index: Int, name: String)
```

Получив указанный код, компилятор Scala создаст класс с двумя приватными переменными экземпляра (типа `Int` по имени `index` и типа `String` по имени `name`) и конструктор, который получает исходные значения для этих переменных в виде параметров. Код данного конструктора проинициализирует две переменные экземпляра значениями, переданными в качестве параметров. Короче говоря, в итоге вы получите ту же функциональность, что и у более многословной версии кода на Java[1]. Класс в Scala быстрее пишется и проще читается, а еще — и это наиболее важно — допустить ошибку при его создании значительно труднее, чем при создании класса в Java.

Еще один фактор, способствующий лаконичности, — используемый в Scala вывод типов. Повторяющуюся информацию о типе можно отбросить, и тогда программы избавятся от лишнего и их легче будет читать.

Но, вероятно, наиболее важный аспект сокращения объема кода — наличие кода, не требующего внесения в программу, поскольку это уже сделано в библиотеке. Scala предоставляет вам множество инструментальных средств для определения эффективных библиотек, позволяющих выявить и вынести за скобки общее поведение. Например, различные аспекты библиотечных классов можно выделить в трейты, которые затем можно перемешивать произвольным образом. Или же библиотечные методы могут быть параметризованы с помощью операций, позволяя вам определять конструкции, которые, по сути, являются вашими собственными управляющими конструкциями. Собранные вместе, эти конструкции позволяют определять библиотеки, сочетающие в себе высокоуровневый характер и гибкость.

Scala — язык высокого уровня

Программисты постоянно борются со сложностью. Для продуктивного программирования нужно понимать код, над которым вы работаете. Чрезмерно сложный код был причиной краха многих программных проектов. К сожалению, важные программные продукты обычно бывают весьма сложными. Избежать сложности невозможно, но ею можно управлять.

Scala помогает управлять сложностью, позволяя повышать уровень абстракции в разрабатываемых и используемых интерфейсах. Представим, к примеру, что есть переменная `name`, имеющая тип `String`, и нужно определить, наличествует ли в этой строковой переменной символ в верхнем регистре. До выхода Java 8 приходилось создавать следующий цикл:

---

[1] Единственное отличие заключается в том, что переменные экземпляра, полученные в случае применения Scala, будут финальными (final). Как сделать их не финальными, рассказывается в разделе 10.6.

```
boolean nameHasUpperCase = false; // Это Java
for (int i = 0; i < name.length(); ++i) {
    if (Character.isUpperCase(name.charAt(i))) {
        nameHasUpperCase = true;
        break;
    }
}
```

А в Scala можно написать такой код:

```
val nameHasUpperCase = name.exists(_.isUpper)
```

Код Java считает строки низкоуровневыми элементами, требующими посимвольного перебора в цикле. Код Scala рассматривает те же самые строки как высокоуровневые последовательности символов, в отношении которых можно применять запросы с *предикатами*. Несомненно, код Scala намного короче и — для натренированного глаза — более понятен, чем код Java. Следовательно, код Scala значительно меньше влияет на общую сложность приложения. Кроме того, уменьшается вероятность допустить ошибку.

Предикат `_.isUpper` — пример используемого в Scala функционального литерала[1]. В нем дается описание функции, которая получает аргумент в виде символа (представленного знаком подчеркивания) и проверяет, не является ли этот символ буквой в верхнем регистре[2].

В Java 8 появилась поддержка *лямбда-выражений* и *потоков* (streams), позволяющая выполнять подобные операции на Java. Вот как это могло бы выглядеть:

```
boolean nameHasUpperCase = // Это Java 8
    name.chars().anyMatch(
        (int ch) -> Character.isUpperCase((char) ch)
);
```

Несмотря на существенное улучшение по сравнению с более ранними версиями Java, код Java 8 все же более многословен, чем его эквивалент на языке Scala. Излишняя тяжеловесность кода Java, а также давняя традиция использования в этом языке циклов может натолкнуть многих Java-программистов на мысль о необходимости новых методов, подобных `exists`, позволяющих просто переписать циклы и смириться с растущей сложностью кода.

В то же время функциональные литералы в Scala действительно воспринимаются довольно легко и задействуются очень часто. По мере углубления знакомства со Scala перед вами будут открываться все больше и больше возможностей для определения и использования собственных управляющих абстракций. Вы поймете, что это поможет избежать повторений в коде, сохраняя лаконичность и чистоту программ.

---

[1] Функциональный литерал может называться предикатом, если результирующим типом будет Boolean.

[2] Такое использование символа подчеркивания в качестве заместителя для аргументов рассматривается в разделе 8.5.

