1

История Java в двух словах

Содержание главы

- Этапы развития Java
- Различие ключевых версий Java
- Пометки об устаревании (Deprecation) и удаление
- Переименования
- Принципы внесения изменений

ВВЕДЕНИЕ

Вы освоили Java и готовы применять его на практике. Или уже используете Java в работе, но обескуражены экосистемой, о которой не говорилось в учебных курсах. Эта книга для тех, кто уже научился писать код на Java и хочет подняться на следующую ступень мастерства. Будь вы студент, специалист, меняющий профессию, или опытный разработчик, осваивающий новый язык, — это ваш шанс изучить экосистему Java.

Книга не претендует на роль всеобъемлющего руководства — для этого потребовалось бы несколько толстенных томов. Начинающим мы рекомендуем сначала обратиться к изданиям *Head First Java*[1]*, 3rd Edition* (O'Reilly Media, 2022) или *Java For Dummies*[2] (For Dummies, 2025). Затем возвращайтесь к этой книге. Ее цель — познакомить читателя с наиболее распространенными фреймворками, инструментами и методиками, которые используются в корпоративной

[1] На русском языке: *Сьерра К., Бейтс Б.* Изучаем Java.

[2] На русском языке: *Бейтс Б.* Java для чайников.

Java-разработке, предоставив достаточно базовой информации и примеров, чтобы сразу погрузиться в тему.

В первой главе мы рассмотрим общую информацию о Java, а в следующих перейдем к специализированным темам. Хотя книгу можно читать в произвольном порядке, мы рекомендуем сначала изучить главы 1–4 для получения базовых сведений.

КАК СКАЧАТЬ КОД К ЭТОЙ ГЛАВЕ

Исходный код к этой главе доступен на https://github.com/realworldjava/Ch01-History. За подробностями обратитесь к файлу `README.md` этого репозитория.

ЭТАПЫ РАЗВИТИЯ JAVA

Язык Java создал и в 1995 году представил миру Джеймс Гослинг (James Gosling) из Sun Microsystems.

Согласно легенде, Java начинался как Oak — новый язык разработки встроенного ПО для смарт-устройств. Идея была новаторской: не писать ПО для каждой встроенной ОС, а разработать компилируемый объектно-ориентированный язык по принципу «Написано однажды — работает везде», который бы генерировал байт-код, и создать для каждой платформы интерпретатор байт-кода, который мог бы его выполнять.

Параллельно было решено включить в язык встроенные фреймворки, которые в других языках поставлялись как внешние модули, — например, средства обеспечения конкурентности (concurrency) или конструкторы пользовательского интерфейса. Завершающими штрихами стали очистка памяти методом сборки мусора, а также оптимизация с помощью JIT-компиляции (Just-In-Time, компиляция на лету) и — вуаля — есть все что нужно для платформы встроенного языка.

Но тут, увы, обнаружилось, что на платформу под названием Oak имеет права другая компания. Долгий вечер в кофейне за кружкой горячего Java — и все прочие варианты ушли в историю.

Много лет спустя, в 2009 году, компанию Sun приобрела корпорация Oracle, получив право собственности на Java. И по сей день Oracle распоряжается и поддерживает Java.

Сообщество Java активно поддерживает обратную совместимость языка, а строгая система тестирования Technology Compatibility Kit (TCK), включающая тысячи обязательных проверок для каждой новой функции, обеспечивает стабильность и надежность. Благодаря этому Java — один из самых успешных языков за всю историю программирования.

В корпорации Oracle осознают высокую ответственность за развитие и сохранение ценностей Java, учитывая его массовое использование. В состав JCP

(Java Community Process) входит Исполнительный комитет (JCP Executive Committee) — по сути, состоящий из множества организаций совет директоров Java, призванный обеспечить представительство сообщества. Одни участники комитета — конкуренты со своими собственными комплектами разработки Java (JDK, Java Development Kit): Amazon, Azul и Microsoft. Другие взаимно дополняют друг друга — в частности, поставщики интегрированных сред разработки (IDE), такие как Eclipse и JetBrains. Есть и просто убежденные сторонники Java, вроде BellSoft и Fujitsu. Наконец, члены сообщества, в частности, SouJava — бразильская группа Java-пользователей — или Кен Фогель (Ken Fogel), индивидуальный участник.

