eu

JAVA 13 FEATURES (WITH EXAMPLES)

Sven Woltmann

Last update: November 27, 2024

Java 13 was released on September 17, 2019.

The changes in Java 13 are pretty modest. In total, only five JDK Enhancement Proposals (JEPs) have made it into the release – and three of them are classified as experimental or preview features.

The article starts with the experimental and preview features, as these are the most exciting changes in Java 13.

Next are performance improvements, enhancements to the JDK class library, and other changes that we rarely encounter in our daily development work.

Contents [hide]

Experimental and Preview Features

I will not go into the experimental and preview features in full depth here. A detailed description will follow in those parts of the series where these features will reach production maturity.

Switch Expressions (Second Preview)

JEP 325 introduced switch expressions as a preview in Java 12. *switch* can be used as a statement or as an expression since then. "Expression" means that *switch* returns a value, such as in the following example from the JEP:

```
int numLetters = switch (day) {
  case MONDAY, FRIDAY, SUNDAY:
    break 6;
  case TUESDAY:
    break 7;
  case THURSDAY, SATURDAY:
    break 8;
  case WEDNESDAY:
    break 9;
};
```

Based on feedback from the developer community, JDK Enhancement Proposal 354 replaces the *break* keyword in switch expressions with the new *yield* keyword:

```
int numLetters = switch (day) {
  case MONDAY, FRIDAY, SUNDAY:
    yield 6;
  case TUESDAY:
    yield 7;
  case THURSDAY, SATURDAY:
    yield 8;
  case WEDNESDAY:
    yield 9;
};
```

The new keyword is only recognized in the scope of a Switch Expression. So if you use *yield* elsewhere in your source code, there is most likely no need to adjust your source code.

Switch Expressions will reach production status in the next release, Java 14. You can find all details about them in the main article on Switch Expressions.

Text Blocks (Preview)

To define a multiline string, we used to use escape sequences for line breaks and double quotes contained in the string. An SQL statement looked like this, for example:

```
String sql =
    "SELECT id, firstName, lastName FROM Employee\n"
        + "WHERE departmentId = \"IT\"\n"
        + "ORDER BY lastName, firstName";
```

JDK Enhancement Proposal 355 allows us to write such a string in a much more readable way:

```
String sql = """
    SELECT id, firstName, lastName FROM Employee
    WHERE departmentId = "IT"
    ORDER BY lastName, firstName""";
```

Text blocks will reach production readiness in Java 15. You can find an introduction in all details in the main article on text blocks.

To use text blocks in Java 13, you must either enable them in your IDE (in IntelliJ via *File→Project Structure→Project Settings→Project→Project language level*) or use the *--enable-preview* parameter when running the *javac* and *java* commands.

Text blocks replaces the withdrawn JEP 326, "Raw String Literals", which was not accepted by the community. If you are interested in the reasons, you can find them on the jdk-dev mailing list.

ZGC: Uncommit Unused Memory (Experimental)

ZGC is an experimental garbage collector introduced in Java 11 that promises extremely short stop-the-world pauses of 10 ms or less.

JDK Enhancement Proposal 351 extends ZGC to return unused heap memory to the operating system after a specific time.

Using *-XX:ZUncommitDelay*, you can specify the time in seconds, after which ZGC returns unused memory. By default, this value is 300 seconds.

The feature is enabled by default and can be disabled with *-XX:-ZUncommit*.

ZGC will reach production status in Java 15. In the corresponding article, I will introduce the new garbage collector in more detail.

Performance Improvements

Dynamic CDS Archives

Java 10 introduced Application Class Data Sharing – a feature that allows creating a so-called shared archive file. This file contains the application classes in a binary form as required by the JVM of the platform used. The file is mapped into the JVM's memory via memory-mapped I/O.

Until now, it was relatively complex to create this file. First, we had to dump a class list during a test run of the application. Only in a second step could we generate the shared archive from this list.

The following sample *java* calls taken from the article linked above:

```
java -Xshare:off -XX:+UseAppCDS
    -XX:DumpLoadedClassList=helloworld.lst
    -cp target/helloworld.jar eu.happycoders.appcds.Main

java -Xshare:dump -XX:+UseAppCDS
    -XX:SharedClassListFile=helloworld.lst
    -XX:SharedArchiveFile=helloworld.jsa
    -cp target/helloworld.jar
```

JDK Enhancement Proposal 350 simplifies this process. As of Java 13, you can specify the *-XX:ArchiveClassesAtExit* parameter to generate the shared archive at the end of the application execution. The additional parameters *-Xshare:on* and *-XX:+UseAppCDS* are no longer required:

```
java -XX:ArchiveClassesAtExit=helloworld.jsa
    -cp target/helloworld.jar eu.happycoders.appcds.Main
```

The created shared archive is much smaller than before (256 KB instead of 9 MB). It now only contains the classes of the application. The JDK classes are loaded from the base archive *classes.jsa* delivered with the JDK.

