

# The Latest Features in Java 24 (Part 1)

You can find more details and the latest version of this PDF at [www.happycoders.eu/java-versions](http://www.happycoders.eu/java-versions).

## Stream Gatherers – JEP 485

With *filter*, *map*, *flatMap*, *mapMulti*, *distinct*, *sorted*, *peak*, *limit*, *skip*, *takeWhile*, and *dropWhile*, the Java stream API only has a limited selection of intermediate stream operations. Alternative operations, such as *window* and *fold*, are not available.

Instead of responding to the diverse wishes of the community and integrating all the requested methods into the JDK, the JDK developers decided instead to develop an API that enables both the JDK developers themselves and the Java community to write any intermediate stream operations. JEP 485 delivers the new *Stream Gatherers* API as well as a whole series of predefined gatherers, including the requested *window* and *fold* operations.

Using the *fixed* and *sliding-window* operations, for example, you can group stream elements into lists of predefined sizes:

```
List<String> words = List.of("the", "be", "two", "of", "and", "a", "in", "that");
List<List<String>> fixedWindows = words.stream().gather(Gatherers.windowFixed(3)).toList();
List<List<String>> slidingWindows = words.stream().gather(Gatherers.windowSliding(3)).toList();
System.out.println(fixedWindows);
System.out.println(slidingWindows);
```

This small example program prints the following:

```
[[the, be, two], [of, and, a], [in, that]]
[[the, be, two], [be, two, of], [two, of, and], [of, and, a], [and, a, in], [a, in, that]]
```

You can find out which other stream gatherers the JDK provides and how you can write your own gatherers for intermediate operations at <https://www.happycoders.eu/java/stream-gatherers/>.

## Synchronize Virtual Threads without Pinning – JEP 491

Since Java 21, virtual threads were “pinned” to their carrier thread when blocking code was called within a *synchronized* block. Therefore, the carrier thread was blocked and could not execute any other virtual threads. This could lead to serious performance problems.

As of Java 24, this problem has been fixed: Virtual threads are now unmounted from the carrier thread during synchronization so that the carrier thread can continue to serve other virtual threads.

Why did virtual threads have to be pinned in the first place, and why does it work without pinning now? Read the answers in [https://www.happycoders.eu/java/java-24-features/#Synchronize\\_Virtual\\_Threads\\_Without\\_Pinning\\_JEP\\_491](https://www.happycoders.eu/java/java-24-features/#Synchronize_Virtual_Threads_Without_Pinning_JEP_491).

## Ahead-of-Time Class Loading & Linking – JEP 483

When starting a Java application, countless classes have to be read, parsed, loaded and linked. Especially for large backend applications, this can lead to long startup times.

Project Leyden has long been working on solutions to execute as many of these preparatory tasks as possible *before* an application is started. JDK Enhancement Proposal 483 introduces the first of these solutions in Java 24: Ahead-of-Time Class Loading & Linking.

In a preparatory phase, all classes required by the application are read, parsed, loaded, and linked, and then saved in this state in a cache. When the application is started, these steps no longer need to be carried out; the application can access the loaded and linked classes directly via the cache.

You can find out exactly how this works, a step-by-step guide to try it out and a comparison with AppCDS at <https://www.happycoders.eu/java/ahead-of-time-class-loading-and-linking/>.

## Warn upon Use of Memory-Access Methods in sun.misc.Unsafe – JEP 498

The *sun.misc.Unsafe* class introduced with Java 1.4 enables fast direct memory access, but it is also risky. Although it was never intended for direct use by developers, it was frequently used due to a lack of alternatives. Today, safe alternatives are available with *VarHandles* (since Java 9) and the *Foreign Function & Memory API* (since Java 22).

For this reason, *sun.misc.Unsafe* is being phased out: in Java 23, the affected methods were marked as *deprecated for removal*; Java 24 issues warnings at runtime; presumably from Java 26, an *UnsupportedOperationException* is thrown; and in a later version, the methods will be removed for good.

The VM option `--sun-misc-unsafe-memory-access` allows warnings to be temporarily deactivated (*allow*), further details to be printed (*debug*) or an *UnsupportedOperationException* to be thrown (*deny*).

# The Latest Features in Java 24 (Part 2)

You can find more details and the latest version of this PDF at [www.happycoders.eu/java-versions](http://www.happycoders.eu/java-versions).

## Late Barrier Expansion for G1 – JEP 475

In garbage collection, a “barrier” refers to additional code that is executed before or after objects are accessed. For example, write barriers help the garbage collector to track references between memory areas, while read barriers ensure that moved objects are correctly addressed.

These barriers are automatically inserted into the machine code when a Java program is loaded. The term “expansion” refers to the fact that the compact bytecode of a Java application is converted into more memory-intensive machine code through optimizations such as inlining or loop unrolling.

Previously, the G1 garbage collector employed “early barrier expansion”, in which barrier code was combined with the application code in a platform-independent “intermediate representation” (IR) and then optimized together with the application code. This had the advantage of a uniform intermediate representation, but led to longer compilation times and optimization errors that were difficult to reproduce.

JEP 475 therefore introduces “Late Barrier Expansion”: The barrier code is now available as optimized machine code and is only inserted *after* the application code optimization. This reduces the compiling effort, and applications run around 10-20% faster. Late Barrier Expansion has been used in ZGC already since Java 15.

## ZGC: Remove the Non-Generational Mode – JEP 490

In Java 21, the “Generational Mode” was introduced for the Z Garbage Collector (ZGC). Generational mode divides objects into a young and an old generation in order to optimize garbage collection. This mode was activated by default since Java 23, but could be deactivated using the VM option `-XX:+UseZGC -XX:-ZGenerational`.

With Java 24, the “Non-Generational Mode” is removed to simplify maintenance of the GC code. The `-XX:-ZGenerational` option no longer has any effect and generates a warning; it will be completely removed in a future version.

## Linking Run-Time Images without JMODs – JEP 493

A JDK installation consists of a runtime image and a set of Java module files in the *jmod* directory. While the runtime image in the *lib/modules* file is used to run a Java application, the module files in the *jmod* directory are used to create custom runtime images using the *jlink* tool.

In the future, it will be possible to create a JDK without *jmod* files. This can reduce the size of a JDK by around 25%, which is particularly advantageous in the cloud environment. The option is initially deactivated, but it could be activated by default in a future Java version.

## Weitere Änderungen in Java 24

**Class-File API (JEP 484)** – The Class-File API is an interface for reading and writing *.class* files, i.e., compiled Java bytecode. The new API is intended to replace the bytecode manipulation framework ASM, which is used intensively in the JDK.

**Quantum-Resistant Module-Lattice-Based Key Encapsulation Mechanism (JEP 496)** – Future quantum computers pose a threat to traditional cryptographic algorithms such as RSA and Diffie-Hellman. The use of ML-KEM (Module-Lattice-Based Key Encapsulation Mechanism) should make it possible in future to exchange keys securely even in the age of quantum computers. You can find an example of this in the [online article on Java 24](#).

**Quantum-Resistant Module-Lattice-Based Digital Signature Algorithm (JEP 497)** – Analogous to the ML-KEM method described in the previous section, the ML-DSA method (Module-Lattice-Based Digital Signature Algorithm), which is also quantum-safe, is included in the JDK. You can also find an example of this in the [Java 24 article](#).

**Permanently Disable the Security Manager (JEP 486)** – The “Security Manager”, which was originally developed to secure Java applets and is almost irrelevant for today’s Java applications, was marked as “deprecated for removal” in Java 17. This was intended to allocate the resources used to maintain the Security Manager to more important projects. In Java 24, the Security Manager can no longer be activated, either when starting an application or during runtime. The attempt leads to an error message.

**Remove the Windows 32-bit x86 Port (JEP 479)** – As there is hardly any need for this port, and maintenance is time-consuming (virtual threads, for example, were not even implemented here), this port has been removed.

**Deprecate the 32-bit x86 Port for Removal (JEP 501)** – With the removal of the 32-bit port for Windows, the 32-bit port for Linux is the last remaining 32-bit port – and thus the last port for which the JDK developers need to implement fallbacks for the 32-bit architecture. To eliminate this extra effort, the Linux 32-bit port is now marked as “deprecated for removal”.

**Deprecate LockingMode Option, along with LM\_LEGACY and LM\_MONITOR** – Java 21 introduced the new “Lightweight Locking” mode for object monitor locking; from Java 23, Lightweight Locking became the default mode. With Java 24, the option `-XX:LockingMode`, with which the “Legacy” mode could be reactivated, is marked as “deprecated”.