Одно из изменений, введенных под руководством Oracle, — более частый и предсказуемый график релизов. Обратите внимание: на рис. 1.1 видно, что прежде версии выпускались неравномерно, через разные промежутки, в разные месяцы. Однажды пользователям Java пришлось ждать новой версии более пяти лет.

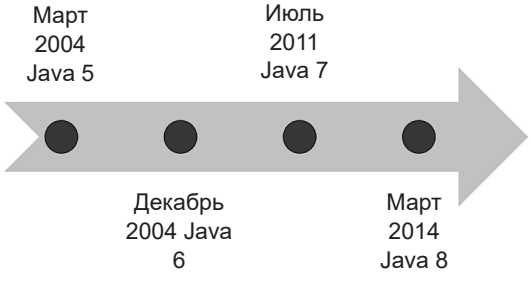

Рис. 1.1. Релизы Java

С новой периодизацией релизов каждая следующая версия Java выходит через шесть месяцев после предыдущей. Если вы скажете, что множество компаний не хотят обновляться каждые шесть месяцев, то будете правы! Есть и менее частотные релизы, именуемые *релизами с долгосрочной поддержкой* (Long-Term Support, LTS). Когда новый график только ввели, LTS-релизы выходили раз в три года. Сейчас они выпускаются раз в два года.

В табл. 1.1 представлены версии Java, выпущенные с момента введения нового графика релизов и до публикации этой книги. Можно убедиться, что релизы теперь выходят регулярно, в предсказуемом ритме. Мы будем использовать Java 21 — последняя LTS-версия на момент выхода книги.

На рис. 1.2 показан график выпуска LTS-релизов. Благодаря установившемуся распорядку теперь можно прогнозировать даты их выхода, что позволяет компаниям планировать обновления.

Многие компании запускают свои программы Java удаленно, например через веб-сайты. До 2020 года это зачастую осуществляли через *Spring* или *Java EE*

(Enterprise Edition). Подробности в главе 6 «Знакомство со Spring Framework» и в главе 14 «Знакомство с экосистемой Java». В 2020 году Oracle передала управление Java EE фонду открытого исходного кода. А поскольку «Java» — торговая марка, то фреймворк Java EE в связи с этим был переименован в Jakarta EE.

Таблица 1.1. Релизы Java

ГОД	МАРТОВСКИЕ РЕЛИЗЫ	СЕНТЯБРЬСКИЕ РЕЛИЗЫ
2017	Н/д (график был введен с сентября)	Java 9
2018	Java 10	**Java 11 (LTS)**
2019	Java 12	Java 13
2020	Java 14	Java 15
2021	Java 16	**Java 17 (LTS)**
2022	Java 18	Java 19
2023	Java 20	**Java 21 (LTS)**
2024	Java 22	Java 23

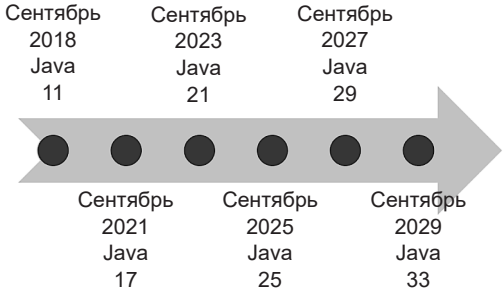

Рис. 1.2. График LTS-релизов с расчетом предстоящих выпусков

РАЗЛИЧИЯ КЛЮЧЕВЫХ ВЕРСИЙ JAVA

С течением времени язык Java существенно изменился. Глядя на ключевые функции, появившиеся с течением времени, можно ощутить эту эволюцию и убедиться, что этот язык будет продолжать совершенствоваться. Более того, вы узнаете, чего ожидать, если вы работаете на не самой свежей версии.

Обратите внимание: в этом разделе приведен далеко не полный список функций. Мы выбрали наиболее значимые функции каждой рассматриваемой версии.

