1 Современный язык Java

В ЭТОЙ ГЛАВЕ

- ✓ Java как платформа и язык
- ✓ Новый цикл выпуска Java
- ✓ Расширенное выведение типов (`var`)
- ✓ Изменения языка и платформы
- ✓ Инкубационная и предварительная функциональность
- ✓ Незначительные изменения языка в Java 11

Добро пожаловать в мир Java, где не перестают происходить увлекательные события. В сентябре 2021 года была выпущена Java 17 — наиболее свежая (на момент написания книги) версия с долгосрочной поддержкой (LTS, Long-Term Support), и самые бесстрашные команды начали переходить на нее[1].

На момент написания книги, если не считать отдельных первопроходцев, приложения Java более или менее поровну распределены между Java 11 (выпущенной в сентябре 2018 года) и намного более старой Java 8 (2014). Java 11 предлагает много интересных возможностей, особенно для команд, которые развертывают свои продукты в облаке, но многие все еще не спешат на нее переходить.

[1] 19 сентября 2023 вышла Java 21 — следующая после 17-й LTS-версия. — *Примеч. ред.*

В части I книги мы потратим немного времени, чтобы рассказать о новых возможностях, которые появились в Java 11 и 17. Хочется надеяться, что этот материал убедит некоторые команды и руководителей, которым неохота мигрировать с Java 8, что в новых версиях все гораздо лучше.

В этой главе мы сосредоточимся на Java 11, потому что: 1) это LTS-версия с наибольшей долей рынка и 2) массовый переход на Java 17 еще не произошел. Однако в главе 3 мы расскажем о новых возможностях Java 17, чтобы вы были в курсе нынешнего положения дел.

Начнем с дуализма «язык или платформа», который лежит в основе современного Java. Это чрезвычайно важная тема, к которой мы будем неоднократно возвращаться в книге, поэтому необходимо с самого начала понять суть.

1.1. ЯЗЫК И ПЛАТФОРМА

Термин «Java» может относиться к нескольким взаимосвязанным предметам. В частности, он может означать как язык программирования, на котором люди пишут код, так и гораздо более широкое понятие — «платформа Java».

Как ни странно, разные авторы иногда по-разному определяют, что считать языком, а что — платформой. Это приводит к недоразумениям, и люди начинают путаться в том, чем различаются эти две сущности и какая из них отвечает за те или иные программные возможности, которые используются в коде приложения.

Давайте проясним это различие прямо сейчас, потому что на него опираются многие темы из этой книги. Вот наши определения:

- *Язык Java* — объектно-ориентированный язык со статической типизацией, над которым мы слегка поиронизировали в разделе «Об этой книге». Надеемся, что он вам уже хорошо знаком. У исходного кода, написанного на языке Java, есть одна очевидная особенность: он понятен для человека (или по крайней мере должен быть таковым!).
- *Платформа Java* — это программная система, которая предоставляет среду выполнения. Ключевой компонент этой среды — JVM (виртуальная машина Java), которая компонует и выполняет ваш код, а код она получает в форме файлов классов (которые не предназначены для чтения человеком). JVM не интерпретирует исходный код на языке Java напрямую, а требует, чтобы сначала он был преобразован в файлы классов.

Один из важных факторов успеха Java как программной системы — ее стандартизация. Иначе говоря, у Java есть спецификации, которые описывают, как она должна работать. Стандартизация позволяет разным компаниям и проектным группам создавать реализации, которые будут работать одинаково (по крайней мере в теории). Спецификации не дают никаких гарантий относительно того, какую производительность покажут разные реализации на одной и той же задаче, но стандарты могут гарантировать правильность результатов.

Систему Java регламентируют несколько разных спецификаций, самые важные из которых — JLS (Java Language Specification, спецификация языка Java) и JVM (VMSpec). Это разделение играет важную роль в современном мире Java; более того, в VMSpec больше нет никаких прямых ссылок на JLS. Позже мы еще вернемся к различиям между этими двумя спецификациями.

ПРИМЕЧАНИЕ В наши дни JVM фактически представляет собой независимую от языка среду общего назначения для выполнения программ. Это одна из причин разделять спецификации.

Когда вы сталкиваетесь с этим дуализмом, возникает очевидный вопрос: «А как тогда связаны эти понятия?» Если их разделили, то как тогда они вместе формируют систему Java?

Язык и платформа связаны друг с другом через общее определение формата файлов классов (файлов .class). Вы не пожалеете, если обстоятельно изучите определение файлов классов (соответствующий материал ждет вас в главе 4), — собственно, это один из способов превратиться из хорошего Java-программиста в выдающегося. На рис. 1.1 изображен полный процесс того, как производится и используется код на Java.