# Bonus: Preview Features in Java 24 (Part 1)

You can find more details and the latest version of this PDF at [www.happycoders.eu/java-versions](http://www.happycoders.eu/java-versions).

## Flexible Constructor Bodies (Third Preview) – JEP 492

Previously, it was not possible to execute additional code before calling the super constructor with *super(...)* or before calling an alternative constructor with *this(...)*. The validation of parameters or the calculation of arguments for the call of *super(...)* or *this(...)*, therefore, had to be extracted to separate methods, which could make the code quite unreadable.

Also, the fact that we could initialize fields in derived classes only after calling *super(...)* could lead to problems: If the super constructor calls a method that has been overwritten in the derived class and accesses its fields, then these fields are still uninitialized at the time the method is called.

Thanks to JDK Enhancement Proposal 492, code such as the following is now also valid:

```
public class Shape {
    private final Color color;

    public Shape(Color color) {
        this.color = color;
        printMe();
    }

    void printMe() {
        System.out.println(
            "color = " + color);
    }
}

public class Rectangle extends Shape {
    private final double width;
    private final double height;

    public Rectangle(Color color, double width, double height) {
        if (width < 0 || height < 0) throw new IllegalArgumentException();
        this.width = width;
        this.height = height;
        super(color);
    }

    void printMe() {
        super.printMe();
        System.out.println("width = " + width + ", height = " + height);
    }
}
```

The following restrictions apply:

In the code before *super(...)* or *this(...)* – the so-called prologue – the code may *initialize* fields but not *read* the class's fields and may not call any of its non-static methods. It must also not create instances of non-static inner classes, as these would then have a reference to the potentially uninitialized parent object.

## Simple Source Files and Instance Main Methods (Fourth Preview) – JEP 495

In future, a simple Java program can be written without a class declaration and with a non-static, non-public, parameterless *main* method. In addition, you can omit *System.out* before *println()*.

```
void main() {
    println("Hello, world!");
}
```

This makes the code on the right a valid and complete Java program.

## Generational Shenandoah (Experimental) – JEP 404

The garbage collectors ZGC and Shenandoah, which specialize in extremely low pause times, originally made no distinction between old and new objects and therefore did not use the “weak generation hypothesis”, which states that most objects die again shortly after their creation and that those objects that have already reached a certain age will generally live even longer.

A “generational garbage collector”, on the other hand, divides the heap into a young and an old generation, which improves performance as older objects need to be cleaned up less often. In Java 21, a “generational mode” was introduced for ZGC, which has been activated by default since Java 23.

In Java 24, the “Generational Mode” is now also introduced for Shenandoah GC. It can be activated with the VM options `-XX:+UnlockExperimentalVMOptions -XX:ShenandoahGCMode=generational`.

## Compact Object Headers (Experimental) – JEP 450

Each Java object has an object header that is currently 12 bytes in size (or 16 bytes if compressed class pointers are deactivated). In Project Lilliput, JDK developers have been working for years on further compressing the header. In Java 24, “Compact Object Headers” can be activated for the first time, initially reducing the header to 8 bytes.

This was achieved by removing unused bits and compressing the so-called “class word” from 32 bits to 22 bits by storing the class information at memory addresses divisible by 1,024 (=  $2^{10}$ ) in the metaspace, so that the lower 10 bits of the address are always 0 and can, therefore, be omitted.

Compact Object Headers are still experimental in Java 24 and must be activated with `-XX:+UnlockExperimentalVMOptions -XX:+UseCompactObjectHeaders`.

# Bonus: Preview Features in Java 24 (Part 2)

You can find more details and the latest version of this PDF at [www.happycoders.eu/java-versions](http://www.happycoders.eu/java-versions).

## Structured Concurrency (Fourth Preview) – JEP 499

*Structured concurrency* enables the simple coordination of concurrent tasks and is characterized by clearly visible start and end points of concurrent subtasks in the code – whereas with *unstructured concurrency*, the start and end of subtasks are much more difficult to recognize in the code (see figure on the right).

With *StructuredTaskScope*, JEP 499 introduces a control structure that clearly defines the start and end of concurrent tasks, enables clean error handling and can cancel subtasks whose results are no longer required in an orderly manner.

In the following example, we concurrently load an order, a customer and an invoice template and create an invoice from them. As soon as one of the subtasks fails, the subtasks that are still running are terminated and the error is propagated to the caller:

```
Invoice createInvoice(int orderId, int customerId, String language)
    throws InterruptedException, ExecutionException {
    try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
        Subtask<Order>      orderSubtask    = scope.fork(() -> orderService.getOrder(orderId));
        Subtask<Customer>   customerSubtask = scope.fork(() -> customerService.getCustomer(customerId));
        Subtask<InvoiceTemplate> templateSubtask = scope.fork(() -> templateService.getTemplate(language));

        scope.join();
        scope.throwIfFailed();

        var order      = orderSubtask.get();
        var customer    = customerSubtask.get();
        var template    = templateSubtask.get();

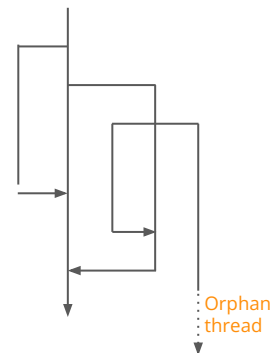
        return Invoice.from(order, customer, template);
    }
}
```

We can just as easily implement a *race()* method that starts two tasks, returns the result of the task that finishes first and cancels the second task (see example on the right).

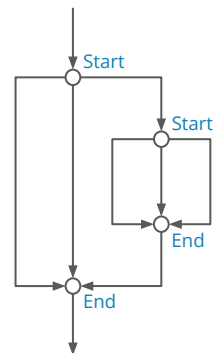
You can find a detailed description at <https://www.happycoders.eu/java/structured-concurrency/>.

```
public static <R> R race(Callable<R> task1, Callable<R> task2)
    throws InterruptedException, ExecutionException {
    try (var scope = new StructuredTaskScope.ShutdownOnSuccess<R>()) {
        scope.fork(task1);
        scope.fork(task2);
        scope.join();
        return scope.result();
    }
}
```

Unstructured Concurrency



Structured Concurrency



## Scoped Values (Fourth Preview) – JEP 487

*Scoped values* allow one or more variables to be passed to one or more methods without having to define them as explicit parameters and pass them from one method to the next.

The following example shows how a web server stores the logged-in user as a *scoped value* and calls the *restAdapter.process(...)* method in its scope. The call chain ultimately leads to the call of *Repository.getData(...)*, and this method can then access the logged-in user via the *scoped value*:

```
public class Server {
    public final static ScopedValue<User>
        LOGGED_IN_USER = ScopedValue.newInstance();
    ...
    private void serve(Request request) {
        User loggedInUser = authenticateUser(request);
        ScopedValue
            .where(LOGGED_IN_USER, loggedInUser)
            .run(() -> restAdapter.process(request));
    }
    ...
}

public class Repository {
    ...
    public Data getData(UUID id) {
        Data data = findById(id);
        User loggedInUser = Server.LOGGED_IN_USER.get();
        if (loggedInUser.isAdmin()) {
            enrichDataWithAdminInfos(data);
        }
        return data;
    }
    ...
}
```

You can find a detailed description at <https://www.happycoders.eu/java/scoped-values/>.

# Bonus: Preview Features in Java 24 (Part 3)

You can find more details and the latest version of this PDF at [www.happycoders.eu/java-versions](http://www.happycoders.eu/java-versions).

## Module Import Declarations (Second Preview) – JEP 494

Since Java 1.0, all classes of the *java.lang* package are automatically imported in every *.java* file. Therefore, we can use classes such as *Object*, *String*, *Integer*, *Exception*, *Thread*, etc. without explicit *import* statements.

We have also always been able to import complete packages. For example, importing *java.util.\** means that we do not have to import classes such as *List*, *Set*, *Map*, *ArrayList*, *HashSet*, and *HashMap* individually.

JDK Enhancement Proposal 494 now enables us to import complete *modules*. To be more precise: all classes in the packages exported by the module. For example, we can import the complete *java.base* module as follows and use the classes of this module (in the example *List*, *Map*, *Collectors*, *Stream*) without further imports:

```
import module java.base;

public static Map<Character, List<String>> groupByFirstLetter(String... values) {
    return Stream.of(values).collect(
        Collectors.groupingBy(s -> Character.toUpperCase(s.charAt(0))));
}
```

If a class name is ambiguous, such as *Date*, then the class must be imported directly (i.e., with *import java.util.Date* or *import java.sql.Date*). In *JShell* and implicitly declared classes, the *java.base* module is imported automatically.