Scala — статически типизированный язык

Системы со статической типизацией классифицируют переменные и выражения в соответствии с видом хранящихся и вычисляемых значений. Scala выделяется как язык своей совершенной системой статической типизации. Обладая системой вложенных типов классов, во многом похожей на имеющуюся в Java, этот язык позволяет вам проводить параметризацию типов с помощью средств *обобщенного программирования*, комбинировать типы с использованием *пересечений* и скрывать особенности типов, применяя *абстрактные типы*[1]. Так формируется прочный фундамент для создания собственных типов, который дает возможность разрабатывать безопасные и в то же время гибкие в использовании интерфейсы.

Если вам нравятся динамические языки, такие как Perl, Python, Ruby или Groovy, то вы можете посчитать немного странным факт, что система статических типов в Scala упоминается как одна из его сильных сторон. Ведь отсутствие такой системы часто называют основным преимуществом динамических языков. Наиболее часто, говоря о ее недостатках, приводят такие аргументы, как присущая программам многословность, воспрепятствование свободному самовыражению программистов и невозможность применения конкретных шаблонов динамических изменений программных систем. Но зачастую эти аргументы направлены не против идеи статических типов в целом, а против конкретных систем типов, воспринимаемых как слишком многословные или недостаточно гибкие. Например, Алан Кей, автор языка Smalltalk, однажды заметил: «Я не против типов, но не знаю ни одной беспроблемной системы типов. Так что мне все еще нравится динамическая типизация»[2].

В этой книге я надеюсь убедить вас в том, что система типов в Scala далека от проблемной. На самом деле она вполне изящно справляется с двумя обычными опасениями, связываемыми со статической типизацией: многословия удается избежать за счет логического вывода типов, а гибкость достигается благодаря сопоставлению с образцом и ряду новых способов записи и составления типов. По мере устранения этих препятствий к классическим преимуществам систем статических типов начинают относиться намного более благосклонно. Среди наиболее важных преимуществ можно назвать верифицируемые свойства программных абстракций, безопасный рефакторинг и более качественное документирование.

Верифицируемые свойства. Системы статических типов способны подтверждать отсутствие конкретных ошибок, выявляемых в ходе выполнения программы. Это могут быть следующие правила: булевы значения никогда не складываются с целыми числами; приватные переменные недоступны за пределами своего класса;

---

[1] Обобщенные типы рассматриваются в главе 19, пересечения (например, A с B и с C) — в главе 12, а абстрактные типы — в главе 20.

[2] *Kay A. C.* Письмо, адресованное Стефану Раму, с описанием термина «Объектно-ориентированное программирование». Июль 2003 [Электронный ресурс]. — Режим доступа: www.purl.org/stefan_ram/pub/doc_kay_oop_en.

функции применяются к надлежащему количеству аргументов; во множество строк можно добавлять только строки.

Существующие в настоящее время системы статических типов не выявляют ошибки других видов. Например, обычно они не обнаруживают бесконечные функции, нарушение границ массивов или деление на ноль. Вдобавок эти системы не смогут определить несоответствие вашей программы ее спецификации (при наличии таковой!). Поэтому некоторые отказываются от них, считая не слишком полезными. Аргументация такова: если эти системы могут выявлять только простые ошибки, а модульные тесты обеспечивают более широкий охват, то зачем вообще связываться со статическими типами? Мы считаем, что в этих аргументах упущено главное. Система статических типов, конечно же, не может *заменить* собой модульное тестирование, однако может сократить количество необходимых модульных тестов, выявляя некие свойства, которые в противном случае нужно было бы протестировать. А модульное тестирование не способно заменить статическую типизацию. Ведь Эдсгер Дейкстра (Edsger Dijkstra) сказал, что тестирование позволяет убедиться лишь в наличии ошибок, но не в их отсутствии[1]. Гарантии, которые обеспечиваются статической типизацией, могут быть простыми, но это реальные гарантии, не способные обеспечить никакие объемы тестирования.

Безопасный рефакторинг. Системы статических типов дают гарантии, позволяющие вам вносить изменения в основной код, будучи совершенно уверенными в благополучном исходе этого действия. Рассмотрим, к примеру, рефакторинг, при котором к методу нужно добавить еще один параметр. В статически типизированном языке вы можете внести изменения, перекомпилировать систему и просто исправить те строки, которые вызовут ошибку типа. Сделав это, вы будете пребывать в уверенности, что были найдены все места, требовавшие изменений. То же самое справедливо для другого простого рефакторинга, например изменения имени метода или перемещения метода из одного класса в другой. Во всех случаях проверка статического типа позволяет быть вполне уверенными в том, что работоспособность новой системы осталась на уровне работоспособности старой.