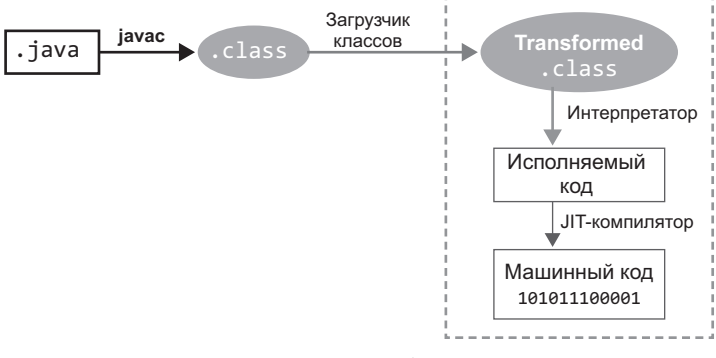

Рис. 1.1. Исходный код на Java преобразуется в файлы .class, а затем обрабатывается во время загрузки перед JIT-компиляцией

Как видно на этой схеме, код на Java начинает свое существование в виде человекочитаемого исходного кода, после чего он с помощью `javac` компилируется в файл .class и загружается в JVM. В процессе загрузки классы часто обрабатываются и изменяются. Многие популярные фреймворки Java преобразуют классы в процессе загрузки, внедряя в них динамическое поведение, например механизмы измерительного контроля или альтернативные контексты поиска загружаемых классов.

ПРИМЕЧАНИЕ Загрузка классов — чрезвычайно важная функция платформы Java, и в главе 4 вы узнаете о ней намного подробнее.

Java — это компилируемый или интерпретируемый язык? Обычно его представляют как язык, который компилируется в файлы .class перед тем, как запускаться на JVM. Если допытываться настойчивее, многие разработчики также пояснят, что байт-код изначально интерпретируется JVM, но проходит JIT-компиляцию на более поздней стадии. Однако здесь представление многих людей ограничивается довольно расплывчатой концепцией байт-кода как фактически машинного кода для воображаемого или упрощенного процессора.

На самом деле байт-код JVM скорее занимает промежуточное положение между исходным кодом, понятным для человека, и машинным кодом. В терминах теории компиляторов байт-код представляет собой разновидность промежуточного языка (IL, Intermediate Language), а не реальный машинный код. То есть преобразование исходного кода на Java в байт-код не является компиляцией в том смысле, в каком ее понимает программист на C++ или Go, а `javac` — это не компилятор вроде `gcc`, а генератор, который создает файлы классов из исходного кода на Java. Настоящий компилятор в экосистеме Java — это JIT-компилятор, как показано на рис. 1.1.

Иногда систему Java описывают как «динамически компилируемую». Этот термин подчеркивает, что реальная компиляция — это JIT-компиляция во время выполнения, а не создание файла класса на этапе сборки.

> **ПРИМЕЧАНИЕ** Из-за того что в Java есть компилятор исходного кода javac, многие разработчики воспринимают Java как статический, компилируемый язык. Далеко не все в курсе, что на стадии выполнения среда Java в действительности чрезвычайно динамична — просто это не заметно на первый взгляд.

Таким образом, правильный ответ на вопрос, является ли Java компилируемым или интерпретируемым языком, звучит так: «И тем и другим».

Разобравшись с различиями между языком и платформой, поговорим о новом цикле выпуска Java.

1.2. НОВЫЙ ЦИКЛ ВЫПУСКА JAVA

Java не всегда был языком с открытым исходным кодом, но после объявления на конференции JavaOne в 2006 году исходный код самого Java (кроме нескольких фрагментов, которые не принадлежали компании Sun) был опубликован на условиях лицензии GPLv2+CE (https://openjdk.java.net/legal/gplv2+ce.html).

Это случилось во времена Java 6, так что Java 7 стала первой версией, которая распространялась по лицензии OSS (Open Source Software). С тех пор разработка открытого исходного кода платформы Java сосредоточена вокруг проекта OpenJDK (https://openjdk.java.net).

Обсуждение проекта происходит в основном в списках рассылки, каждый из которых посвящен отдельному аспекту всей кодовой базы. Существуют как «постоянные» рассылки (например, `core-libs` — базовые библиотеки), так

и временные, которые создаются ради конкретных проектов OpenJDK вроде `lambda-dev` (лямбда-функции) и перестают быть активными, когда проект завершается. По большей части эти списки рассылки стали полезными форумами, где можно обсуждать возможную будущую функциональность, что позволяет широкому кругу разработчиков участвовать в создании новых версий Java.

> **ПРИМЕЧАНИЕ** Компания Sun Microsystems была приобретена Oracle незадолго до выпуска Java 7. Таким образом, все выпуски Java от Oracle основаны на кодовой базе с открытым исходным кодом.

Новые версии Java с открытым исходным кодом выпускались по мере готовности функциональности: фактически выпуск определялся одной доминирующей функциональной возможностью, например лямбда-функции в Java 8 или модули в Java 9.

Однако после Java 9 принцип выпусков изменился. Компания Oracle решила, что, начиная с Java 10, язык Java будет выпускаться по строгому графику. Это значит, что OpenJDK теперь опирается на магистральную модель разработки, которая подразумевает следующее:

- Новые возможности разрабатываются в отдельной ветви, а слияние выполняется только после стабилизации кода.
- Выпуски происходят с жесткой периодичностью.
- Запаздывающие возможности не задерживают выпуск, а переносятся на следующий выпуск.
- Текущая основная ветвь разработки всегда должна быть пригодной к выпуску (теоретически).
- Если необходимо экстренное исправление, его можно подготовить и выпустить в любой момент.
- Отдельные проекты OpenJDK используются для экспериментов и исследования перспективных направлений.