## Primitive Types in Patterns, instanceof, and switch (2nd Preview) – JEP 488

Since Java 16 ("Pattern Matching for instanceof") and Java 21 ("Pattern Matching for switch"), we can use *instanceof* and *switch* to check whether an object is of a particular type and, if so, bind this object to a variable of this type, execute a certain program path and use the new variable in this program path.

Thanks to JDK Enhancement Proposal 488, we can now also use all *primitive* types in pattern matching. However, the exact mode of operation is different from the use of objects – as there is no inheritance with primitive types:

Let *a* be a *variable* of a primitive type (i.e., *byte*, *short*, *int*, *long*, *float*, *double*, *char*, or *boolean*) and *B* one of these primitive types. Then, *a instanceof B* is *true* if the precise value of *a* can also be stored in a variable of type *B*.

Here is an example:

```
int value = ...
if (value instanceof byte b) {
    System.out.println("b = " + b);
}
```

You can read the code as follows: If the value of the *value* variable can also be stored in a *byte* variable, then assign this value to the *byte* variable *b* and print it. This would be the case for *value* = 5, for example, but not for *value* = 1000, as a *byte* can only store values from -128 to 127.

Just as with objects, you can, also for primitive types, add further checks directly in the *instanceof* check using *&&*. The following code, for example, only prints positive *byte* values (i.e., 1 to 127):

```
if (value instanceof byte b && b > 0) {
    System.out.println("b = " + b);
}
```

Matching from and against floating point numbers is particularly interesting – you can find out more at <https://www.happycoders.eu/java/java-23-features/>.

## Key Derivation Function API (Preview) – JEP 478

A key derivation function (KDF) is a method for deriving one or more new cryptographic keys from a secret value such as a password, a passphrase or a cryptographic key. A standardized API is required so that security providers can implement and offer KDF algorithms and we can use them in applications.

You can find out how this API works at <https://www.happycoders.eu/java/java-24-features/>.

## Vector API (Ninth Incubator) – JEP 489

The *Vector API* will make it possible to map vector operations such as the following to the vector instructions of modern CPUs as efficiently as possible. This means that such calculations can be performed in just a few CPU cycles (or in just a single CPU cycle up to a certain vector size):

$$\begin{bmatrix} 7 \\ 13 \\ 8 \end{bmatrix} + \begin{bmatrix} 16 \\ 3 \\ 6 \end{bmatrix} = \begin{bmatrix} 7+16 \\ 13+3 \\ 8+6 \end{bmatrix} = \begin{bmatrix} 23 \\ 16 \\ 14 \end{bmatrix}$$

As the Vector API is still in the incubator stage and may therefore still be subject to significant changes, I am not showing any code at this point.



# The Latest Features in Java 23

You can find more details and the latest version of this PDF at [www.happycoders.eu/java-versions](http://www.happycoders.eu/java-versions).

## Markdown Documentation Comments – JEP 467

To format JavaDoc comments, we have always had to use HTML. JDK Enhancement Proposal 467 allows us to write JavaDoc comments in Markdown from Java 23 onwards.

Markdown comments must be introduced with three slashes. The following example shows how the JavaDoc comment of the *Math.ceilMod(...)* method can be written in Markdown:

```
/// Returns the ceiling modulus of the `long` and `int` arguments.
///
/// The ceiling modulus is `r = x - (ceilDiv(x, y) * y)`,
/// has the opposite sign as the divisor `y` or is zero, and
/// is in the range of `-abs(y) < r < +abs(y)`.
///
/// The relationship between `ceilDiv` and `ceilMod` is such that:
///
/// - `ceilDiv(x, y) * y + ceilMod(x, y) == x`
///
/// For examples, see [#ceilMod(int, int)].
///
/// @param x the dividend
/// @param y the divisor
/// @return the ceiling modulus `x - (ceilDiv(x, y) * y)`
/// @throws ArithmeticException if the divisor `y` is zero
/// @see #ceilDiv(Long, int)
/// @since 18
```

The example uses:

- `...` to format source code,
- a blank line for vertical spacing,
- a hyphen for a bulleted list,
- and square brackets for links.

You can also use:

- **\*\*...\*\*** for bold,
- *\*...\** oder `_..._` for italics,
- ````` oder `~~~` for code blocks,
- indentation for code blocks,
- "1." for numbered lists

JavaDoc tags such as @param, @throws etc. can be used without changes. Within code or code blocks, they are not evaluated but printed as they are.

## ZGC: Generational Mode by Default – JEP 474

The “Generational Mode” of the Z Garbage Collector (ZGC) was introduced in Java 21. In this mode, the ZGC uses the “Weak Generational Hypothesis” and stores new and old objects in two separate areas: the “young generation” and the “old generation.” The young generation mainly contains short-lived objects and needs to be cleaned up more frequently, while the old generation contains long-lived objects and needs to be cleaned up less often.

Since Java 21, you can activate the generational mode with the VM option `-XX:+UseZGC -XX:+ZGenerational`.

As the generational mode leads to considerable performance increases for most use cases, it is active by default in Java 23 when the ZGC is activated with `-XX:+UseZGC`. Generational mode can be deactivated with `-XX:+UseZGC -XX:-ZGenerational`.

## Support for Duration Until Another Instant

To determine the duration between two *Instant* objects, you previously had to use *Duration.between(...)*. Since Java developers intuitively look for such a function in *Instant*, the *until(...)* method has been added to *Instant*:

```
Instant now = Instant.now();
Instant later = Instant.now().plus(ThreadLocalRandom.current().nextInt(), SECONDS);
Duration duration = now.until(later);
```

## Deprecate the Memory-Access Methods in sun.misc.Unsafe for Removal – JEP 471

The methods provided in *sun.misc.Unsafe* for accessing on- and off-heap memory were initially intended only for internal use within the JDK. However, due to a lack of alternatives, they were also used extensively outside the JDK.

Today, we have stable alternatives: Java 9 introduced *VarHandles*, and Java 22 introduced the *Foreign Function & Memory API*.

Therefore, JDK Enhancement Proposal 471 marked all memory-access methods within *sun.misc.Unsafe* as *deprecated for removal*. You can find a list of the affected methods and the respective alternatives in the JEP.

## Other Changes in Java 23

**Console Methods With Explicit Locale** – The methods *Console.printf(...)*, *format(...)*, *readLine(...)*, and *readPassword(...)* have been overloaded with a variant with an additional *Locale* parameter so that, e.g., numbers can be formatted country-specifically.

**Thread.suspend/resume, ThreadGroup.suspend/resume/stop are removed** – These unsafe methods, which were already marked as *deprecated* in Java 1.2, have been marked as *deprecated for removal* in recent years, and they throw an *UnsupportedOperationException* in newer Java versions. They were finally removed in Java 23.

**Removal of Module jdk.random** – All classes of the *jdk.random* module have been moved to the *java.base* module. If you are using the Java module system and have specified *requires jdk.random* somewhere, you can remove this entry in Java 23 (the *java.base* module is automatically included).

# The Latest Features in Java 22

You can find more details and the latest version of this PDF at [www.happycoders.eu/java-versions](http://www.happycoders.eu/java-versions).

## Unnamed Variables & Patterns – JEP 456

As of Java 22, you can use the underscore "\_" to mark variables, pattern variables and entire patterns that you will not use. For example, in exceptions, lambdas, and pattern matching:

```
try {
    number = Integer.parseInt(string);
} catch (NumberFormatException _) {
    System.err.println("Not a number");
}

if (object instanceof Path(Position(int x1, int y1), _)) {
    System.out.printf("object is a path starting at x = %d, y = %d\n", x1, y1);
}

switch (object) {
    case Byte _, Short _, Integer _, Long _ -> System.out.println("Integer number");
    case Float _, Double _ -> System.out.println("Floating point number");
    default -> System.out.println("Not a number");
}

map.computeIfAbsent(key, _ -> new ArrayList<>())
    .add(value);
```

## Launch Multi-File Source-Code Programs – JEP 458

Since Java 11, we have been able to run Java programs consisting of just one file directly without having to explicitly compile them with *javac* beforehand. In Java 22, this capability has been extended so that programs consisting of several files can now also be started without prior compilation.

For example, save the following two classes in one file each, *Hello.java* and *Greetings.java*:

```
public class Hello {
    public static void main(String[] args) {
        System.out.println(Greetings.greet(args[0]));
    }
}

public class Greetings {
    public static String greet(String name) {
        return "Hello %s!".formatted(name);
    }
}
```

You can then start the program as follows without first compiling it with *javac*:

```
$ java Hello.java World
Hello World!
```

## Foreign Function & Memory API – JEP 454

The Foreign Function & Memory API (or FFM API for short) makes it possible to access code outside the Java Virtual Machine (e.g., functions in libraries written in other programming languages than Java) and native memory (i.e., memory that is not managed by the JVM on the heap) from Java.

In the example on the right, we call the *strlen()* function of the standard C library to determine the length of the string "Happy Coding!".