Документирование. Статические типы — документация программы, проверяемой компилятором на корректность. В отличие от обычного комментария, аннотация типа никогда не станет устаревшей (по крайней мере, если содержащий ее исходный файл недавно успешно прошел компиляцию). Более того, компиляторы и интегрированные среды разработки (integrated development environments, IDE) могут использовать аннотации для выдачи более качественной контекстной справки. Например, IDE может вывести на экран все элементы, доступные для выбора, путем определения статического типа выражения, которое выбрано, и дать возможность просмотреть все элементы этого типа.

Хотя статические типы в целом полезны для документирования программы, иногда они могут вызывать раздражение тем, что засоряют ее. Обычно полезным

---

[1] *Dijkstra E. W.* Notes on Structured Programming. — Апрель 1970 [Электронный ресурс]. — Режим доступа: www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF.

считается документирование тех сведений, которые читателям программы самостоятельно извлечь довольно трудно. Полезно знать, что в методе, определенном так:

```
def f(x: String) = ...
```

аргументы метода `f` должны принадлежать типу `String`. В то же время может вызвать раздражение по крайней мере одна из двух аннотаций в следующем примере:

```
val x: HashMap[Int, String] = new HashMap[Int, String]()
```

Понятно, что было бы достаточно показать отношение `x` к типу `HashMap` с `Int`-типами в качестве ключей и `String`-типами в качестве значений только один раз, дважды повторять одно и то же нет смысла.

В Scala имеется весьма сложная система логического вывода типов, позволяющая опускать почти всю информацию о типах, которая обычно вызывает раздражение. В предыдущем примере вполне работоспособны и две менее раздражающие альтернативы:

```
val x = new HashMap[Int, String]()
val x: Map[Int, String] = new HashMap()
```

Вывод типа в Scala может заходить довольно далеко. Фактически пользовательский код нередко вообще обходится без явного задания типов. Поэтому программы на Scala часто выглядят похожими на программы, написанные на динамически типизированных языках скриптов. Это, в частности, справедливо для прикладного клиентского кода, который склеивается из заранее написанных библиотечных компонентов. Но для них самих это менее характерно, поскольку в них зачастую применяются довольно сложные типы, не допускающие гибкого использования таких схем. И это вполне естественно. Ведь сигнатуры типов элементов, составляющих интерфейс повторно используемых компонентов, должны задаваться в явном виде, поскольку составляют существенную часть соглашения между компонентом и его клиентами.

1.4. Истоки Scala

На идею создания Scala повлияли многие языки программирования и идеи, выработанные на основе исследований таких языков. Фактически обновления в Scala незначительны — большинство характерных особенностей языка уже применялось в том или ином виде в других языках программирования. Инновации в Scala появляются в основном из того, как его конструкции сводятся воедино. В этом разделе будут перечислены основные факторы, оказавшие влияние на структуру языка Scala. Перечень не может быть исчерпывающим, поскольку в дизайне языков программирования так много толковых идей, что перечислить здесь их все просто невозможно.

На внешнем уровне Scala позаимствовал существенную часть синтаксиса у Java и C#, которые, в свою очередь, взяли большинство своих синтаксических соглаше-

ний у C и C++. Выражения, инструкции и блоки — в основном из Java, как, собственно, и синтаксис классов, создание пакетов и импорт[1]. Кроме синтаксиса, Scala позаимствовал и другие элементы Java, такие как его основные типы, библиотеки классов и модель выполнения.

Scala многим обязан и другим языкам. Его однородная модель объектов впервые появилась в Smalltalk и впоследствии была принята языком Ruby. Его идея универсальной вложенности (почти каждую конструкцию в Scala можно вложить в любую другую) реализована также в Algol, Simula, а в последнее время в Beta и gbeta. Его принцип единообразного доступа к вызову методов и выбору полей пришел из Eiffel. Его подход к функциональному программированию очень близок по духу к применяемому в семействе языков ML, видными представителями которого являются SML, OCaml и F#. Многие функции высшего порядка в стандартной библиотеке Scala присутствуют также в ML или Haskell. Толчком для появления в Scala неявных параметров стали классы типов языка Haskell — в более классическом объектно-ориентированном окружении они дают аналогичные результаты. Используемая в Scala основная библиотека многопоточного вычисления на основе акторов — Akka — создавалась под сильным влиянием особенностей языка Erlang.