Новые версии Java выпускаются через каждые шесть месяцев. Разные поставщики (Oracle, Eclipse Adoptium, Amazon, Azul и т. д.) могут объявить *любой* из этих выпусков версией с долгосрочной поддержкой (LTS, Long Term Support). Однако на практике все поставщики традиционно назначают LTS-версию раз в три года.

> **ПРИМЕЧАНИЕ** В конце 2021 года обсуждалась идея уменьшить периодичность LTS с трех лет до двух. Возможно, следующей LTS-версией станет Java 21 в 2023 году (вместо Java 23 в 2024-м)[1].

Первой LTS-версией стала Java 11, а Java 8 была включена в число LTS-версий задним числом. Компания Oracle предполагала, что сообщество Java будет охотно

[1] Так и вышло. — *Примеч. ред.*

поспевать за обновлениями и осваивать новые выпуски по мере выхода. Однако на практике сообщество (особенно корпоративные клиенты) не приняло такой подход, предпочитая обновляться с одной LTS-версии на следующую. Конечно, при этом новые возможности Java осваиваются медленнее и нововведения пробуксовывают. Однако такова реальность корпоративных продуктов, и многие все еще рассматривают обновление версии Java как весьма масштабное мероприятие.

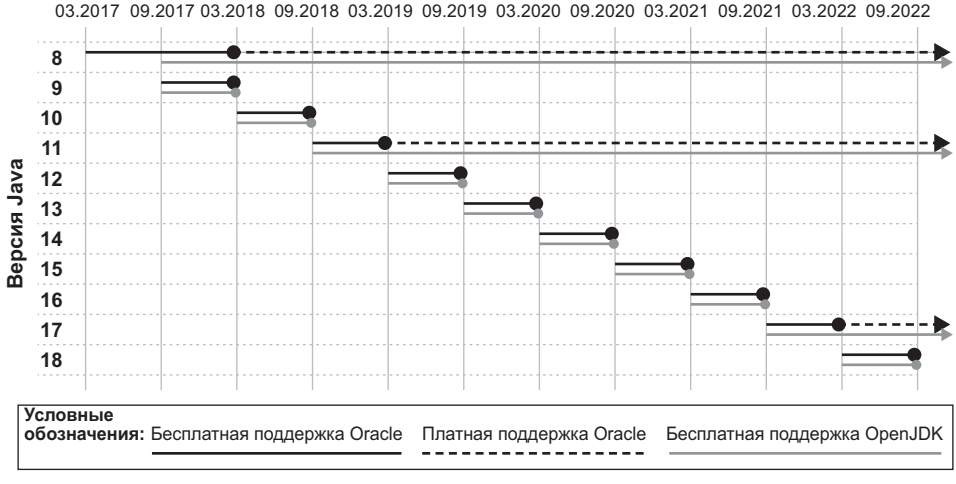

Рис. 1.2. Временна́я шкала недавних и будущих выпусков Java

Это означает, что хотя на графике на рис. 1.2 раз в полгода значится основной выпуск, на практике широко применяются только LTS-версии: Java 17 (сентябрь 2021 года), Java 11 (сентябрь 2018 года) и Java 8, выпущенная до появления модулей (в 2014 году). Java 8 и Java 11 занимают примерно одинаковую долю рынка, хотя Java 11 в последнее время захватила более 50 % рынка и быстро наращивает популярность. Ожидается, что переход на Java 17 произойдет намного быстрее, чем переход с Java 8 на Java 11, потому что самые серьезные препятствия, связанные с системой модулей и ограничениями безопасности, уже были преодолены при предыдущей миграции.

Другое важное изменение в новой модели выпуска заключается в том, что компания Oracle пересмотрела лицензию на распространение своих продуктов. Хотя JDK от Oracle строится из исходного кода OpenJDK, на двоичные файлы не распространяется лицензия OSS. JDK от Oracle — закрытое ПО, и по состоянию на JDK 11 Oracle предоставляет поддержку и обновления для каждой версии только на шесть месяцев. Это означает, что многие люди, которые полагались на бесплатные обновления Oracle, теперь сталкиваются с выбором:

- платить Oracle за поддержку и обновления или
- использовать другой дистрибутив, который предоставляет двоичные файлы на условиях открытой лицензии.

Среди альтернативных поставщиков JDK — Eclipse Adoptium (ранее AdoptOpenJDK), Alibaba (Dragonwell), Amazon (Corretto), Azul Systems (Zulu), IBM, Microsoft, Red Hat и SAP.

> **ПРИМЕЧАНИЕ** Двое авторов этой книги (Мартин и Бен) стояли у истоков проекта AdoptOpenJDK, который позднее развился в независимый проект сообщества Eclipse Adoptium, цель которого — выпускать высококачественные бесплатные двоичные дистрибутивы Java с открытым кодом. За дополнительной информацией обращайтесь на сайт adoptium.net.

При меняющихся лицензиях и таком количестве поставщиков выбрать правильную версию Java для вас и вашей команды оказывается весьма ответственным решением. К счастью, авторитетные лидеры экосистемы Java написали подробные руководства по этой теме, а в приложении А приведена их сводка.