```
Linker linker = Linker.nativeLinker();
SymbolLookup stdlib = linker.defaultLookup();
MethodHandle strlen = linker.downcallHandle(
    stdlib.find("strlen").orElseThrow(),
    FunctionDescriptor.of(JAVA_LONG, ADDRESS));
try (Arena offHeap = Arena.ofConfined()) {
    MemorySegment str = offHeap.allocateFrom("Happy Coding!");
    long len = (long) strlen.invoke(str);
    System.out.println("len = " + len);
}
```

You can find a detailed description at <https://www.happycoders.eu/java/foreign-function-memory-api/>.

## Region Pinning for G1 – JEP 423

While working with memory addresses of Java objects when working with JNI (Java Native Interface), the garbage collector must not move these objects to another position in memory. As the G1 garbage collector was previously unable to protect individual objects from being moved, garbage collection was completely paused until these memory addresses were released again. Depending on the behavior of an application, this could have drastic consequences for memory consumption and performance.

Thanks to "region pinning" added in Java 22, the G1 Garbage Collector can now "pin" a memory region containing such an object, i.e., not move the objects it contains to another memory address, so that garbage collection no longer has to be paused completely.

# The Latest Features in Java 21 (Part 1)

You can find more details and the latest version of this PDF at [www.happycoders.eu/java-versions](http://www.happycoders.eu/java-versions).

You can find the non-finalized JDK Enhancement Proposals from Java 21 on pages 2 and 3.

With the finalization of *virtual threads*, *pattern matching for switch*, and *record patterns*, the Java 21 long-term support (LTS) release is a milestone in Java history. I could not summarize these and the other exciting Java 21 features on a single page. I have, therefore, divided them into two pages.

## Virtual Threads – JEP 444 🌟

When scaling server applications, threads are often the bottleneck. Their number is limited, and they often have to wait for events, such as the response to a database query or a remote call, or they are blocked by locks.

Existing solutions such as *CompletableFuture* or reactive frameworks lead to code that is difficult to read and maintain, e.g:

```
@GET
@Path("/product/{productId}")
public CompletionStage<Response> getProduct(@PathParam("productId") String productId) {
    return productService.getProductAsync(productId).thenCompose(product -> {
        if (product.isEmpty()) {
            return CompletableFuture.completedFuture(Response.status(NOT_FOUND).build());
        }
        return warehouseService
            .isAvailableAsync(productId)
            .thenCompose(available -> available
                ? CompletableFuture.completedFuture(0)
                : supplierService.getDeliveryTimeAsync(product.get().supplier(), productId))
            .thenApply(daysUntilShippable ->
                Response.ok(new ProductPageResponse(product.get(), daysUntilShippable)).build());
    });
}
```

Unlike asynchronous code like the one above, virtual threads allow programming in the traditional, familiar, imperative thread-per-request style (the `@RunOnVirtualThread` annotation is defined in Jakarta EE 11):

```
@GET
@Path("/product/{productId}")
@RunOnVirtualThread
public ProductPageResponse getProduct(@PathParam("productId") String productId) {
    Product product = productService.getProduct(productId).orElseThrow(NotFoundException::new);
    boolean available = warehouseService.isAvailable(productId);
    int shipsInDays = available ? 0 : supplierService.getDeliveryTime(product.supplier(), productId);
    return new ProductPageResponse(product, shipsInDays, Thread.currentThread().toString());
}
```

Sequential code is not only much easier to write and read, but also easier to debug, as the program flow can be tracked in a debugger and stack traces reflect the expected call stack.

This is made possible by mapping virtual threads – completely transparently for the developer – to a few platform threads (as the conventional threads are now called). As soon as a virtual thread has to wait or is blocked, the platform thread executes another virtual thread. In this way, we can execute several million (!) virtual threads with just a few operating system threads.

The best part is that we don't have to change existing code. We only have to instruct our application framework not to create platform threads but virtual threads (e.g. with `@RunOnVirtualThread`).

You can find out exactly how virtual threads work, what they can and cannot do, and for which purposes they are not suitable at <https://www.happycoders.eu/java/virtual-threads/>.

## Sequenced Collections – JEP 431 🌟

The new *SequencedCollection* interface offers uniform methods to read, remove or insert the first and last element of an ordered *Collection*, such as *List*, *Deque*, *LinkedHashSet*, or *TreeSet*. Below, you will find a few examples of how Java code can be greatly simplified using the new interface methods:

Operation	Code before Java 21	Code since Java 21
Reading the first element of a <i>List</i>	<code>list.get(0);</code>	<code>list.getFirst();</code>
Reading the last element of a <i>List</i>	<code>list.get(list.size() - 1);</code>	<code>list.getLast();</code>
Reading the first element of a <i>LinkedHashSet</i>	<code>linkedHashSet.iterator().next();</code>	<code>linkedHashSet.getFirst();</code>
Reading the last element of a <i>LinkedHashSet</i>	<i>only possible via iteration over the complete set</i>	<code>linkedHashSet.getLast();</code>

Other methods of *SequencedCollection* are *addFirst(...)*, *addLast(...)*, *removeFirst(...)*, *removeLast(...)*, and *reversed()*, which can be used to iterate over the *Collection* in reverse order.

Another interface is *SequencedMap*, which is implemented by *LinkedHashMap*, *TreeMap*, and *ConcurrentSkipListMap* and provides the methods *firstEntry()*, *lastEntry()*, *pollFirstEntry()*, *pollLastEntry()*, *putFirst(...)*, *putLast(...)*, and *reversed()*.



# The Latest Features in Java 21 (Part 2)

You can find more details and the latest version of this PDF at [www.happycoders.eu/java-versions](http://www.happycoders.eu/java-versions).  
You can find the non-finalized JDK Enhancement Proposals from Java 21 on pages 2 and 3.

## Pattern Matching for switch – JEP 441

*Pattern Matching for switch* makes it possible to formulate switch statements and expressions over any object. You can now also use patterns, qualified enum constants (preceded by a type name) and *null* as case labels. An example:

```
Object o = . . .
switch (o) {
    case String s when s.length() > 5 -> System.out.println("o is a long String, its length is " + s.length());
    case String s -> System.out.println("o is a short String: " + s);
    case Month.JANUARY -> System.out.println("o is the month of January");
    case null -> System.out.println("o is null");
    default -> System.out.println("o is something else");
}
```

When using one of the new switch features (pattern, qualified enum constant or *case null* label), the switch must be exhaustive, i.e., it must cover all conceivable cases or contain a *default* label.

This enables highly expressive code, especially in combination with sealed classes. Let *Shape* be a sealed class that only allows the child classes *Rectangle* and *Circle*. Then the following switch statement is considered exhaustive:

```
Shape shape = . . .
switch (shape) {
    case Rectangle r -> System.out.printf("Rectangle: top left = %s; bottom right = %s\n", r.tl(), r.br());
    case Circle c -> System.out.printf("Circle: center = %s; radius = %f\n", c.center(), c.radius());
}
```

If you now extend *Shape* with another child class, the compiler will no longer recognize this switch as complete, and you will be forced to extend it with the new class. This eliminates many potential sources of error.

## Record Patterns – JEP 440

A *record pattern* can be used to deconstruct a Java record into its components during pattern matching. Let *Position* be a record with x and y coordinates and *Path* a record with a start and end position. Then, for example, the following is possible:

```
switch (o) {
    case Position(int x, int y) -> System.out.printf("o is a position: %d/%d", x, y);
    case Path(Position(int x1, int y1),
              Position(int x2, int y2)) -> System.out.printf("o is a path: %d/%d -> %d/%d", x1, y1, x2, y2);
    case String s -> System.out.printf("o is a string: %s\n", s);
    default -> System.out.printf("o is something else: %s", o);
}
```

## Generational ZGC – JEP 439

The "Weak Generational Hypothesis" states that most objects die shortly after they have been created, and that objects that have survived several GC cycles usually stay alive even longer.

A so-called "generational garbage collector" makes use of this hypothesis by dividing the heap into two logical areas: a "young generation", in which new objects are created, and an "old generation", into which objects that have reached a certain age are moved. As objects in the old generation are very likely to become even older, the performance of an application can be increased by the garbage collector scanning the old generation less frequently.

Previously, the Z Garbage Collector, or ZGC for short, made no distinction between "old" and "new" objects. This changes in Java 21 if you activate the generational ZGC with the following VM options: `-XX:+UseZGC -XX:+ZGenerational`

## Other Changes in Java 21

**Generational Shenandoah (Experimental) (JEP 404)** – In addition to the Z Garbage Collector, the "Shenandoah Garbage Collector" introduced in Java 15 will also support generations in the future.

**Deprecate the Windows 32-bit x86 Port for Removal (JEP 449)** – The 32-bit version of Windows 10 is hardly used anymore, support ends in October 2025, and Windows 11 – on the market since October 2021 – has never been published in a 32-bit version. Accordingly, the 32-bit Windows port has been marked as "deprecated for removal".

**Prepare to Disallow the Dynamic Loading of Agents (JEP 451)** – Loading agents at runtime, e.g., by a profiler, represents a significant security risk and must be explicitly permitted in future.

**Key Encapsulation Mechanism API (JEP 452)** – Key Encapsulation Mechanism (KEM) is a modern encryption technology that enables the exchange of symmetric keys via an asymmetric encryption method.

# The Latest Features in Java 20

You can find more details and the latest version of this PDF at [www.happycoders.eu/java-versions](http://www.happycoders.eu/java-versions).

The preview features published in Java 20 – Scoped Values (JEP 429), Record Patterns (JEP 432), Pattern Matching for switch (JEP 433), Foreign Function & Memory API (JEP 434), Virtual Threads (JEP 436), Structured Concurrency (JEP 437), and Vector API (JEP 438) – have been finalized in Java 21 and Java 22 or are still in the preview stage as of Java 22.

No final JDK Enhancement Proposals (JEPs) were implemented in Java 20.  
In addition to the preview JEPs mentioned above, Java 20 contains the following notable changes:

## Javac Warns about Type Casts in Compound Assignments with Possible Lossy Conversions ★

Let's start with a little quiz. What is the difference between the following two statements?

```
a += b;
a = a + b;
```

Most Java developers will say: There is none. But that is wrong. If, for example, *a* is a *short*, and *b* is an *int*, then the second line will result in a compiler error:

```
java: incompatible types: possible lossy conversion from int to short
```

That is because *a + b* results in an *int*, which may not be assigned to the *short* variable *a* without an explicit cast.

The first line, on the other hand, is permitted, as the compiler inserts an *implicit* cast in a so-called "compound assignment". If *a* is a *short*, then *a += b* is equivalent to:

```
a = (short) (a + b);
```

However, when casting from *int* to *short*, the upper 16 bits are truncated. Information is therefore lost, as the following example shows:

```
short a = 30_000;
int b = 50_000;
a += b;
System.out.println("a = " + a);
```

The program does not print 80000 (hexadecimal 0x13880), but 14464 (hexadecimal 0x3880).

To warn developers of this potentially undesirable behavior, a corresponding compiler warning was (finally!) added in Java 20.

## Idle Connection Timeouts for HTTP/2

The system property `jdk.httpclient.keepalive.timeout` can be used to set how long inactive HTTP/1.1 connections are kept open. As of Java 20, this property also applies to HTTP/2 connections.

Furthermore, the system property `jdk.httpclient.keepalive.timeout.h2` has been added, with which you can overwrite this value specifically for HTTP/2 connections.

## HttpClient Default Keep Alive Time is 30 Seconds

If the aforementioned system property `jdk.httpclient.keepalive.timeout` is not defined, a default value of 1,200 seconds applied until Java 19. In Java 20, the default value was reduced to 30 seconds.

## IdentityHashMap's Remove and Replace Methods Use Object Identity

*IdentityHashMap* is a special *Map* implementation in which keys are not considered equal if the `equals()` method returns *true*, but if the key objects are identical, i.e., comparing them using the `==` operator yields *true*.

However, when the default methods `remove(Object key, Object value)` and `replace(K key, V oldValue, V newValue)` were added to the *Map* interface in Java 8, the JDK developers forgot to overwrite these methods in *IdentityHashMap* to use `==` instead of `equals()`.

This error was corrected in Java 20, eight and a half years later. The fact that the bug was not noticed for so long is a sign that *IdentityHashMap* is little used and may contain other bugs.

# The Latest Features in Java 19

You can find more details and the latest version of this PDF at [www.happycoders.eu/java-versions](http://www.happycoders.eu/java-versions).

The preview features released in Java 19 – Record Patterns (JEP 405), Foreign Function & Memory API (JEP 424), Virtual Threads (JEP 425), Vector API (JEP 426), Pattern Matching for switch (JEP 427), and Structured Concurrency (JEP 428) – have been finalized in Java 21 and Java 22 or are still in the preview stage as of Java 22.

## New Methods to Create Preallocated HashMaps ★

*HashMap* provides a constructor with which you can define the initial capacity of the map, e.g., like this:

```
Map<String, String> map = new HashMap<>(120);
```

Intuitively, one would think that this *HashMap* offers space for 120 entries. However, this is not the case!

That is because the *HashMap* is initialized with a default load factor of 0.75. This means that, as soon as the *HashMap* is 75% full, it gets rebuilt ("rehashed") with twice the size. This is to ensure that the elements are distributed as evenly as possible across the *HashMap*'s buckets and that no bucket contains more than one element.

The *HashMap* initialized with a capacity of 120 can, therefore, only hold  $120 \times 0.75 = 90$  entries.

To create a *HashMap* for 120 entries, you previously had to calculate the capacity by dividing the number of mappings by the load factor:  $120 \div 0.75 = 160$ . A *HashMap* for 120 entries, therefore, had to be created as follows:

```
Map<String, Integer> map = new HashMap<>(160);
```

Java 19 makes it easier for us – we can now write the following instead:

```
Map<String, Integer> map = HashMap.newHashMap(120);
```

The *newHashMap(...)* method has also been implemented in *LinkedHashMap* and *WeakHashMap*. Take a look at the source code of the new methods – you will see that they perform exactly the same calculation that we performed manually before.

## Additional Date-Time Formats ★

Using the *DateTimeFormatter.ofLocalizedDate(...)*, *ofLocalizedTime(...)*, and *ofLocalizedDateTime(...)* methods and a subsequent call to *withLocale(...)*, we have been able to generate date and time formatters since Java 8. We can control the exact format using the *FormatStyle* enum, which can take the values *FULL*, *LONG*, *MEDIUM* and *SHORT*.

In Java 19, the *ofLocalizedPattern(String requestedTemplate)* method was added, which we can use to define more flexible formats. Here is an example:

```
DateTimeFormatter formatter = DateTimeFormatter.ofLocalizedPattern("yMMM");
System.out.println("US: " + formatter.withLocale(Locale.US).format(LocalDate.now()));
System.out.println("Germany: " + formatter.withLocale(Locale.GERMANY).format(LocalDate.now()));
System.out.println("Japan: " + formatter.withLocale(Locale.JAPAN).format(LocalDate.now()));
```

The code leads to the following output:

```
US: Jan 2024
Germany: Jan. 2024
Japan: 2024年1月
```

## New System Properties for System.out and System.err

If the automatic detection of the configured character set of the console/terminal does not work when printing to *System.out* or *System.err*, you can set the character set as of JDK 19 via the following VM options, for instance, to UTF-8:

```
-Dstdout.encoding=utf-8 -Dstderr.encoding=utf-8
```

If you don't want to do this every time you start the program, you can also configure these settings globally by defining the following environment variable (yes, it starts with an underscore):