Возможно, вас удивит, что мы не рассматриваем в этом разделе вывод типов с использованием ключевого слова `var`. Хотя оба автора этой книги порой используют `var` в рабочем коде, мы сочли, что будет понятнее, если указать в книге, какие есть типы, и убедиться, что представлены все импорты.

Кодирование дженериков в Java 5

До Java 5 компилятор не мог определить, что именно вы намеревались поместить в список. В Java 5 появились *дженерики* (*generic*), позволяющие явно определять тип — в данном примере это `String`:

```
1:  import java.util.ArrayList;
2:  import java.util.List;
3:
4:  public class Java5Generics {
5:
6:     private static List<String> createList() {
7:        List<String> buildingNames = new ArrayList<String>();
8:        buildingNames.add("Firehouse");
9:        return buildingNames;
10:    }
11:
12:    public static void main(String[] args) {
13:       List<String> buildingNames = createList();
14:
15:       System.out.println(buildingNames);  // [Firehouse]
16:    }
17: }
```

Теперь, когда мы проинформировали компилятор о своих намерениях, он стал нашим помощником. При попытке добавить `Integer`, `LocalDate` или что-то еще в `buildingNames` компилятор выдаст ошибку.

Вы можете задаваться вопросом, почему в строке 7 используется более длинная версия вместо `new ArrayList<>()`. Дело в том, что оператор «ромб» (`<>`) появился в Java 7, а этот пример относится к Java 5. Но даже это демонстрирует эволюцию языка!

Функциональное программирование в Java 8

Функциональное программирование позволяет писать код в более декларативном стиле. Вы скорее конкретизируете, что нужно сделать, чем управляете состоянием объектов. Вы больше фокусируетесь на выражениях, чем на циклах.

В Java 8 появились лямбда-выражения и ссылки на методы, которые упрощают написание кода *отложенного выполнения*, известного также как «код, который запускается позже». Они оба позволяют легко передавать исполняемый код в метод. В коде ниже можно заметить два лямбда-выражения и ссылку на метод:

```
1:  import java.util.Arrays;
2:  import java.util.List;
3:
4:  public class Java8FunctionalProgramming {
5:
6:    public static void main(String[] args) {
```

```
7:
8:        List<String> computerCompanies = Arrays.asList(
9:           "Dell", "Acer", "Microsoft", "IBM", "Lenovo");
10:       computerCompanies.stream()
11:          .filter(n -> n.length() > 5)
12:          .map(n -> "*" + n)
13:          .sorted()
14:          .forEach(System.out::println);
15:    }
16: }
```

Этот код выводит следующие две строки:

```
*Lenovo
*Microsoft
```

Строка 10 запускает конвейер потока (stream pipeline). В строках 11 и 12 используются лямбда-выражения, чтобы указать, что должно происходить на каждом шаге. Символ `->` подсказывает, что перед вами лямбда-выражение. Строка 11 пропускает через конвейер только те названия, что состоят как минимум из пяти символов. Строка 12 добавляет звездочку в начало каждой строки конвейера. Обратите внимание, что исходный список `computerCompanies` остается неизменным.

Далее, в строке 14 можно заметить первую ссылку на метод. Символ `::` говорит нам, что это ссылка на метод, который в данном случае выводит значения. Ссылка на метод — более компактная версия лямбда-выражения. Эквивалентом будет запись `n -> System.out.println(n)`.

Возможно, вы заметили, что в строке 8 мы вызывали `Arrays.asList()`, а не `List.of()`, который появился только в Java 9!

Модульное программирование — начиная с Java 11

Модульная система платформы Java (Java Platform Module System, JPMS) группирует код на более высоком уровне, чем пакеты. Основное назначение модуля — предоставлять группы связанных пакетов, предлагая разработчикам определенный набор функциональных возможностей. Это похоже на JAR-файл, за исключением того, что разработчик выбирает, какие пакеты доступны вне модуля. Примечание: многие компании предпочитают не использовать модули, так как их применение опционально.