Хотя модель выпуска Java перешла на выпуск по графику, подавляющее большинство команд разработчиков все еще используют JDK 8 или 11. Эти LTS-выпуски сопровождаются сообществом (в том числе крупными фирмами-разработчиками), и для них все еще регулярно публикуются обновления безопасности и исправления ошибок. Изменения в LTS-версиях намеренно имеют небольшую область действия и относятся к «хозяйственным» обновлениям. Помимо исправлений, касающихся безопасности и мелких ошибок, разрешен лишь минимальный набор изменений: в основном это исправления, которые необходимы, чтобы обеспечить правильную работу LTS-выпусков в течение их предполагаемого срока эксплуатации. В частности, сюда относятся:

- добавление новой японской эры;
- обновления базы данных часовых поясов;
- TLS 1.3;
- добавление Shenandoah — сборщика мусора с низкой задержкой для крупных современных нагрузок.

Еще одно необходимое изменение — обновление сценариев сборки для macOS, чтобы они работали с новейшей версией среды Xcode от Apple, а значит, продолжали функционировать в новых выпусках macOS.

В проектах по сопровождению JDK 8 и 11 (которые иногда называются «проектами обновлений») все еще сохраняется потенциальная возможность обратно портировать новую функциональность, но она минимальна. Например, одно из правил гласит, что портируемая функциональность не должна нарушать семантику программ. Примеры допустимых изменений — поддержка TLS 1.3 или обратное портирование Java Flight Recorder в Java 8u272. После того как мы прояснили различия между языком и платформой и разобрались с новой моделью выпусков, давайте перейдем к техническим возможностям современного Java. О первой возможности, с которой мы познакомимся, разработчики просили почти с самого первого выпуска Java ведь она позволяет сократить объем кода при написании программ.

1.3. РАСШИРЕННОЕ ВЫВЕДЕНИЕ ТИПОВ (СИНТАКСИС VAR)

Java традиционно имеет репутацию весьма пространного языка. Тем не менее в последних версиях язык эволюционировал так, чтобы шире использовать *выведение типов* (type inference). Оно позволяет компилятору исходного кода определить часть нужной информации автоматически, чтобы вам не приходилось указывать все явно.

> **ПРИМЕЧАНИЕ** Выведение типов позволяет сократить объем шаблонного кода, устранить повторения и сделать код более компактным и удобочитаемым.

Эта тенденция началась в Java 5, где были добавлены обобщенные методы, которые поддерживают крайне ограниченную форму выведения типа для аргументов обобщенных типов. Вместо того чтобы явно указывать нужный тип, как в следующем примере:

```
List<Integer> empty = Collections.<Integer>emptyList();
```

параметр обобщенного типа в правой части можно было опустить:

```
List<Integer> empty = Collections.emptyList();
```

Такая запись вызова обобщенных методов стала настолько привычной, что многие разработчики с трудом вспомнят форму с явными аргументами типов. И это хорошо: значит, выведение типов справляется со своей задачей и избавляет код от лишних шаблонных конструкций, чтобы он стал понятнее.

Следующее важное улучшение выведения типов появилось в Java 7, где изменилась работа с обобщениями. До Java 7 такой код встречался очень часто:

```
Map<Integer, Map<String, String>> usersLists =
                        new HashMap<Integer, Map<String, String>>();
```

Здесь вы крайне развернуто объявляете, что у вас есть какие-то пользователи, которые идентифицируются по `userid` (целое число), и что у каждого пользователя есть специфический для него набор свойств (представленный как отображение строк на строки).

Но на самом деле почти половина этого кода — дублирующиеся символы, которые не несут никакой полезной информации. Начиная с Java 7, можно использовать такую запись:

```
Map<Integer, Map<String, String>> usersLists = new HashMap<>();
```

При этом компилятор сам определяет правильный тип для выражения в правой части — он не ограничивается подстановкой текста, определяющего полный тип.

> **ПРИМЕЧАНИЕ** Так как пара угловых скобок в сокращенном объявлении типа напоминает бубновую масть, эта форма называется «бубновым синтаксисом» (diamond syntax).

В Java 8 выведение типов расширилось до поддержки лямбда-выражений. Например, здесь алгоритм выведения типов может сделать вывод, что `s` имеет тип `String`:

```
Function<String, Integer> lengthFn=s-> s.length();
```

В современном Java выведение типов сделало еще один шаг вперед: появилось выведение типов локальных переменных (LVTI, Local Variable Type Inference), которое также обозначается `var`. Эта возможность, добавленная в Java 10, позволяет автоматически выводить типы *переменных* вместо типов *значений*, как в следующем примере:

```
var names = new ArrayList<String>();
```

Это достигается благодаря тому, что `var` сделано зарезервированным, «волшебным» именем типа, а не ключевым словом языка. Теоретически `var` по-прежнему можно использовать в качестве имени переменной, метода или пакета.

> **ПРИМЕЧАНИЕ** У правильного использования `var` есть важный побочный эффект: на передний план выводится предметная область вашего кода, а не информация о типе. Но кому много дано, с того много и спросится! Тщательно выбирайте имена переменных, чтобы помочь будущим читателям вашего кода.

С другой стороны, код, в котором ранее `var` использовалось как имя типа, придется перекомпилировать. Впрочем, практически все разработчики Java следуют соглашению о том, что имена типов должны начинаться с буквы в верхнем регистре, так что, по всей видимости, количество экземпляров существующих типов с именем `var` пренебрежимо мало. Это означает, что формально ничто не мешает написать код вроде приведенного в следующем листинге.