```
_JAVA_OPTIONS="-Dstdout.encoding=utf-8 -Dstderr.encoding=utf-8"
```

## Linux/RISC-V Port – JEP 422

Due to the increasing use of RISC-V hardware, a Linux port for this architecture will be made available for the corresponding architecture from Java 19 onwards.

# The Latest Features in Java 18

You can find more details and the latest version of this PDF at [www.happycoders.eu/java-versions](http://www.happycoders.eu/java-versions).

The preview features published in Java 18 – Vector API (JEP 417), Foreign Function & Memory API (JEP 419), and Pattern Matching for switch (JEP 420) – have been finalized in Java 21 and Java 22 or are still in the preview stage in Java 22.

## UTF-8 by Default – JEP 400 🌟

The character encoding determines how strings are converted into bytes and vice versa in numerous JDK methods, e.g., in *FileReader*, *FileWriter*, *InputStreamReader*, *InputStreamWriter*, *Formatter*, and *Scanner*. Until Java 17, the standard character encoding depended on the operating system and language, which led to numerous problems, e.g., if a text file was written on one operating system and read on another without explicitly specifying the character encoding.

The standard character encoding could be overwritten using the system property *-Dfile.encoding*, but newer methods such as *Files.readString(...)* ignore the character encoding configured this way and always use UTF-8.

As of Java 18, the standard character encoding on all platforms and languages is UTF-8, so that the problems described above should be a thing of the past.

## Code Snippets in Java API Documentation – JEP 413 🌟

As of Java 18, we can insert code examples in JavaDoc comments with the *@snippet* tag. Parts of the code can be marked with *@highlight* or linked with *@link*, as in the example on the right.

You can also refer to marked code locations in other Java files. You can find more details in the [Java 18 article](#).

```
/**
 * {@snippet :
 * // @highlight region regex="\bwrite.*?\b" type="highlighted":
 * // @link regex="\bBufferedWriter\b" target="java.io.BufferedWriter":
 * try (BufferedWriter writer = Files.newBufferedWriter(path)) {
 *     writer.write(text);
 * }
 * // @end
 * }
 */
```

## Internet-Address Resolution SPI – JEP 418 ★

To find the IP address for a host name using *InetAddress.getByName(...)*, for example, the operating system's resolver is called, i.e., the *hosts* file is usually consulted first and then the configured DNS servers. Calling the corresponding operating system function is blocking and sometimes takes quite a long time. In addition, we cannot adjust the hostname resolution in tests, for example, in order to redirect requests to a test server.

The *Internet Address Resolution SPI* makes it very easy to replace the standard hostname resolver with your own. You can find out exactly how this works in the [Java 18 article](#).

## Simple Web Server – JEP 408

Java 18 comes with a rudimentary web server, which you can start with the command on the right.

Use *-b* to specify the IP address on which the web server

should accept connections, and *-p* to specify the port. If you omit this information, the web server will listen to 127.0.0.1:8000.

The *-d* parameter specifies the directory that the web server should deliver. By default, this is the current directory. You can also start the web server from Java code (in fact, the *jwebserver* command does just that):

```
SimpleFileServer.createServer(
    new InetSocketAddress("127.0.0.10", 4444), Path.of("/tmp"), OutputLevel.INFO).start();
```

```
$ jwebserver -b 127.0.0.100 -p 4444 -d /tmp
Serving /tmp and subdirectories on 127.0.0.100 port 4444
URL http://127.0.0.100:4444/
```

## Reimplement Core Reflection with Method Handles – JEP 416

Java has two reflection mechanisms – *core reflection* and *method/variable handles*:

```
Field field = String.class.getDeclaredField("value"); // ← Core reflection
VarHandle handle = MethodHandles.privateLookupIn(String.class, MethodHandles.Lookup()) // ← Method handles
    .findVarHandle(String.class, "value", byte[].class);
```

Maintaining both mechanisms represents a considerable amount of extra work. For this reason, the code of the reflection classes *java.lang.reflect.Method*, *Field*, and *Constructor* was re-implemented in Java 18 based on method handles.

## Deprecate Finalization for Removal – JEP 421

*Finalization*, which has existed since Java 1.0, is insecure, error-prone and can lead to performance problems. As alternatives, *try-with-resources* was introduced in Java 7, and the *Cleaner API* in Java 9. Accordingly, the *finalize()* methods in *Object* and numerous other classes of the JDK class library were marked as "deprecated" in Java 9. As of Java 18, the *finalize()* methods are marked as "deprecated for removal".

# The Latest Features in Java 17 (Part 1)

You can find more details and the latest version of this PDF at [www.happycoders.eu/java-versions](http://www.happycoders.eu/java-versions).

## Sealed Classes – JEP 409 🌟

Until now, there were only limited options for restricting the inheritability of a class – by making it *final* or marking it as non-*public*. In Java 17, we can use the keywords *sealed*, *non-sealed*, and *permits* to control inheritability in a much more granular way. In the example on the right, the *Shape* interface may only be implemented by the *Circle* and *Square* classes.

```
public sealed class Shape permits Circle, Square {}
public final class Circle extends Shape {}
public final class Square extends Shape {}
```

*Sealed classes* enable, for example, exhaustiveness analysis in *Pattern Matching for switch*, introduced in Java 21. You can find further use cases and characteristics at <https://www.happycoders.eu/java/sealed-classes/>.

## Strongly Encapsulate JDK Internals – JEP 403 🌟

From Java 9 to 15, you have probably seen the warning *"WARNING: An illegal reflective access operation has occurred"*. The reason for this is that for a transitional period after the introduction of the module system in Java 9, deep reflection on code that existed before Java 9 was permitted in the so-called "relaxed strong encapsulation" mode. In Java 16, the default mode was changed to "strong encapsulation", which led to an error and program termination instead of a warning. With the VM option *--illegal-access=permit*, you could restore the previous behavior in Java 16.

In Java 17, only "strong encapsulation" mode is available; the VM option no longer has any effect and generates a warning. Reflective access must now be explicitly allowed for specific modules and packages using the following VM option:

```
--add-opens <name of the called module>/<package name>=<name of the calling module or ALL-UNNAMED>
```

## Add java.time.InstantSource 🌟

The *java.time.Clock* class is extremely useful for writing tests that check time-dependent functionality. If you inject *Clock* into the application classes via dependency injection, for example, you can mock it in tests and set a specific time for the test using *Clock.fixed(...)*, for example.

However, as *Clock* provides the *getZone()* method, you always have to think about the specific time zone with which you instantiate a *Clock* object.

To enable time zone-independent time sources, the *java.time.InstantSource* interface was extracted from *Clock* in Java 17. It only provides the *instant()* and *millis()* methods for querying the time, with *millis()* being implemented as a default method.

The example at the top right shows a method that measures the execution time of a *Runnable*. In production, we can instantiate *Timer* with the system clock:

```
Timer timer = new Timer(Clock.systemDefaultZone());
```

We can test the method by mocking *InstantSource*, having the *instant()* method return two constant values and comparing the return value of *measure(...)* with the difference between these values (see test method on the right).

```
public class Timer {
    private final InstantSource instantSource;
    public Timer(InstantSource instantSource) {
        this.instantSource = instantSource;
    }
    public Duration measure(Runnable runnable) {
        Instant start = instantSource.instant();
        runnable.run();
        Instant end = instantSource.instant();
        return Duration.between(start, end);
    }
}

@Test
void shouldReturnDurationBetweenStartAndEnd() {
    var instantSource = mock(InstantSource.class);
    when(instantSource.instant())
        .thenReturn(ofEpochMilli(1640033566000L))
        .thenReturn(ofEpochMilli(1640033566075L));
    Timer timer = new Timer(instantSource);
    Duration duration = timer.measure(() -> {});
    assertThat(duration).isEqualTo(ofMillis(75));
}
```

## Hex Formatting and Parsing Utility 🌟

Java 17 provides the new class *java.util.HexFormat*, which provides a standardized API for displaying and parsing hexadecimal numbers. *HexFormat* supports all primitive integers (*int*, *byte*, *char*, *long*, *short*) as well as byte arrays – but not floating point numbers. Here is an example:

```
HexFormat hf = HexFormat.of();
System.out.println(hf.toHexDigits('A')); // → 20ac
System.out.println(hf.toHexDigits((byte) 100)); // → 64
System.out.println(hf.toHexDigits((short) 10_000)); // → 2710
System.out.println(hf.toHexDigits(1_000_000_000)); // → 3b9aca00
System.out.println(hf.withUpperCase().formatHex(new byte[] {-54, -2, -70, -66})); // → CAFEBABE
System.out.println(hf.withDelimiter(" ").formatHex("Happy Coding!".getBytes())); // → 48 61 70 70 79 ...
```

You can find additional methods for displaying and parsing hexadecimal strings in the [HexFormat JavaDoc](#).



# The Latest Features in Java 17 (Part 2)

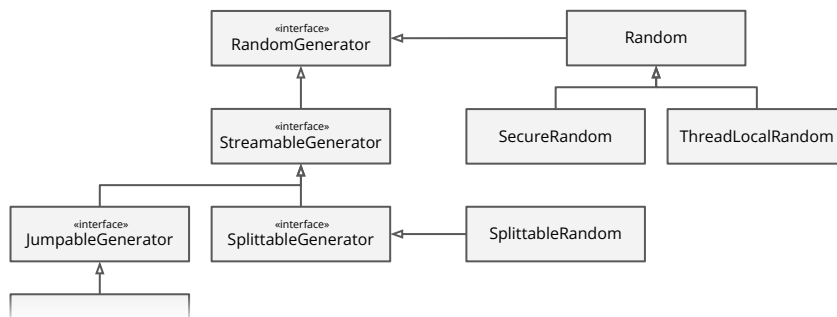
You can find more details and the latest version of this PDF at [www.happycoders.eu/java-versions](http://www.happycoders.eu/java-versions).

The preview features published in Java 17 – Pattern Matching for switch (JEP 406), Foreign Function & Memory API (JEP 412), and Vector API (JEP 414) – have been finalized in Java 21 and Java 22 or are still in the preview stage in Java 22.

## Enhanced Pseudo-Random Number Generators – JEP 356 ★

In Java 17, a set of inheriting interfaces was introduced for existing and new random number generators, so that the use of interfaces makes it much easier than before to exchange specific algorithms.

The methods common to all random number generators, such as `nextInt()` and `nextDouble()`, are defined in `RandomGenerator`. If you only need these methods, you should always use this interface in future.



## Context-Specific Deserialization Filters – JEP 415

The deserialization of objects poses a significant security risk. Attackers can construct objects via the data stream to be deserialized, which they can ultimately use to execute arbitrary code in arbitrary classes (available on the classpath). In Java 9, global and `ObjectInputStream`-specific deserialization filters were introduced, which can be used to specify which classes may be deserialized and which may not.

Java 17 extends this protection mechanism to include *context-specific* deserialization filters, which can be defined for a specific thread, class, module or third-party library, for example.

## Unified Logging Supports Asynchronous Log Flushing

With asynchronous logging, log messages are first written to a buffer by the application thread. A separate I/O thread then forwards them to the configured output (console, file, or network). This means that the application thread does not have to wait for the corresponding I/O subsystem to process the message.

While most logging frameworks support asynchronous logging, log messages from the JVM itself (e.g., those from the garbage collector) could previously only be written out synchronously.

As of Java 17, asynchronous logging can be activated for the JVM itself via the VM option `-Xlog:async`. The size of the log buffer can be defined via `-XX:AsyncLogBufferSize=<size in bytes>`.

## Other Changes in Java 17

**Restore Always-Strict Floating-Point Semantics (JEP 306)** – Before Java 1.2, floating point operations were executed "strictly" by default, i.e., they led to identical results on all architectures. Java 1.2 transitioned to the then faster "standard floating point semantics" of the processor architecture. If you wanted to continue to perform strict calculations, you had to mark a class with the `strictfp` keyword. As modern CPUs perform strict operations without loss of performance, strict calculations are used again by default from Java 17 onwards. The `strictfp` keyword is now superfluous and results in a compiler warning.

**New macOS Rendering Pipeline (JEP 382)** – The Swing rendering pipeline for macOS has been switched to the Metal framework, which replaced the OpenGL library previously used under macOS in 2018.

**macOS/AArch64 Port (JEP 391)** – Apple announced that it will be converting Macs from x64 to AArch64 CPUs in the long term. Accordingly, an AArch64 port will be provided.

**Deprecate the Applet API for Removal (JEP 398)** – Java applets are no longer supported by any modern web browser and were already marked as "deprecated" in Java 9. In Java 17, they are marked as "deprecated for removal".

**Remove RMI Activation (JEP 407)** – RMI Activation allows objects that have been destroyed on the target JVM to be instantiated again as soon as they are accessed. As this feature was time-consuming to maintain and rarely used, it was removed.

**Remove the Experimental AOT and JIT Compiler (JEP 410)** – The Graal compiler provided in Java 9 and 10 as an AOT or JIT compiler was only used a little and was removed from the Oracle JDK in Java 16 and also from the OpenJDK in Java 17.

**Deprecate the Security Manager for Removal (JEP 411)** – The Security Manager should protect the computer from downloaded Java applets. Just like Java applets, the Security Manager has now also been marked as "deprecated for removal".

# The Latest Features in Java 16

You can find more details and the latest version of this PDF at [www.happycoders.eu/java-versions](http://www.happycoders.eu/java-versions).

The previews published in Java 16 – Vector API (JEP 338), Foreign Linker API (JEP 389), Foreign-Memory Access API (JEP 393), and Sealed Classes (JEP 397) – will be finalized in Java 17 and Java 22 or are still in the preview stage in Java 22.

## Pattern Matching for instanceof – JEP 394 🌟

As of Java 16, we no longer have to explicitly cast a variable to the target type after an *instanceof* check. Instead, we can bind the variable to a new variable of the corresponding type, e.g., like this:

```
if (o instanceof String s && !s.isEmpty()) System.out.println("o in capital letters: " + s.toUpperCase());
```

## Records – JEP 395 🌟

Java records offer a compact notation method for defining classes with only final fields.

Find out what particularities you need to be aware of when working with records at <https://www.happycoders.eu/java/java-records/>.

```
record Position(int x, int y) {}
record Path(Point start, Point end) {}

Path p = new Path(new Pos(0,8), new Pos(1,5));
int startX = p.start().x();
```

## Strongly Encapsulate JDK Internals by Default – JEP 396 🌟

You have probably encountered the warning "*WARNING: An illegal reflective access operation has occurred*". This is because for a transitional period after the introduction of the Java module system in JDK 9, deep reflection on code that already existed before Java 9 was allowed by default in the so-called "relaxed strong encapsulation" mode.

In Java 16, the default mode is changed to "strong encapsulation", which now leads to an error and program termination instead of a warning. With the VM option `--illegal-access=permit`, you can temporarily restore the previous behavior. However, the more sustainable solution is to explicitly allow reflective access with the following VM option:

```
--add-opens <name of the called module>/<package name>=<name of the calling module or ALL-UNNAMED>
```

## New Stream Methods 🌟

The new *Stream.toList()* method is a short form for the frequently used *Stream.collect(Collectors.toUnmodifiableList())*.

*Stream.mapMulti(...)* is an alternative to *flatMap(...)* that does not require the creation of temporary streams. You can find out exactly how *mapMulti(...)* works in the [article on Java 16](#).

## Warnings for Value-Based Classes – JEP 390 🌟

The following classes were annotated as *@ValueBased* and their constructors marked as "deprecated for removal":

- all wrapper classes of the primitive data types (*Byte*, *Short*, *Integer*, *Long*, *Float*, *Double*, *Boolean*, and *Character*),
- *Optional* and its primitive variants,
- numerous classes of the date/time API, such as *LocalDateTime*,
- the collections created by *List.of()*, *Set.of()*, and *Map.of()*.

This is a preparation for *Value Objects* developed in [Project Valhalla](#), i.e., objects without identity and header.

## Migrate from Mercurial to Git + Migrate to GitHub – JEPs 357 + 369 🌟

With Java 16, the JDK code was migrated from the version management system Mercurial to the much more widely used Git. As hosting platform, GitHub was chosen. The JDK source code has since been available here: <https://github.com/openjdk/>

## Other Changes in Java 16

**Enable C++14 Language Features (JEP 347)** – The C++ part of the JDK was previously limited to the C++98/03 language specification – a standard that is over 20 years old. In JDK 16, support is increased to C++14.

**ZGC: Concurrent Thread-Stack Processing (JEP 376)** – The so-called "thread stack processing", which in ZGC was previously carried out in so-called safepoints, i.e., in stop-the-world phases, runs parallel to the running application from Java 16 onwards.

**Unix-Domain Socket Channels (JEP 380)** – As of version 16, Java supports Unix domain sockets, which are used for inter-process communication (IPC) within a host and are significantly faster than TCP sockets.

**Alpine Linux Port (JEP 386)** – The JDK is extended by a port for Alpine Linux, which is highly popular in cloud environments.

**Elastic Metaspace (JEP 387)** – In Java 16, the memory footprint of the metaspace, i.e., the memory area in which class metadata is stored, has been reduced. Metaspace memory is also returned to the operating system more quickly.

**Windows/AArch64 Port (JEP 388)** – Windows on the ARM64/AArch64 processor architecture has become an important platform. Accordingly, a port is also provided for this architecture, based on the existing Linux/AArch64 port.

**Packaging Tool (JEP 392)** – The *jpackage* tool packages a Java application together with the Java runtime environment (i.e., the JVM and the required modules of the JDK class library) as an installation package for various operating systems.

# The Latest Features in Java 15

You can find more details and the latest version of this PDF at [www.happycoders.eu/java-versions](http://www.happycoders.eu/java-versions).

The preview features published in Java 15 – Sealed Classes (JEP 360), Pattern Matching for instanceof (JEP 375), Foreign-Memory Access API (JEP 383) and Records (JEP 384) – will be finalized in Java 16, 17 and 22.

## Text Blocks – JEP 378 🌟

As of Java 15, you can code multi-line strings as shown on the right, delimited by three quotation marks on each side. You no longer need to escape individual quotation marks. The indentation common to all lines is removed.

You can find details at <https://www.happycoders.eu/java/java-text-blocks/>.

