1

How We Got Here: History of Java in a Nutshell

WHAT'S IN THIS CHAPTER?

- Understanding the Stewardship of Java
- Differentiating Key Java Versions
- Working with Deprecation and Retirement
- Identifying Renames
- Understanding the Principles of Change

INTRODUCTION

You've learned Java, and you're ready to start using it at work. Or you are already using it in work, but you've discovered the daunting ecosystem that they never taught you in school. This book is for those who have already learned how to code in Java and are looking for the next step. Whether you are a student, career changer, or professional programmer switching languages, this is your opportunity to learn about the Java ecosystem.

This book is not intended to be a comprehensive guide to the Java ecosystem; that would require many thousand-page tomes. If you are just learning Java, we recommend *Head First Java*, *3rd Edition* (O'Reilly Media, 2022) or *Java For Dummies* (For Dummies, 2025). Then come back to this book. The goal of this book is to expose the reader to some of the most common frameworks, tools, and techniques used in enterprise Java development shops, with enough background and examples to dive right in.

In this chapter, we will cover some information about Java in general before getting into specialized topics in the later chapters. While chapters can be read out of order, we recommend reading Chapters 1–4 before skipping around.

CODE DOWNLOADS FOR THIS CHAPTER

The source code for this chapter is available on the book page at `www.wiley.com`. Click the Downloads link. The code can also be found at `https://github.com/realworldjava/Ch01-History`. See the `README.md` file in that repository for details.

UNDERSTANDING THE STEWARDSHIP OF JAVA

Java was created by James Gosling at Sun Microsystems and released to the world in 1995.

As legend has it, Java started life as "Oak," a new language for building embedded software in smart devices. The idea was novel: instead of writing software for each embedded operating system, design a "write once, run anywhere," compiled, object-oriented language that produces bytecode, and design a bytecode interpreter for each platform that could execute it.

While they were at it, they would include built-in frameworks that were, in other languages, snap-on additions, such as concurrency or UI designers. Then cap it off with memory management in the form of garbage collection and optimize it using just-in-time (JIT) compiling, and *voilà*, they had everything needed for an embedded language platform.

But alas, it was soon discovered that there was already an "Oak" platform brewing. A long night in a coffee shop over some hot Java brew, and the rest is history.

Many years later, in 2009, Oracle acquired Sun, and ownership of Java transferred to Oracle. To this day, Oracle remains the steward, or caretaker, of Java.

The Java community's commitment to ensuring that Java is backward compatible and its robust series of thousands of Technology Compatibility Kit (TCK) tests that must pass for every feature provide the stability and dependability that have thrust Java into its place as one of the most successful programming languages in history.

Oracle recognizes its stewardship as a critical responsibility, given the sheer number of people and companies heavily using Java. The JCP (Java Community Process) has an Executive Committee (basically Java's board of directors) with many organizations as members to ensure the community is represented. Some past/future members are competitors with their own JDKs (Java Development Kits), for example, Amazon, Azul, and Microsoft. Others are complementary, notably Integrated Development Environment (IDE) providers like Eclipse and JetBrains. Others are simply strong proponents of Java like BellSoft and Fujitsu. Finally, there are community members like SouJava (Brazilian Java User Group) and Ken Fogel (individual member).

One of the changes introduced under Oracle's stewardship was a more frequent and predictable release schedule. Notice in Figure 1.1 that releases have varied in both frequency and month of release. In one case, Java users had to wait more than five years for a new version.

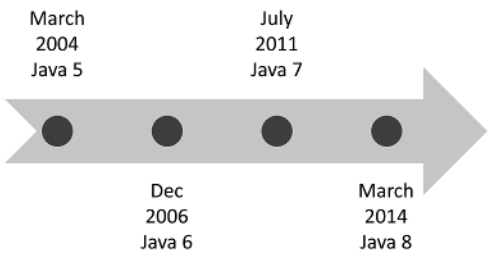

FIGURE 1.1: Java releases

With the new release cadence, a new version of Java comes out every six months. You may be thinking that many companies don't want to upgrade every six months, and you'd be right! There are also less frequent releases called *Long-Term Support* (*LTS*) releases. When the new release cycle started, LTS releases were every three years. They are now every two years.

