What a Year! Java 10 and 10 Big Java Milestones

Java has made tremendous strides in the past 12 months, with exciting **new features and capabilities for developers** of all kinds.

Table of Contents

Introduction: Twelve Months of Java Excitement

Since its first release in 1995, the Java language and Java platform have grown and evolved—and today, Java remains the cornerstone platform for desktop, server, embedded systems, and now cloud development.

Never content to rest on their well-earned laurels, Oracle and the Java community are taking the Java platform to new heights—and that includes both the open source OpenJDK and the commercially licensed Oracle Java SE products.

Take a tour of 10 new changes released with Java 9 and Java 10 over the past 12 months—a period that saw two releases of the Java platform.

01: JAVA 10 DEBUTS A NEW SIX-MONTH FEATURE RELEASE MODEL

Twelve busy months. Two releases, Java 9 and Java 10. What's going on?

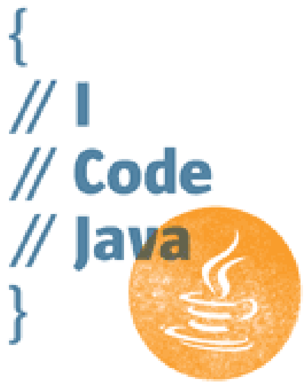
Easy: Starting with Java 10, released in March 2018, Java has moved to a faster release cadence, with relatively smaller feature releases every six months. This is a significant change from Java's previous model of one large release every two to three years.

This accelerated schedule allows for rapid iterations of the Java platform, and the ability, for Java developers, to leverage new APIs and features faster than ever.

Before JDK 10, Java's improvements on APIs, the language, and JVM were only delivered when big features were ready—that is, every three years, more or less. That's too slow to keep up with today's ever-accelerating pace of innovation.

For Java to remain competitive, the Java community determined that the platform must not just continue to move forward—it must move forward faster than ever before, with a smoother delivery of new features.

Shipping a Java release every six months is rapid enough to ensure that small features can be delivered quickly, while allowing enough time to properly integrate larger features when they are ready. As you can see with the features added with Java 10, just six months after the release of Java 9, this new cadence ensures that Oracle can deliver value on each release and maintain the high quality expected by the Java community.



"Java remains the cornerstone platform for desktop, server, embedded systems, and now cloud development."

This accelerated schedule allows for rapid iterations of the Java platform, and the ability, for Java developers, to leverage new APIs and features faster than ever.

02: APPLICATION CLASS-DATA SHARING SPEEDS STARTUP, SAVES MEMORY

The Application Class-Data Sharing functionality, now available to everyone in Java 10, allows application classes to be placed in a shared archive. The benefit can be significantly faster startup times, as well as reduced memory footprint.

In previous releases of Java, each process had its own copy of all application classes. The JVM loaded a fresh copy of those classes each time it created a new process. With cloud applications having potentially tens of thousands of classes—and continually creating and killing processes—this is not only wasteful of memory and resources, but also time-consuming.

With Application Class-Data Sharing, the classes are loaded into a shared memory space as needed. When a process is created and needs an already loaded class, the JVM points to shared-class metadata instead of loading new instances of the classes for each process.

Code Sample #1: Application Class-Data Sharing

Using the AppCDS archive

Once the AppCDS archive is created, you can use it when starting the application. Do this by specifying the -Xshare:on -XX:+UseAppCDS command-line options, with the -XX:SharedArchiveFile option to indicate the name of the archive file. For example:

```
$ java -Xshare:on -XX:+UseAppCDS -XX:SharedArchiveFile=hello.jsa \
    -cp hello.jar HelloWorld
```

"With jlink, there is now a single tool that can analyze the entire application, and perform optimizations to ensure that the distribution only contains what is required."

This functionality extends the previous, more limited Class-Data Sharing functionality, which allowed the application's bootstrap class loader to share class data prior to beginning execution. Beginning with Java 10, the JVM can also load and share application class data while the application is running. This new feature is also ideally suited to microservices and serverless architecture.

03: LINK-TIME UTILITY OPTIMIZES APPLICATION BYTECODE FOR DEPLOYMENT

An application is designed. Written. Tested. Ready to deploy to the targeted server or other device containing a JVM. However, sometimes there are opportunities to optimize the size of the application beyond what's normally done by the Java compiler and JVM runtime environment. That's where jlink comes in.

jlink, a new utility added with Java 9, introduces further improvements at link time, which is an optional phase between compile time (when the developer's source code is translated into JVM-readable bytecode) and runtime (when the bytecode is loaded into and executed on the target device's JVM). Link time provides a unique opportunity to perform what's known as whole-world optimization on the complete application.

Large programs can now be written and compiled in discrete, manageable pieces, called modules. A finished program might include many modules created by the development team, as well as modules reused from other projects or from open source communities. However, not every application will need the same set of modules. In the past, it was time-consuming to minimize the runtime needed for an application, and the level of granularity available was not as good as what can now be achieved.