Листинг 1.1. Плохой код

```
package var;

public class Var {
  private static Var var = null;

  public static Var var() {
    return var;
  }

  public static void var(Var var) {
    Var.var = var;
  }
}
```

А потом вызвать его таким образом:

```
var var = var();
if (var == null) {
  var(new Var());
}
```

Тем не менее «ничто не мешает» еще не означает «так стоит делать». Писать такой код, как в листинге 1.1, — верный способ нажить врагов; кроме того, такой код наверняка не пройдет код-ревью!

Синтаксис `var` предназначен для того, чтобы сократить объем кода на Java, и он выглядит привычно для программистов, которые приходят в Java из других языков. Он не вводит в язык динамическую типизацию, и каждой переменной Java по-прежнему соответствует статический тип — просто этот тип не всегда обязательно указывать явно.

Выведение типов в Java действует *локально*, и в случае с `var` алгоритм проверяет только объявление локальной переменной. Это означает, что `var` нельзя использовать для полей, аргументов методов или возвращаемых типов. Компилятор применяет разновидность *анализа ограничений* (constraint solving), чтобы определить, существует ли тип, который удовлетворял бы всем требованиям кода в том виде, в каком он записан.

> **ПРИМЕЧАНИЕ** `var` реализуется исключительно компилятором исходного кода (`javac`) и никак не влияет на выполнение или быстродействие.

Например, в объявлении `lengthFn` в недавнем примере анализатор ограничений способен заключить, что тип параметра `s` метода должен быть совместим с типом `String`, который явно указан как тип параметра `Function`. Конечно, в Java тип `String` — конечный (`final`), поэтому компилятор выводит, что типом `s` является именно `String`.

Чтобы компилятор мог выводить типы, программист должен предоставить достаточно информации, которая позволит решить систему ограничивающих уравнений. Например, такой код:

```
var fn = s -> s.length();
```

не содержит достаточно информации для того, чтобы компилятор мог вывести тип `fn`, так что компилироваться он не будет. Стоит отметить важный частный случай:

```
var n = null;
```

Компилятор не может вывести тип, потому что значение `null` можно присвоить переменной любого ссылочного типа, так что нельзя однозначно заключить, какой тип должна принимать `n`. В таких случаях говорят, что соответствующая система уравнений ограничения типов является *недоопределенной*. Этот математический термин означает, что решаемых уравнений меньше, чем неизвестных.

Можно представить себе такое выведение типов, когда анализируется не только исходное объявление локальных переменных, но и другие участки кода:

```
var n = null;
String.format(n);
```

Более сложный алгоритм выведения (или человек) смог бы сделать вывод, что `n` в этом примере относится к типу `String`, потому что метод `format()` получает строку в первом аргументе.

На первый взгляд идея выглядит заманчиво, но как это часто бывает в программировании, она связана с компромиссами. Чем сложнее алгоритм, тем больше времени требуется на компиляцию и тем больше шансов на неудачу при выведении. В свою очередь, это означает, что программисту придется развивать более тонкую интуицию, чтобы правильно обращаться с выведением нелокальных типов.

В других языках могут быть другие компромиссы, но в Java ситуация предельно однозначна: для выведения типов используются только объявления. Выведение типов локальных переменных призвано сократить объем шаблонного кода и избавиться от дублирования. Однако выведение стоит применять только там, где оно помогает сделать код яснее, а не как грубый инструмент, который применяется для всех задач без разбора (см. антипаттерн «Золотой молоток» (Golden Hammer)).

Например, LVTI полезно в таких случаях:

- В простых инициализаторах, где правая часть — вызов конструктора или статического фабричного метода.
- Если удаление явного типа позволяет исключить повторяющуюся или избыточную информацию.
- Если имена переменных уже обозначают их типы.
- Если локальная переменная действует и используется в простой и узкой области.

Стюарт Маркс (Stuart Marks), один из ключевых разработчиков языка Java, представляет полный набор практических правил использования LTVI в руководстве по адресу http://mng.bz/RvPK.

Завершая этот раздел, рассмотрим более сложный вариант использования `var` — так называемые *необозначаемые* (nondenotable) типы. Они допустимы в Java, однако их нельзя указывать как тип переменной, а можно только выводить как тип присваиваемого выражения. Рассмотрим простой пример с использованием интерактивной среды `jshell`, которая появилась в Java 9:

```
jshell> var duck = new Object() {
   ...> void quack() {
   ...>     System.out.println("Кря-кря!");
   ...> }
   ...> }
duck ==> $0@5910e440

jshell> duck.quack();
Кря-кря!
```

У переменной `duck` необычный тип: фактически это `Object`, но расширенный методом с именем `quack()`. Хотя объект может крякать как утка, у его типа нет

имени, поэтому этот тип нельзя использовать ни как параметр метода, ни как тип возвращаемого значения.

С LVTI этот тип может фигурировать как выводимый тип локальной переменной, что позволяет использовать его внутри метода. Конечно, такой тип недоступен за пределами этой узкой локальной области видимости, так что его существование — скорее технический курьез, чем полезная возможность.