Table 1.1 shows releases from the start of the new release cadence to the publishing of this book. As you can see, there are now regular, predictable releases. This book uses Java 21, which was the latest LTS at the time of printing.

Figure 1.2 shows the LTS release schedule. You can predict LTS dates into the future now that there is a pattern. This helps companies plan.

Many companies run their Java programs remotely, such as through a website. Until 2020, this was often done through *Spring* or *Java EE* (Enterprise Edition.) See Chapter 6, "Getting to Know the Spring Framework," and Chapter 14, "Getting to Know More of the Ecosystem," for more details. In 2020, Oracle handed over stewardship of Java EE to an open-source foundation. Since "Java" is a trademark, the Java EE framework was renamed to "Jakarta EE" at that point.

TABLE 1.1: Java Releases

YEAR	MARCH RELEASE	SEPTEMBER RELEASE
2017	n/a (cadence started in September)	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

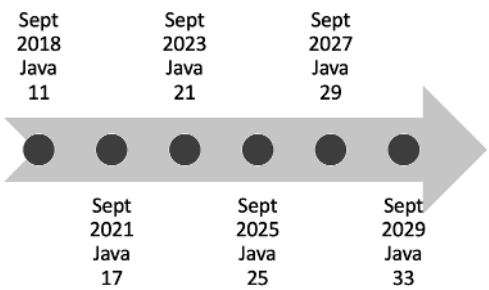

FIGURE 1.2: LTS release schedule, with future predictions

DIFFERENTIATING KEY JAVA VERSIONS

The Java language has evolved quite a bit over time. By looking at key features added over time, you can get a feel for this evolution along with an expectation that the language will continue to improve. Additionally, you will know what to expect if your work project is not on the latest version.

Note that this section is not even close to a full list of features. We picked the most significant feature of each version covered.

You might be wondering why we aren't including type inference with the `var` keyword in this section. While both of this book's authors use `var` in some cases in our work code, we felt it would be clearer in this book to specify what the types are and ensure all imports are represented.

Coding Generics in Java 5

Before Java 5, the compiler had no way of knowing what you intended to put in a list. Java 5 introduced *generics*, which allow you to specify the type, in the case of the following example, a `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: }
```

Now that we've informed the compiler about our intent, it is our assistant. The compiler will produce an error if we attempt to add an `Integer` or `LocalDate` or anything else to `buildingNames`.

You might be wondering why the line 7 doesn't say `new ArrayList<>()` and instead uses a longer version. The diamond operator (`<>`) was added in Java 7, and this example is for Java 5. Even this shows the evolution of the language, though!

Coding with Functional Programming from Java 8

Functional programming is a way of writing code more declaratively. You specify what you want to do rather than dealing with the state of objects. You focus more on expressions than loops.

Java 8 introduced lambdas and method references to facilitate writing code with *deferred execution*, also known as "code that runs later." Both let you easily pass executable code to a method. In the following code, you can see two lambdas and a method reference:

```
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: }
```

This code outputs these two lines:

```
*Lenovo
*Microsoft
```

Line 10 kicks off the stream pipeline. Lines 11 and 12 use lambdas to specify what should happen at each step. The `->` gives you a clue that you are looking at a lambda. Line 11 only allows names with at least five characters through the pipeline. Line 12 adds an asterisk to the beginning of each line in the pipeline. Notice it does not change the original `computerCompanies` contents.

Then on line 14, you can see the first method reference. The `::` tells you it is a method reference and in this case prints the values. A method reference is a more compact version of a lambda. Writing `n -> System.out.println(n)` would have been equivalent.

You might have noticed that we didn't call `List.of()` instead of `Arrays.asList()` on line 8. `List.of()` was added in Java 9!

Coding Modules from Java 11

The *Java Platform Module System* (JPMS) groups code at a higher level than packages. The main purpose of a module is to provide groups of related packages to offer a particular set of functionality to developers. It's like a JAR file except a developer chooses which packages are accessible outside the module. Many companies choose not to use modules and indeed their use is optional.

Java modules were introduced in Java 9, not Java 11. We are listing them under Java 11 because that was the first LTS release of Java that supported modules. Any projects on a version of Java lower than 8 will upgrade to Java 11, 17, 21, etc. They will not go to 9 since support for Java 9 ended in March 2018 (six months after its release in September 2017).

A module is a Java Archive (JAR) consisting of a `module-info.java` file at the root of the project and one or more Java packages. For example, here is a module with two packages containing one class each:

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

This example shows `.class` files instead of `.java` files because the JAR file contains executable bytecode. But wait. You don't want the callers of your module to reference `PrivateHelper`. That's where the `module-info` file comes into play:

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

The `module` keyword lets Java know we are specifying a module here rather than a `class` or `interface` or other Java top-level type. The `exports` directive tells Java that callers can reference the `utils` package directly from their code. Since the `internal` package is not mentioned in the module-info file, callers cannot use it directly, only through `BookUtils`. Finally, the `requires` directive specifies that we use the `java.sql` module, which is provided by Oracle with the JDK. There is another module we use called `java.base`, but that one is provided automatically without having to specify it. It contains common classes such as `String` and `List`.

Coding Text Blocks and Records from Java 17

It was hard to pick one favorite from Java 17, so we picked two. Text blocks are also known as *multiline strings*. Text blocks were added in Java 15. Like modules, we are listing them under the first LTS release that made them available. You can see a text block here surrounded by three double quotes (`"`):

```
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: }
```

This outputs these four lines:

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

Notice that the indentation of our text block is preserved and the incidental whitespace used for IDE formatting is not. Also note that the triple quotes in line 9 will introduce a blank line in the string. If you want to omit the trailing line break, place the three closing double quotes (`"`) at the end of line 8.

Records were introduced in Java 17 and got rid of a ton of boilerplate code when creating an immutable class. A simple record is used in this example:

```
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: }
```

This outputs the following:

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

Line 3 creates a record with two fields. A record is like an immutable class that includes the following:

- An instance variable for each parameter listed
- Getters (without the `get`/`is` prefix)
- A constructor taking a parameter for each instance variable
- `equals()` and `hashCode()` methods that include the value of each instance variable
- A `toString()` method that prints each field of the record in a convenient, easy-to-read format

Line 8 demonstrates that the getter method is `x()` rather than `getX()`. Finally, line 9 implicitly calls `toString()` showing you can get a useful implementation without writing any code.

Records are convenient when you need a simple immutable class. You can override methods in the record body or add your own methods. When you need more customization power or setters such as for Java Persistence API (JPA) entities, Project Lombok may better match your needs. See Chapter 8, "Annotation-Driven Code Using Project Lombok," for details.

Learning About Virtual Threads from Java 21

A long time ago Java used the raw `Thread` class for concurrency. Although this remains the low-level unit of concurrency, the language then added the `Executors` class and more powerful concurrency techniques such as fork join and `parallelStream()`. However, parallelization is not free; there are setup and coordination costs in addition to the memory resources of the threads themselves.

As computers get more central processing units (CPUs), parallelization becomes more important. Java 21 introduced virtual threads to greatly reduce the cost of concurrency. See Chapter 9, "Parallelizing Your Application Using Java Concurrency," for a more detailed explanation and example of virtual threads.

WORKING WITH DEPRECATION AND RETIREMENT

As more and more gets added to the Java language, you might be wondering how things get removed. First, the developers identify classes or methods as *deprecated*, which means further usage is discouraged but still allowed. Suppose you want users to stop passing a parameter to `magicNumber()`.