With jlink, there is now a single tool that can analyze the entire application, and perform optimizations to ensure that the distribution only contains what is required—the necessary modules and their dependencies.

The output from running jlink is a custom runtime image, which is reduced in size, compared to the complete JDK image, and easier to distribute and deploy. Not only that, but since the jlink-optimized custom runtime image contains only what's necessary for the application, the image can be loaded and executed more efficiently by the JVM.

04: EXPERIMENTAL JAVA-BASED JIT COMPILER

A just-in-time (JIT) compiler translates programming code into fast, efficient machine-optimized computer code only a few moments before it's needed by the application. Thanks to the rapid iterations of software driven by the Java release cadence, developers can get their hands on Graal, an experimental JIT now included with OpenJDK in Java 10.

Graal is the basis of the experimental Java ahead-of-time (AOT) compiler, which was first introduced in JDK 9. Enabling it to be used as an experimental JIT compiler in Java 10 is one of the initiatives of Project Metropolis, and is the next step in investigating the feasibility of a Java-based JIT for the JDK. Graal has the potential to improve application performance by aggressively optimizing code for specific microprocessor architectures.

The Graal JIT is strictly experimental at this stage, and can only compile code for Linux running on the x86 platform. The goal is for developers to get hands-on experience using Graal as a JIT, and for contributors to the Java platform to run a battery of standardized tests on Graal in their own software development and deployment environments.

The benefits: Java developers gain access to technologies quickly, thanks to the faster release cadence—and work can proceed more quickly on the Graal JIT.

"Thanks to the rapid iterations of software driven by the Java release cadence, developers can get their hands on Graal, an experimental JIT now included with OpenJDK in Java 10."

New for Java 10, the full GC mode functionality of the G1 garbage collector has been rewritten as faster parallel code.

05: THE PARALLEL FULL GARBAGE COLLECTOR DUMPS THE TRASH EFFICIENTLY

In Java, garbage collection (GC) is the process of looking at memory, identifying which objects are in use and which are not, and deleting the unused objects. GC frees up memory for use by other objects in the program—and without GC, many programs would eventually run out of memory. The JVM runs GC automatically based on various criteria, such as a large number of unreferenced, no-longer-used objects that are still consuming memory.

The highly efficient garbage collector in Java is named G1. Most of the time, the JVM runs G1 in conjunction with application threads so that programs keep running without interruption. Sometimes, however, the JVM needs to run a "full garbage collection" when there are excessive unreferenced objects in memory. In those cases, the JVM pauses application execution for a few moments and runs G1 in "full GC mode," dedicating all available processor resources to run GC as fast as possible.

New for Java 10, the full GC mode functionality of the G1 garbage collector has been rewritten as faster parallel code. Previously, full GC functionality ran in only a single thread. The benefit: The full GC executes faster, thereby pausing application execution for a shorter period of time.

"With cloud-based servers that can run hundreds or thousands of Java applications, the Java community saw the need to break Java SE and Java applications into modules that contained essential functionality."

06: JAVA PLATFORM MODULE SYSTEM SHRINKS JAVA DEPLOYMENTS

The Java Platform Module System (JPMS), introduced with Java 9 in September 2017, optimizes the process of configuring and deploying large Java programs.

Previously, with only a few exceptions, it was difficult to break the Java platform, and large programs, into smaller pieces that can be used on small devices—or to eliminate parts of the Java platform that aren't needed to save memory or storage space. This often resulted in wasted resources.

With cloud-based servers that can run hundreds or thousands of Java applications, and with Java applications themselves becoming larger and more complex, the Java community saw the need to break Java SE and Java applications into modules that contained essential functionality.

The JPMS, a central component of Project Jigsaw, allows the Java SE platform to be decomposed into a set of components, which can be assembled by developers into custom configurations that contain only the functionality actually required by the developer's application. These custom configurations can be optimized to run faster in some cases, and also be more secure by removing unneeded functions and APIs.

07: CONVENIENCE FACTORY METHODS MAKE SMALL COLLECTIONS A SNAP

In Java, a collection is a set of objects. The collection might be as simple as a list of numbers, or may be as complex as a queue, tree, or hash. Developers frequently use collections throughout applications—perhaps hundreds of collections may be used.

Sometimes those collections are large or complicated, and the objects within the collection are changed while the program is running. Often, however, developers require collection that is small, simple, and unchanging—like a list of the numbers from 1 to 10.

Traditionally, Java requires several lines of code to create such small, unmodifiable collections, with separate lines of code to declare and initialize an empty collection, then one line of code to add each individual element, and a final instruction to declare the resulting collection immutable.

The convenience factory methods for collections, introduced in Java 9, are used for creating collections with small numbers of elements. The code can be written in a single expression. The benefit: Collections are easier to create and the code is easier to understand.

"The convenience factory methods for collections, introduced in Java 9, are used for creating collections with small numbers of elements."