Scala — не первый язык, в котором акцент сделан на масштабируемости и расширяемости. Исторические корни расширяемых языков для различных областей применения обнаруживаются в статье Петера Ландина (Peter Landin) 1966 года The Next 700 Programming Languages («Следующие 700 языков программирования»)[2]. (Описываемый в ней язык Iswim, наряду с Lisp, — один из первых в своем роде функциональных языков.) Происхождение конкретной идеи трактовки инфиксного оператора в качестве функции можно проследить до Iswim и Smalltalk. Еще одна важная идея заключается в разрешении использования функционального литерала (или блока) в качестве параметра, позволяющего библиотекам определять управляющие конструкции. Она также проистекает из Iswim и Smalltalk. И у Smalltalk, и у Lisp довольно гибкий синтаксис, широко применявшийся для создания закрытых предметно-ориентированных языков. Еще один масштабируемый язык, C++, который может быть адаптирован и расширен благодаря применению перегрузки

---

[1] Главное отличие от Java касается синтаксиса для объявления типов: вместо «Тип переменная», как в Java, задействуется форма «переменная: Тип». Используемый в Scala постфиксный синтаксис типа похож на синтаксис, применяемый в Pascal, Modula-2 или Eiffel. Основная причина такого отклонения имеет отношение к логическому выводу типов, зачастую позволяющему опускать тип переменной или тип возвращаемого методом значения. Легче использовать синтаксис «переменная: Тип», поскольку двоеточие и тип можно просто не указывать. Но в стиле языка C, применяющем форму «Тип переменная», просто так не указывать тип нельзя, поскольку при этом исчезнет сам признак начала определения. Неуказанный тип в качестве заполнителя требует какое-нибудь ключевое слово (C# 3.0, в котором имеется логический вывод типов, для этой цели задействует ключевое слово var). Такое альтернативное ключевое слово представляется несколько более надуманным и менее привычным, чем подход, который используется в Scala.

[2] *Landin P. J.* The Next 700 Programming Languages // Communications of the ACM, 1966. — № 9 (3). — P. 157–166.

операторов и своей системе шаблонов, построен, по сравнению со Scala, на низкоуровневой и более системно-ориентированной основе. Кроме того, Scala — не первый язык, объединяющий в себе функциональное и объектно-ориентированное программирование, хотя, вероятно, в этом направлении продвинулся гораздо дальше прочих. К числу других языков, объединивших некоторые элементы функционального программирования с объектно-ориентированным, относятся Ruby, Smalltalk и Python. Расширения Java-подобного ядра некоторыми функциональными идеями были предприняты на Java-платформе в Pizza, Nice, Multi-Java и самом Java 8. Существуют также изначально функциональные языки, которые приобрели систему объектов. В качестве примера можно привести OCaml, F# и PLT-Scheme.

В Scala применяются также некоторые нововведения в области языков программирования. Например, его абстрактные типы — более объектно-ориентированная альтернатива обобщенным типам, его трейты позволяют выполнять гибкую сборку компонентов, а экстракторы обеспечивают независимый от представления способ сопоставления с образцом. Эти нововведения были представлены в статьях на конференциях по языкам программирования в последние годы[1].

Резюме

Ознакомившись с текущей главой, вы получили некоторое представление о том, что представляет собой Scala и как он может помочь программисту в работе. Разумеется, этот язык не решит все ваши проблемы и не увеличит волшебным образом вашу личную продуктивность. Следует заранее предупредить, что Scala нужно применять искусно, а для этого потребуется получить некоторые знания и практические навыки. Если вы перешли к Scala от языка Java, то одними из наиболее сложных аспектов его изучения для вас могут стать система типов Scala, которая существенно богаче, чем у Java, и его поддержка функционального стиля программирования. Цель данной книги — послужить руководством при поэтапном, от простого к сложному, изучении особенностей Scala. Полагаем, что вы приобретете весьма полезный интеллектуальный опыт, расширяющий ваш кругозор и изменяющий взгляд на проектирование программных средств. Надеемся, что вдобавок вы получите от программирования на Scala истинное удовольствие и познаете творческое вдохновение.

В следующей главе вы приступите к написанию кода Scala.

---

[1] Для получения большей информации см.: *Odersky. M., Cremet V., Röckl C., Zenger M.* A Nominal Theory of Objects with Dependent Types // In Proc. ECOOP'03, Springer LNCS. 2003. July. — P. 201–225; *Odersky M., Zenger M.* Scalable Component Abstractions // Proceedings of OOPSLA. 2005. October. — P. 41–58; *Emir B., Odersky M., Williams J.* Matching Objects With Patterns // Proc. ECOOP, Springer LNCS. 2007. July. — P. 273–295.