```
1:  public class DeprecationExample {
2:     /**
3:      * Returns a number
4:      * @deprecated Use {@link #getNumber()} instead
5:      * @param num number
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: }
```

Line 4 uses a Javadoc tag, `@deprecated`, to include in the documentation that using this method is discouraged. The `@link` tag is used to explain that the `magicNumber()` call without a parameter is preferred instead.

Line 8 shows the annotation marking the method as `@Deprecated`. IDEs show the use of deprecated code as a warning to call your attention to it. See Chapter 2, "Getting to Know your IDE: The Secret to Success," for more information on IDEs.

Line 8 also shows the `forRemoval` attribute, which was added in Java 9. This is used to signify whether the intent is to remove the deprecated code in the future or merely encourage alternative application programming interfaces (APIs). This allows developers to make intelligent decisions on whether to migrate code.

Oracle keeps a list of APIs that were removed at `https://docs.oracle.com/en/java/javase/21/migrate/removed-apis.html`. This page shows what was removed in each version between Java 9 and Java 21. For example, in Java 21, this was one of two APIs that were removed:

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

You are unlikely to have used this method. Oracle values backward compatibility and does not remove code in broad use. In fact, some of the methods in `java.util.Date` have been deprecated since Java 1.1!

IDENTIFYING RENAMES

Given the evolution of the Java ecosystem over the years, there are some renames you should be familiar with. This section helps you identify whether old guides/documentation are equivalent or obsolete.

Changing to Jakarta EE

As mentioned, in 2020, Oracle announced the migration of Java EE to Jakarta EE. Since that wasn't so long ago, documentation mentioning Java EE is still relatively useful. After all, a Java EE tutorial on *servlets*, commonly used to serve web pages, is mostly the same as Jakarta EE servlets, and the concepts described in good tutorials will all apply.

The key difference is that Jakarta EE 9 renamed the package name prefix from `javax` to `jakarta` since Java is trademarked by Oracle. Additionally, new features are covered only in the Jakarta EE documentation.

Renaming Certifications

Oracle had a number of certifications including Sun Certified Java Associate (SCJA), Sun Certified Java Professional (SCJP), and Sun Certified Master Java Developer (SCMJD) that you could earn. All of these certifications were renamed to begin with "O," for Oracle, giving us certifications like OCJA, OCJP, and OCMJD. When looking at résumés, remember that these certifications are all equivalent.

Many of these certifications don't exist anymore or have changed significantly since the renaming, so learning from websites with the old Sun cert names is unlikely to be useful.

UNDERSTANDING THE PRINCIPLES OF CHANGE

Change is constant in Java and keeps the language current and useful. Some changes are inspired by other languages. You should expect that the language will continue to evolve as more versions are released. These changes will include enhancements to the language and new APIs.

Changes to the Java ecosystem come from three main places.

- **Bug fixes:** Java provides fixes to bugs that are reported in the bug database. You are unlikely to encounter these changes, but it is important to be aware that Java is committed to fixing reported bugs.

- **Java Enhancement Proposals (JEPs):** For example, virtual threads are JEP 444. Some Java features are released as previews before they get committed to as final. Virtual threads had its first preview in Java 19 as JEP 425 and second preview in Java 20 as JEP 436 before being fully released in Java 21.
- **Java Specification Requests (JSRs):** JSRs are bigger and more impactful than JEPs. For example, lambdas were JSR 335.

While committed to a six-month release cycle, Java heavily values backward combability of both the language itself and APIs. Using a later version of Java shouldn't prevent your code from compiling or running. Backward compatibility isn't always met, but Oracle tries to minimize the change as much as possible.

Preview features assist in this goal since they aren't locked into backward compatibility yet. This is why you shouldn't use preview features in production; they could change in any way. Backward compatibility is why so few deprecated APIs are actually removed.

FURTHER REFERENCES

- `https://docs.oracle.com/en/java/javase/15/text-blocks/index.html`

  Oracle's guide to text blocks
- `https://docs.oracle.com/en/java/javase/14/language/records.html`

  Oracle's guide to records
- *OCP Certified Professional Java SE 17 Developer Study Guide* (Sybex, 2022)

  Jeanne's certification study guide for more details on all features
- *The Java Module System* (Manning, 2019)

  Book on modules with great detail

SUMMARY

The following are the key takeaways from the chapter:

- Java is owned by Oracle, with others helping guide the process.
- New versions of Java are released every six months, with less frequent LTS versions.
- New features are continually being added.
- APIs that are no longer encouraged for use are deprecated and, optionally, tagged for removal.
- Backward compatibility is a central tenet of Java's evolution and ensures that existing Java applications will run smoothly without requiring extensive modifications when a new version of Java is released.