Несмотря на ограничения LVTI, оно дает некоторое представление о том, как Java подходит к возможности, которая есть в других языках и обычно называется *структурной типизацией* в языках со статической типизацией и *утиной типизацией* в языках с динамической типизацией (например, Python).

1.4. ИЗМЕНЕНИЯ ЯЗЫКА И ПЛАТФОРМЫ

На наш взгляд, важно понимать не только *что* изменилось в языке, но и *почему*. Многие из тех, кто следит за разработкой новых версий Java, активно интересуются новыми возможностями языка, однако не всегда понимают, сколько работы требуется, чтобы полноценно реализовать соответствующие изменения и подготовить их для массового использования.

Возможно, вы также заметили, что в такой зрелой среде выполнения, как Java, возможности языка нередко приходят из других языков или библиотек, развиваются в популярных фреймворках и только потом добавляются в сам язык или среду выполнения. Мы надеемся пролить свет на эти процессы, а заодно рассеять несколько сопутствующих мифов. Но если эволюция Java вас не особо интересует, переходите прямо к разделу 1.5, где речь пойдет об изменениях в языке.

В развитии языка Java одни изменения требуют больше инженерных усилий, а другие — меньше. На рис. 1.3 мы попытались представить, какие бывают типы изменений и насколько затратно реализовывать каждый из них по сравнению с остальными.

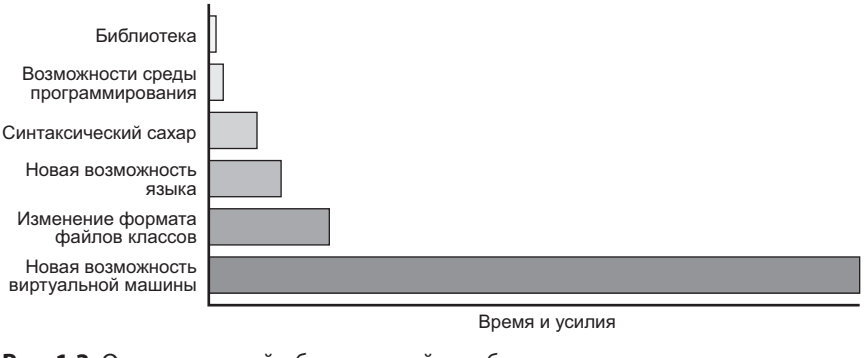

Рис. 1.3. Относительный объем усилий, необходимых для реализации разных типов новой функциональности

В общем случае лучше выбирать путь, который требует меньше усилий. Это означает, что если новую возможность можно реализовать в виде библиотеки, обычно так и стоит сделать. Но многие решения нелегко, а то и вообще невозможно реализовать в виде библиотек или функций IDE. Ряд возможностей приходится внедрять на более глубоком уровне внутри платформы. Вот как некоторые из недавних нововведений можно классифицировать по нашей шкале сложности:

- *Изменения в библиотеке*: фабричные методы коллекций (Java 9).
- *Синтаксический сахар*: подчеркивания в числах (Java 7).
- *Небольшая новая возможность языка*: `try`-с-ресурсами (Java 7).
- *Изменение формата файлов классов*: аннотации (Java 5).
- *Новая возможность JVM*: гнезда и соседи (nestmates) (Java 11).
- *Крупное обновление языка*: лямбда-выражения (Java 8).

Давайте поближе познакомимся с тем, как вносятся изменения на разных уровнях шкалы сложности.

1.4.1. Синтаксический сахар

Синтаксическим сахаром (syntactic sugar) называются удобства, которые упрощают программистам работу с уже существующей функциональностью. Как правило, на ранней стадии компиляции синтаксический сахар удаляется из внутреннего представления программы на уровне компилятора.

Таким образом, изменения из категории синтаксического сахара легко реализовать, потому что обычно они требуют относительно небольшого объема работы, а изменения вносятся только в компилятор (`javac` в случае Java).

Возможно, к этому моменту у вас появился вопрос: «Какие изменения в спецификации считаются незначительными?» Одним из самых прямолинейных изменений в Java 7 стало добавление единственного слова — `String` — в раздел 14.11 спецификации JLS, что позволило использовать строки в инструкциях `switch`. Трудно придумать изменение, которое было бы еще мельче, однако даже эта поправка затронула ряд других аспектов спецификации. Всякое нововведение ведет к последствиям, которые приходится отслеживать по всей структуре языка.

1.4.2. Изменения языка

Для *любого* изменения необходимо выполнить полный набор действий (или по крайней мере проанализировать, нужно ли выполнять каждое из них):

- Обновить JLS.
- Реализовать прототип в компиляторе исходного кода.
- Добавить соответствующую библиотечную поддержку.

- Написать тесты и примеры.
- Обновить документацию.

Кроме того, если изменение затрагивает платформу или JVM, необходимы также следующие действия:

- Обновить VMSpec.
- Реализовать изменения в JVM.
- Добавить поддержку изменений в инструменты JVM и файлы классов.
- Проанализировать последствия для рефлексии.
- Проанализировать последствия для сериализации.
- Изучить, как изменение повлияет на низкоуровневые программные компоненты, например JNI (Java Native Interface).