Java-модули появились в Java 9, а не в Java 11. Но мы представляем их в разделе Java 11, так как это был первый LTS-релиз с поддержкой модулей. Все проекты на версиях ниже Java 8 обновят до Java 11, 17, 21 и т. д., минуя 9-ю версию, поскольку поддержка Java 9 прекращена в марте 2018 года, спустя 6 месяцев после ее релиза в сентябре 2017-го.

Модуль — это Java-архив (JAR), состоящий из файла `module-info.java` в корне проекта и одного или нескольких Java-пакетов. Например, вот модуль с двумя пакетами, каждый из которых содержит один класс:

```
book-module
| -- module- info.class
| -- com
   | -- wiley
      | -- realworldjava
         | -- modulecode
            | -- internal
               | -- PrivateHelper.class
            | -- utils
               | -- BookUtils.class
```

В примере показаны файлы с расширением `.class`, а не `.java`, поскольку JAR-файл содержит исполняемый байт-код. Но подождите. Вы же не хотите, чтобы вызывающие объекты вашего модуля ссылались на `PrivateHelper`. Вот тут-то и вступает в игру файл `module-info`:

```
module com.wiley.realworldjava.modulecode {
   exports com.wiley.realworldjava.modulecode.utils;
   requires java.sql;
}
```

Ключевое слово `module` сообщает Java, что мы указываем здесь модуль, а не `class`, `interface` или другой тип верхнего уровня Java. Директива `exports` разрешает внешнему коду напрямую обращаться к пакету `utils`. Поскольку пакет `internal` не указан в файле module-info, внешний код может обращаться к нему только опосредованно — через `BookUtils`. Наконец, директива `requires` указывает, что мы используем модуль `java.sql`, который поставляет Oracle в составе JDK. Есть еще модуль`java.base`, но он подключается автоматически, не требуя, чтобы его указывали. Он содержит распространенные классы, такие как `String` и `List`.

Программирование текстовых блоков и записей — начиная с Java 17

В Java 17 трудно было выбрать какое-то одно самое интересное нововведение, поэтому мы выбрали два. Текстовые блоки (text blocks) также известны как *multiline strings*. Они появились в Java 15. Но, как и в случае с модулями, мы рассматриваем их в контексте первого представляющего их LTS-релиза. Вот пример текстового блока, обрамленного тремя двойными кавычками (`"`):

```
1:  public class Java17TextBlocks {
2:
3:     public static void main(String[] args) {
4:
5:        String data = """
6:           Apple,Fruit
```

```
7:             Banana,Fruit
8:           Potato,Vegetable
9:           """;
10:
11:       String formatted = String.format("*%s*", data);
12:       System.out.println(formatted);
13:    }
14: }
```

Он выводит эти четыре строки:

```
*Apple,Fruit
 Banana,Fruit
Potato,Vegetable
*
```

Обратите внимание, что отступы текстового блока сохраняются, а случайные пробелы, которые используются для форматирования в IDE, — нет. Также обратите внимание, что тройные кавычки в строке 9 добавят пустую строку к имеющейся строке. Чтобы избежать этого, можно закрыть блок тремя двойными кавычками (") в конце строки 8.

Записи (records) были представлены в Java 17 и избавили от массы шаблонного кода при создании неизменяемого класса. В этом примере используется простая запись:

```
1:  public class Java17Records {
2:
3:     record Coordinate(double x, double y) {}
4:
5:     public static void main(String[] args) {
6:
7:        Coordinate coord = new Coordinate(1.2, 5.9);
8:        System.out.println(coord.x());
9:        System.out.println(coord);
10:    }
11: }
```

Выводится следующее:

```
1.2
Coordinate[x=1.2, y=5.9]
```

Строка 3 создает запись с двумя полями. Запись подобна неизменяемому классу, который включает:

- переменную экземпляра для каждого из перечисленных параметров;
- геттеры (без префикса `get`/`is`);
- конструктор, принимающий параметр для каждой переменной экземпляра;
- методы `equals()` и `hashCode()`, включающие значение каждой переменной экземпляра;

- метод `toString()`, который выводит каждое поле записи в удобном и легкочитаемом формате.