The shared archive is used as follows as of Java 13:

```
java -XX:SharedArchiveFile=helloworld.jsa
    -cp target/helloworld.jar eu.happycoders.appcds.Main
```

In the article linked at the beginning of this section, you can find an example of using AppCDS with step-by-step instructions. Try to reproduce the example and use the new *-XX:ArchiveClassesAtExit* option instead of the previous two steps.

Soft Max Heap Size

You can use the new command line parameter *-XX:SoftMaxHeapSize* to set a "soft" upper limit for the heap size. The garbage collector will then try to keep the heap below this limit and only exceed it if necessary to avoid an *OutOfMemoryError*.

The application scenario lies in environments in which you pay for the actual RAM usage. Thus, the heap can generally be kept small but may temporarily grow beyond the soft upper limit when memory requirements increase.

Currently, only the (experimental) ZGC supports this feature.

(There is no JDK enhancement proposal for this feature.)

JDK Class Library Enhancements

Several methods have been added to the *ByteBuffer* class, allowing read/write operations to be performed at specified buffer positions instead of the position managed by the *ByteBuffer*, as was previously the case.

If you need a refresher on ByteBuffer, I recommend this ByteBuffer basics article.

ByteBuffer.slice()

Using *ByteBuffer.slice()* you can create a view on a section of the buffer. This method, which already existed before Java 13, returns a view that starts at the buffer's current position and whose capacity and limit correspond to the remaining bytes in the buffer.

New is the method *ByteBuffer.slice(int index, int length)*. It allows you to create a view that starts at position *index* and contains *length* bytes. The new method thus ignores the position, capacity, and limit of the underlying buffer.

New *ByteBuffer.get()* and *put()* Methods

Analogously, there are two new *get()*, and two new *put()* methods, which do not read/write the data at the *current* position of the buffer – but at an explicitly specified position:

- *get(int index, byte[] dst, int offset, int length)* – transfers *length* bytes from the buffer position specified by *index* into the byte array *dst* starting at position *offset*.

- *get(int index, byte[] dst)* – transfers data from the buffer position specified by *index* into the byte array *dst*. The number of bytes transferred is equal to the length of the destination array.
- *put(int index, byte[] src, int offset, int length)* – transfers *length* bytes from the byte array *src* starting at position *offset* into the buffer starting at position *index*.
- *put(int index, byte[] src)* – transfers all bytes from the byte array *src* into the buffer starting at position *index*.

The buffer's position remains unchanged for all four methods.

FileSystems.newFileSystem()

Using the FileSystems.newFileSystem(Path path, ClassLoader loader) method, you can create a pseudo-file system with contents mapped to a file (such as a ZIP or JAR file).

The method was overloaded in Java 13 with a variant, which makes it possible to pass a provider-specific file system configuration: *FileSystems.newFileSystem(Path path, Map env, ClassLoader loader)*

Furthermore, two variants have been added, each without the *loader* parameter. A class loader is only required if the so-called *FileSystemProvider* for the file type to be mapped is not registered in the JDK but must be loaded via the specified class loader. For standard file types like ZIP or JAR, this is not required.

Other Changes in Java 13 (Which You Don’t Necessarily Need to Know About as a Java Developer)

This section lists changes that rather few Java developers will come into contact with.

Reimplement the Legacy Socket API

The *java.net.Socket* and *java.net.ServerSocket* APIs have existed since Java 1.0, and the underlying code (a mix of Java and C code) is difficult to maintain and extend, especially in light of Project Loom, which aims to introduce Virtual Threads (lightweight threads managed by the JVM).

JDK Enhancement Proposal 353 replaces the old implementation with a more modern, better maintainable, and extensible implementation, which in particular should be adaptable to Project Loom without further refactorings.

Unicode 12.1

As in the previous two Java releases, Unicode support has been increased in Java 13 – to version 12.1. That means that classes such as *String* and *Character* must handle the new characters, code blocks, and type systems.

For an example, see the article about Unicode 10 in Java 11.

(No JDK enhancement proposal exists for Unicode 12.1 support.)