08: OPTIONAL CLASS ENHANCEMENTS HANDLES NULL

The Optional class, introduced with Java 8, is used to extend objects to indicate that the value of the object may be `null`—literally no value, which is not the same as zero or an empty string. Normally, a program throws an error when it unexpectedly tries to manipulate an object with a `null` value. However, in some situations, `null` may be a valid state for an object, and thus the need to create the Optional class.

The Java community decided to extend the Optional class with enhancements, which appeared in Java 9. One enhancement is the Stream method, which facilitated converting a stream of Optional objects into a stream of values. Without the Stream method, the programmer would have to write code to filter out `null` values before converting the objects into values, or otherwise process those objects.

Code Sample #2: Optional Class Enhancement

Stream of Optional

```
// Convert List<CustomerID> to List<Customer>, ignoring unknowns

// Java 8
   List<Customer> list = custIDlist.stream()
      .map(Customer::findByID)
      .filter(Optional::isPresent)[1]
      .map(Optional::get)[2]
      .collect(Collectors.toList());

// Java 9 adds Optional.stream(), allowing filter/map to be fused into a flatMap:
   List<Customer> list = custIDlist.stream()
      .map(Customer::findByID)
      .flatMap(Optional::stream)[3]
      .collect(Collectors.toList());
```

Assume findByID() returns Optional<Customer>

1 *Let only present Optionals through*

2 *Extract values from them*

3 *Optional.stream() allows filter() and map() to be fused into flatMap()*

The value of the Optional class enhancement is it's easier to work with Optional objects, which not only simplifies the developer's work, but also makes it easier to create robust code that will hand the presence of `null` values, a common source of runtime errors.

09: STREAM API ENHANCEMENTS HELP SOLVE CODING CHALLENGES

Streams are a sequence of elements that can be worked on either sequentially or in parallel. To process a stream, the group of source elements is placed in a pipeline; operations are applied to the pipeline; and then a terminal operation determines what to do with the results. If only some elements in the stream need to be processed, the programmer specifies a filter, which is applied to the pipeline.

Streams provide an efficient way to process data, in part because streams are lazy and don't waste processor resources: Computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed. Streams are becoming increasingly important, especially for developers using functional methodologies, or who are moving to cloud-based serverless computing.

Java 8 saw the first implementation of streams in the Java platform. The Java community enhanced the Streams API in Java 9 to add more methods. The `iterate` method, for example, offers a simple way to create loops that are bounded by a maximum value or condition, reducing the possibility of infinite loops.

"JShell, introduced with Java 9, is a tool designed specifically to help developers with that trial-and-error process."

Code Sample #3: Stream API Enhancements

Here's a simple example that takes the new static "`iterate`", and the new "`takeWhile`":

```
IntStream
      .iterate(1, n -> n + 1)
      .takeWhile(n -> n < 10)
      .forEach(System.out::println);
```

Overall, the Stream API enhancements in Java 9 offer important improvements to developers using streams, making code easier to write and understand.

10: JSHELL GIVES DEVELOPERS IMMEDIATE FEEDBACK WHEN LEARNING JAVA

The Java libraries contain thousands of packages, classes, and interfaces with their respective methods and fields. It can be a challenge for developers to figure out exactly which library can help solve a particular programming problem most elegantly—or most effectively. Another challenge: Learning how to program these libraries often takes some trial and error before one masters it.

JShell, introduced with Java 9, is a tool designed specifically to help developers with that trial-and-error process, which is set up as what's called a Read-Eval-Print Loop (REPL). In other words, load the data, test the code, and see the results immediately.

JShell is a command-line tool that facilitates the developer to interactively evaluate declarations, statements, and expressions of the Java programming language. Most importantly: Statements and expressions need not occur within a method, and variables and method need not occur within a class. That eliminates a lot of the overhead needed to set up the tests so that developers can focus on the REPL scenario itself.

The benefit: Developers can experiment with the Java language and libraries quickly, interactively test how a class or method will work in their specific application context, get immediate feedback, and then quickly leverage what they have learned in their programs.

"Java is the #1 platform for development in the cloud, and with more than 12 million developers worldwide running Java, the platform has never been so popular."

THE #1 PLATFORM FOR DEVELOPMENT IN THE CLOUD

Java is the #1 platform for development in the cloud, and with more than 12 million developers worldwide running Java, the platform has never been so popular. The past 12 months have seen two new releases of Java—and with releases on a six-month schedule, the best will only get better.

Visit **website** to see how Java 10 powers our digital world—and can power yours, too.

ORACLE CORPORATION

Worldwide Headquarters
500 Oracle Parkway, Redwood Shores, CA 94065, USA

Worldwide Inquiries
TELE + 1.650.506.7000 + 1.800.ORACLE1
FAX + 1.650.506.7200
oracle.com

CONNECT WITH US

facebook.com/oracle youtube.com/oracle linkedin.com/company/oracle twitter.com/oracle

Integrated Cloud Applications & Platform Services

Java™

ORACLE®