В строке 8 продемонстрировано использование геттера `x()` (без префикса `getX()`), а строка 9 неявно вызывает `toString()`, показывая, что можно получить полезную реализацию, не написав ни строчки кода.

Записи удобны, если нужен простой неизменяемый класс. Можно переопределять методы в теле записи или добавлять свои. Если требуется более детальная настройка или сеттеры, например, для JPA-сущностей (Java Persistence API entities), вероятно, в большей степени ответит вашим потребностям Project Lombok. Подробнее см. главу 8 «Project Lombok: улучшение кода с помощью аннотаций».

Виртуальные потоки в Java 21

Давным-давно в Java для обеспечения конкурентности использовался класс Thread. Хотя он по-прежнему остается низкоуровневой единицей, обеспечивающей конкурентность, позже были добавлены класс `Executors` и более мощные инструменты конкурентности вроде fork/join и `parallelStream()`. Однако параллелизация — это не бесплатно: помимо ресурсов памяти, которые требуют сами потоки, будут и затраты на настройку и координацию потоков.

По мере роста в компьютерах количества центральных процессоров (CPU), параллелизация становится все важнее. В Java 21 появились виртуальные потоки (virtual threads), что радикально снизило стоимость конкурентности. Более подробное объяснение и пример виртуальных потоков вы найдете в главе 9 «Параллелизация приложения с использованием конкурентности Java».

ЧТО ДЕЛАТЬ С УСТАРЕВАЮЩИМИ ЭЛЕМЕНТАМИ И КАК ИХ УДАЛЯТЬ

В Java постоянно добавляется все больше новых элементов, а как же удалять устаревшие? Сначала разработчики помечают классы или методы как *устаревшие* (*deprecated*), это означает, что дальше их использовать не рекомендовано, но все еще допустимо. Скажем, вам надо, чтобы пользователи прекратили передавать параметр в `magicNumber()`.

```
1:  public class DeprecationExample {
2:      /**
3:      * Returns a number
4:      * @deprecated USe{@link #getNumber()} instead
5:      * @param num число
6:      * @return number based on num
7:      */
8:      @Deprecated(forRemoval = true)
9:      public int getNumber(int num) {
```

```
10:        return num % 2;
11:     }
12:
13:     public int getNumber() {
14:        return 42;
15:     }
16: }
```

Javadoc-тег `@deprecated` в строке 4 применен, чтобы указать в документации, что использовать этот метод не рекомендуется. А тег `@link` здесь, чтобы пояснить, что предпочтительнее вызывать `magicNumber()`[1] без параметра.

В строке 8 показана аннотация, маркирующая метод как `@Deprecated`. IDE отображают случаи использования устаревшего кода в виде предупреждений, чтобы привлечь к ним ваше внимание. Больше информации об IDE ищите в главе 2 «Освоение IDE: секрет успеха».

В строке 8 мы также видим атрибут `forRemoval`, который появился с Java 9. Он указывает, планируется ли полное удаление устаревшего кода или просто рекомендуется использовать альтернативные API. Это позволяет разработчикам принимать взвешенные решения о необходимости миграции.

Перечень Oracle удаленных API хранится на странице https://docs.oracle.com/en/java/javase/21/migrate/removed-apis.html. Там указано, что было удалено в каждой версии — с Java 9 до Java 21. Например, в Java 21 удален один из двух API:

```
java.lang.ThreadGroup.allowThreadSuspension(boolean)
```

Вряд ли вы пользовались этим методом. Oracle ценит обратную совместимость и крайне осторожно подходит к удалению кода, который широко используется. На самом деле некоторые методы в `java.util.Date` устарели еще со времен Java 1.1!

ВЫЯВЛЕНИЕ ПЕРЕИМЕНОВАНИЙ

В процессе эволюции экосистемы Java были сделаны ключевые переименования, о которых важно знать. Этот раздел поможет определить, актуальны ли старые руководства и документация или они уже устарели.