Complete List of All Changes in Java 13

This article has presented all the features of Java 13 that are defined in JDK Enhancement Proposals – and enhancements to the JDK class library that are not associated with any JEP.

For a complete list of changes, see the official Java 13 Release Notes.

Summary

Java 13 is a very modest release.

In the second preview of "Switch Expressions", *break* was replaced by *yield*. Multiline strings finally made their way into the language with the "Text Blocks" preview.

The experimental ZGC can return unused memory to the operating system and may be configured with a "soft" maximum heap size.

"Dynamic CDS Archives" makes employing Application Class Data Sharing a piece of cake from Java 13 onwards.

ByteBuffer has been extended with methods to read and write at absolute positions, and there are some new variants of the *FileSystems.newFileSystem()* method.

The Java 1.0 Socket API has been completely rewritten to be fit for the lightweight threads developed in Project Loom.

If you liked the article, I would be happy about a comment, or if you shared the article via one of the share buttons at the end.

If you want to be informed when the next part of the series goes online, click here to sign up for the HappyCoders newsletter.

← Previous:
New features in Java 12

Next: →
New features in Java 14

Java Versions PDF Cheat Sheet (updated to Java 23)

Stay up-to-date with the latest Java features with this PDF Cheat Sheet

- ✓ Avoid lengthy research with this **concise overview of all Java versions up to Java 23**.
- ✓ Discover the **innovative features** of each new Java version, summarized on a single page.
- ✓ **Impress your team** with your up-to-date knowledge of the latest Java version.

First name...

Email address...

Send Me the PDF Now!

You'll receive this PDF by subscribing to my newsletter. You can unsubscribe at any time. Read my privacy policy.

About the Author

I'm a freelance software developer with more than two decades of experience in scalable Java enterprise applications. My focus is on optimizing complex algorithms and on advanced topics such as concurrency, the Java memory model, and garbage collection. Here on HappyCoders.eu, I want to help you become a better Java programmer. Read more about me here.

Leave a Reply

Your email address will not be published. Required fields are marked *

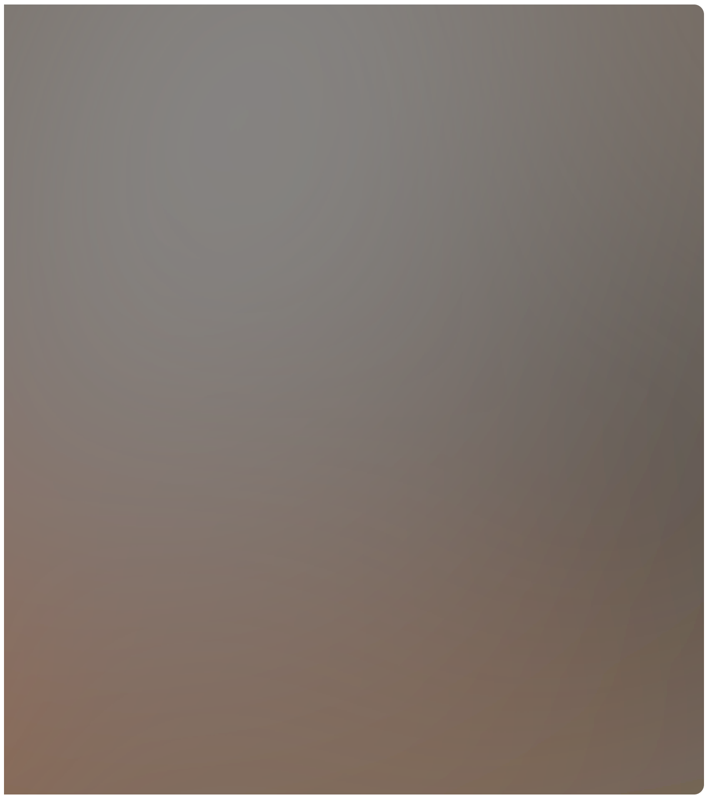
Comment *

Name *

Email *

Post Comment

You might also like the following articles



Sven Woltmann

December 4, 2024

AHEAD-OF-TIME CLASS LOADING & LINKING – TURBO FOR JAVA APPLICATIONS

Sven Woltmann December 3, 2024

PRIMITIVE TYPES IN PATTERNS, INSTANCEOF, AND SWITCH

Sven Woltmann December 3, 2024

Sven Woltmann December 2, 2024

Advanced Java topics, algorithms and data structures.

JOIN OUR FREE NEWSLETTER

Boost your skills: Become a better Java programmer.

CLICK HERE TO SUBSCRIBE!