Немалый объем работы — и это после того, как вы проанализировали последствия изменений по всей спецификации языка!

Одна из проблемных областей при внесении изменений — система типов. Дело не в том, что эта система в Java чем-то плоха; просто в языках с богатыми системами статических типов обычно оказывается много потенциальных точек взаимодействия между разными составляющими этих систем. Изменения в них могут быть чреваты неприятными неожиданностями.

1.4.3. Документы JSR и JEP

Для внесения изменений в платформу Java используются два основных механизма, и первый из них — *JSR* (Java Specification Request, «запрос на спецификацию Java»), который подчиняется процедуре *JCP* (Java Community Process, «процесс сообщества Java»). С помощью этого метода определяются стандартные API — как внешние библиотеки, так и основные внутренние API платформы.

Исторически это был единственный способ вносить изменения в платформу Java, и он лучше всего подходил, чтобы закрепить консенсус по поводу уже зрелой технологии. Однако в последние годы стремление реализовывать изменения быстрее (и в меньших масштабах) привело к тому, что появилась упрощенная альтернатива — документы *JEP* (JDK Enhancement Proposal, «предложение по улучшению JDK»). Кумулятивные платформенные документы JSR теперь состоят из JEP, ориентированных на следующую версию Java. Процесс JSR обеспечивает дополнительную защиту интеллектуальной собственности для всей экосистемы.

Обсуждая новые возможности Java, часто бывает удобно ссылаться на будущую или недавно внедренную функциональность по номеру JEP. Полный список всех JEP (в том числе реализованных и отозванных) можно найти по адресу https://openjdk.java.net/jeps/0.

1.4.4. Инкубационная и предварительная функциональность

В новой модели выпуска Java появились два механизма, которые позволяют опробовать предлагаемые возможности перед тем, как закрепить их в более поздней версии. Цель этих механизмов — обеспечить улучшенную функциональность благодаря обратной связи от гораздо более широкого круга пользователей, сохраняя возможность изменить или аннулировать функциональность до того, как она станет неотъемлемой частью Java.

К *инкубационной функциональности* (incubating features) относятся новые API и их реализация, которая в простейшей форме представляет собой новый API, поставляемый в виде автономного модуля (модули Java подробнее рассматриваются в главе 2). Имя модуля выбирается так, чтобы оно ясно указывало, что это временный API. Когда функциональность окончательно утвердится, имя меняется на постоянное.

> **ПРИМЕЧАНИЕ** Это означает, что если код опирается на нефинальную версию инкубационной функциональности, то в него придется вносить изменения, когда она станет финальной.

Характерный пример инкубационной функциональности — поддержка версии 2 протокола HTTP (обычно обозначаемой HTTP/2). В Java 9 она присутствовала в виде инкубационного модуля `jdk.incubator.http`. Имя модуля в сочетании с пространством имен `jdk.incubator` вместо `java` четко обозначало, что эта возможность еще не утверждена и может измениться. В Java 11 эта функциональность была стандартизирована и перемещена в модуль `java.net.http` в пространстве имен `java`.

> **ПРИМЕЧАНИЕ** Еще один пример инкубационной функциональности встретится в главе 18, когда мы будем обсуждать Foreign Access API — часть проекта OpenJDK с кодовым названием Panama.

Главное преимущество этого подхода — в том, что инкубационную функциональность можно изолировать в одном пространстве имен. Разработчики могут ее быстро опробовать и даже использовать в промышленном коде — при условии, что они готовы внести изменения в код, а также перекомпилировать и перекомпоновать его, когда функциональность будет стандартизирована.

Предварительная функциональность (preview features) — другой механизм, с помощью которого свежие версии Java предоставляют возможности, которые еще не окончательно утверждены. Эта функциональность изолирована в меньшей степени, чем инкубационная, потому что реализуется как часть самого языка на более глубоком уровне. Теоретически для этого может понадобиться поддержка на следующих уровнях:

- компилятор `javac`;
- формат байт-кода;
- файлы классов и загрузка классов.

Предварительная функциональность доступна, только если передать компилятору и среде выполнения конкретные флаги. Если пытаться использовать предварительную функциональность без флагов, это вызовет ошибку как на стадии компиляции, так и во время выполнения.

В результате работать с предварительной функциональностью значительно сложнее, чем с инкубационной, и поэтому ее невозможно использовать в промышленном коде. Начнем хотя бы с того, что предварительная функциональность представляется соответствующей версией формата файла класса, которая не финализирована и, возможно, не будет поддерживаться ни в одной эксплуатационной версии Java.

Таким образом, предварительная функциональность подходит только для экспериментов, внутреннего тестирования и ознакомления. К сожалению, почти во всех вариантах развертывания кода, предназначенного для реальной эксплуатации, можно использовать только полностью финализированную функциональность.

В Java 11 нет никакой предварительной функциональности (хотя в Java 12 появилась первая предварительная версия *switch-выражений*), так что в этом разделе трудно привести соответствующий пример. Мы подробнее рассмотрим предварительную функциональность в главе 3, когда пойдет речь о Java 17.

1.5. НЕБОЛЬШИЕ ИЗМЕНЕНИЯ В JAVA 11

В выпусках, последовавших за Java 8, появилось довольно много небольших нововведений. Давайте кратко пройдемся по самым важным из них, однако имейте в виду, что здесь мы рассмотрим далеко не все изменения.