Преобразование в Jakarta EE

Как уже упоминалось, в 2020 году Oracle объявил о переходе с Java EE на Jakarta EE. Поскольку это было не так давно, документация с упоминанием Java EE все еще более-менее полезна. В конце концов, руководство по Java EE-*сервлетам*

[1] В тексте оригинала написано `magicNumber()`, но в коде используется `getNumber()`. — *Примеч. пер.*

(*servlet*) для Java EE и Jakarta EE практически идентичны. И концепции, описанные в грамотных руководствах, по-прежнему будут применяться.

Ключевое различие состоит в том, что в Jakarta EE 9 префикс имени пакета переименован с `javax` на `jakarta`, так как Java — торговая марка Oracle. Кроме того, новые функции описываются исключительно в документации Jakarta EE.

Переименование сертификаций

У Oracle был ряд сертификаций, которые можно было получить, — в частности, Sun Certified Java Associate (SCJA), Sun Certified Java Professional (SCJP) и Sun Certified Master Java Developer (SCMJD). Все эти сертификации переименованы, их названия начинаются с «O», в связи с Oracle, — и теперь у нас есть такие сертификации, как OCJA, OCJP, OCMJD. Просматривая резюме, помните, что эти сертификации эквивалентны друг другу.

Многие из этих сертификаций упразднены или значительно переработаны с момента переименования, так что вряд ли обучение по веб-сайтам со старыми названиями сертификаций (Sun) будет продуктивным.

ПРИНЦИПЫ ВНЕСЕНИЯ ИЗМЕНЕНИЙ

Изменения в Java происходят постоянно, благодаря чему язык остается актуальным и полезным. Некоторые изменения вдохновлены другими языками. С каждым релизом можно ожидать улучшения языка и появления новых API.

Изменения Java-экоситемы обусловлены тремя основными факторами.

- *Исправление ошибок* (*Bug fixes*): в Java оперативно исправляют баги, зарегистрированные в базе данных ошибок. Хотя разработчики редко впрямую сталкиваются с такими изменениями, важно знать о приверженности Java к устранению ошибок.
- *Предложения по улучшению Java* (*Java Enhancement Proposals, JEP*): например, виртуальные потоки JEP 444. Какие-то функции Java выпускаются как превью, пока не будут приняты в финальной версии. Виртуальные потоки вышли в своей первой превью-версии в Java 19 как JEP 425, затем в Java 20 как JEP 436, прежде чем появилась полная версия в Java 21.
- *Запросы на спецификацию Java* (*Java Specification Request, JSR*): более масштабные изменения, сильнее влияющие на экосистему, чем JEP. Например, лямбда-выражения — это JSR 335.

Несмотря на шестимесячный цикл релизов, Java придает большое значение обратной совместимости как самого языка, так и API. Более поздняя версия Java не должна препятствовать компиляции и запуску кода. Хотя обратная совместимость обеспечивается не всегда, Oracle стремится свести изменения к минимуму.

Функции в превью-версиях способствуют достижению этой цели, поскольку они пока не должны обеспечивать обратную совместимость. Вот почему не стоит использовать в продакшене функции превью-версий: они могут каким-либо образом измениться. Обратная совместимость — причина, по которой так мало устаревших API действительно удаляются.

ДОПОЛНИТЕЛЬНЫЕ МАТЕРИАЛЫ

- https://docs.oracle.com/en/java/javase/15/text-blocks/index.html — руководство Oracle по текстовым блокам.
- https://docs.oracle.com/en/java/javase/14/language/records.html — руководство Oracle по записям.
- OCP Certified Professional Java SE 17 Developer Study Guide (Sybex, 2022) — руководство по изучению сертификации от Жанны для более детального освоения функций.
- The Java Module System (Manning, 2019) — книга о модулях с глубокой детализацией.

РЕЗЮМЕ

Ключевые выводы главы:

- Java принадлежит Oracle, но и другие компании также помогают вести этот проект.
- Новые версии выходят каждые шесть месяцев, но LTS-версии — реже.
- Постоянно добавляются новые функции.
- API, которые больше не рекомендованы, обозначаются как устаревшие, а в некоторых случаях — как предназначенные к удалению.
- Обратная совместимость — ключевой принцип развития Java, обеспечивающий бесперебойную работу существующих Java-приложений без необходимости внесения существенных изменений при выходе новой версии Java.