1.5.1. Фабрики коллекций (JEP 213)

Часто звучали предложения расширить Java так, чтобы язык поддерживал простой способ объявления *литералов коллекций* — списков, отображений и т. д. Эта идея выглядит привлекательно, потому что во многих других языках такая возможность поддерживается в том или ином виде, и в самом Java всегда были литералы массивов:

```
jshell> int[] numbers = {1, 2, 3};
numbers ==> int[3] { 1, 2, 3 }
```

Но несмотря на всю внешнюю привлекательность, добавить эту возможность на уровне языка оказывается не так-то просто. Например, хотя разработчикам прекрасно знакомы реализации `ArrayList`, `HashMap` и `HashSet`, основополагающий принцип проектирования коллекций Java гласит, что они должны представляться как интерфейсы, а не как классы. Также доступны другие реализации, которые применяются достаточно широко.

А это значит, что если бы появился новый синтаксис, который напрямую связан с конкретными реализациями (какими бы популярными они ни были), это пошло

бы вразрез с идеологией языка. Вместо этого было решено добавить в соответствующие интерфейсы простые фабричные методы; при этом использовался тот факт, что в Java 8 стало возможно включать статические методы в интерфейсы. В результате получается код следующего вида:

```
Set<String> set = Set.of("a", "b", "c");

var list = List.of("x", "y");
```

Хотя запись оказывается более громоздкой, чем при гипотетической поддержке на уровне языка, накладная сложность с точки зрения реализации оказывается значительно ниже. Новые методы реализуются в виде набора перегруженных версий:

```
List<E> List<E>.<E>of()
List<E> List<E>.<E>of(E e1)
List<E> List<E>.<E>of(E e1, E e2)
List<E> List<E>.<E>of(E e1, E e2, E e3)
List<E> List<E>.<E>of(E e1, E e2, E e3, E e4)
List<E> List<E>.<E>of(E e1, E e2, E e3, E e4, E e5)
List<E> List<E>.<E>of(E e1, E e2, E e3, E e4, E e5, E e6)
List<E> List<E>.<E>of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)
List<E> List<E>.<E>of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)
List<E> List<E>.<E>of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9)
List<E> List<E>.<E>of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9,
  E e10)
List<E> List<E>.<E>of(E... elements)
```

Определены наиболее частые случаи (до 10 элементов), а также форма с переменным числом аргументов (vararg, вариадическая) для маловероятного сценария, когда в коллекции должно быть более 10 элементов.

Для отображений ситуация немного усложняется, потому что у них есть два обобщенных параметра — тип ключа и тип значения. А значит, хотя простые случаи можно записывать в виде:

```
var m1 = Map.of(k1, v1);
var m2 = Map.of(k1, v1, k2, v2);
```

нет простого способа записать для отображения подобную форму с переменным числом аргументов. Вместо такой формы используется другой фабричный метод — `ofEntries()` в сочетании со статическим вспомогательным методом `entry()`:

```
Map.ofEntries(
    entry(k1, v1),
    entry(k2, v2),
    // ...
    entry(kn, vn));
```

И последнее, что стоит учитывать разработчикам, — фабричные методы порождают экземпляры неизменяемых типов:

```
jshell> var ints = List.of(2, 3, 5, 7);
ints ==> [2, 3, 5, 7]

jshell> ints.getClass();
$2 ==> class java.util.ImmutableCollections$ListN
```

Эти классы — новые реализации интерфейсов коллекций Java, которые являются неизменяемыми. Не путайте их со знакомыми изменяемыми классами, такими как `ArrayList` и `HashMap`. При попытке изменить экземпляры этих типов возбуждается исключение.

1.5.2. Удаление корпоративных модулей (JEP 320)

Долгое время в стандартном выпуске Java (Java SE) был ряд модулей, которые на самом деле относились к корпоративному выпуску (Java EE), в том числе:

- JAXB
- JAX-WS
- CORBA
- JTA

В Java 9 следующие пакеты, реализовавшие эти технологии, были перемещены в неосновные модули и объявлены устаревшими, чтобы их можно было удалить:

- `java.activation` (JAF)
- `java.corba` (CORBA)
- `java.transaction` (JTA)
- `java.xml.bind` (JAXB)
- `java.xml.ws` (JAX-WS, а также некоторые сопутствующие технологии)
- `java.xml.ws.annotation` (Common Annotations)

В ходе дальнейшей оптимизации платформы в Java 11 эти модули были удалены. Также из основного дистрибутива Java SE были исключены следующие три модуля со вспомогательными инструментами и механизмом агрегирования:

- `java.se.ee` (модуль-агрегатор для шести модулей, перечисленных выше)
- `jdk.xml.ws` (инструменты для JAX-WS)
- `jdk.xml.bind` (инструменты для JAXB)

Если в проектах, созданных в Java 11 и выше, приходится пользоваться этими возможностями, то требуется подключить явную внешнюю зависимость. Это означает, что некоторые программы, зависящие от этих API, не испытывают проблем со сборкой в Java 8, но в Java 11 сценарий их сборки приходится модифицировать. Эта проблема подробнее рассматривается в главе 11.