```
String sql = """
    SELECT id, title, text FROM Article
    WHERE category = "Java"
    ORDER BY title""";
```

## New String and CharSequence Methods 🌟

The *String* and *CharSequence* classes have been extended in Java 15 to include the following methods:

- *String.formatted(Object... args)* – placeholders in strings can now also be replaced as follows, for example:  
`String message = "User %,d %s logged in at %s".formatted(userId, username, ZonedDateTime.now())`
- *String.stripIndent()* – removes the same number of spaces from the beginning of each line of the string, so that the line with the previously lowest number of spaces starts at the beginning of the line. Also removes all trailing spaces.
- *String.translateEscapes()* – replaces escaped escape sequences such as `"\\n"` with evaluated escape sequences such as `"\n"`.
- *CharSequence.isEmpty()* – checks whether a character sequence, e.g., a *String* or *StringBuilder*, is empty.

## ZGC: A Scalable Low-Latency Garbage Collector – JEP 377 ★

The Z Garbage Collector, ZGC for short, promises not to exceed pause times of 10 ms while reducing the overall throughput of the application by no more than 15% compared to the G1 garbage collector (the reduction in throughput is the cost of the low latency). ZGC supports heap sizes up to 16 TB. Just like G1, ZGC is region-based, NUMA-aware and returns unused memory to the operating system.

You can activate ZGC with the following VM option: `-XX:+UseZGC`

The VM option `-XX:SoftMaxHeapSize` can be used to configure a "soft" upper heap limit, which ZGC only exceeds if it is absolutely necessary to avoid an *OutOfMemoryError*.

## Shenandoah: A Low-Pause-Time Garbage Collector – JEP 379

Likewise, Shenandoah promises minimal pause times regardless of the heap size. Shenandoah also returns unused memory to the operating system. However, there is currently no support for NUMA and *SoftMaxHeapSize*.

You can activate Shenandoah with the following VM option: `-XX:+UseShenandoahGC`

## Other Changes in Java 15

**Edwards-Curve Digital Signature Algorithm (EdDSA) (JEP 339)** – EdDSA is a modern signature method that is faster than previous signature methods such as DSA and ECDSA while offering the same level of security.

**Hidden Classes (JEP 371)** – Frameworks such as Jakarta EE and Spring generate numerous classes dynamically at runtime. Until now, these classes could not be distinguished from regular classes and were therefore visible to all other classes. As of Java 15, these framework-specific classes can be defined as "hidden" and thus remain concealed from the rest of the application.

**Remove the Nashorn JavaScript Engine (JEP 372)** – The JavaScript engine "Nashorn" introduced in Java 8 was marked as "deprecated for removal" in Java 11 and is completely removed in Java 15.

**Reimplement the Legacy DatagramSocket API (JEP 373)** – The *java.net.DatagramSocket* and *java.net.MulticastSocket* APIs, which have existed since Java 1.0, consist of a mixture of outdated Java and C code that is difficult to maintain and extend. The code contains some bugs and cannot be easily adapted to virtual threads developed in [Project Loom](https://openjdk.org/projects/loom/). Therefore, the old implementation has been replaced by a simpler, more modern, easier to maintain implementation.

**Disable and Deprecate Biased Locking (JEP 374)** – Biased locking is an optimization of thread synchronization that mainly benefits classes such as *Vector*, *Hashtable*, and *StringBuffer*, where every access is synchronized. As these classes are hardly used anymore, and biased locking is difficult to maintain, it is deactivated and marked as "deprecated".

**Remove the Solaris and SPARC Ports (JEP 381)** – The outdated Solaris and SPARC ports were marked as "deprecated" in Java 14 and have been removed from the JDK in Java 15.

**Deprecate RMI Activation for Removal (JEP 385)** – RMI (Remote Method Invocation) Activation allows objects that have been destroyed on the target JVM to be automatically instantiated again as soon as they are accessed. As this feature is complex to maintain and rarely used, it is marked as "deprecated".

**Compressed Heap Dumps** – As of Java 15, the `jcmd GC.heap_dump` command can save the heap dump in a ZIP-compressed way by using the option `-gz=<compression level>`. Level 1 is the fastest, level 9 the strongest compression.

# The Latest Features in Java 14

You can find more details and the latest version of this PDF at [www.happycoders.eu/java-versions](http://www.happycoders.eu/java-versions).

The preview features released in Java 14 – Pattern Matching for instanceof (JEP 305), Packaging Tool (JEP 343), Records (JEP 359), ZGC on macOS / Windows (JEPs 364 + 365), Text Blocks (JEP 368), and Foreign-Memory Access API (JEP 370) – will be finalized in Java 15, 16 and 22.

## Switch Expressions – JEP 361

As of Java 14, switch statements can be written in the so-called arrow notation. One or more constants are followed by an arrow and then a single instruction or a code block.

The error-prone fallthrough to the next label inherited from C no longer exists with this notation (see example 1).

Switch can now also be used as an expression and return a value. With a switch expression, for example, we can write a recursive Fibonacci function quite elegantly (see example 2).

Find more details about switch expressions at [www.happycoders.eu/java/switch-expressions/](http://www.happycoders.eu/java/switch-expressions/).

Example 1:

```
switch (dayOfWeek) {
    case MONDAY, TUESDAY -> System.out.print("Beginning");
    case WEDNESDAY, THURSDAY -> System.out.print("Middle");
    case FRIDAY -> System.out.print("Almost!!!");
    case SATURDAY, SUNDAY -> System.out.print("Weekend");
}
```

Example 2:

```
public static int fibonacci(int i) {
    return switch (i) {
        case 0, 1 -> i;
        default -> fibonacci(i - 1) + fibonacci(i - 2);
    };
}
```

## Helpful NullPointerExceptions – JEP 358

As of Java 14, a *NullPointerException* no longer only shows you in which line it occurred, but also which object in this line is *null*, for instance, like this:

```
Exception in thread "main" java.lang.NullPointerException:
Cannot invoke "Map.getValue()" because the return value of "Container.getMap()" is null
at eu.happycoders.BusinessLogic.calculate(BusinessLogic.java:80)
```

## Accounting Currency Format Support

In some countries, e.g., in the USA, negative numbers in accounting are not indicated by a minus sign, but by round brackets. Java 14 adds the so-called language tag extension "u-cf-account", which makes it possible to specify additional information in a *Locale* object as to whether it is used in the context of accounting:

```
NumberFormat nf = NumberFormat.getCurrencyInstance(Locale.US);
NumberFormat nfAccounting = NumberFormat.getCurrencyInstance(Locale.forLanguageTag("en-US-u-cf-account"));
System.out.println("Normal: " + nf.format(-14.95)); // → -$14.95
System.out.println("Accounting: " + nfAccounting.format(-14.95)); // → ($14.95)
```

## Other Changes in Java 14

**Non-Volatile Mapped Byte Buffers (JEP 352)** – You can use a *MappedByteBuffer* to map a file into memory so that it can then be read and modified using regular memory access operations. Modified data must be regularly transferred to the storage medium. For NVMMs, more efficient methods are available than for conventional storage media. As of Java 14, you can specify the *ExtendedMapMode.READ\_ONLY\_SYNC* or *READ\_WRITE\_SYNC* modes when calling *FileChannel.map(...)*.

**NUMA-Aware Memory Allocation for G1 (JEP 345)** – On modern servers, the physical distance between the CPU core and the memory module plays an increasingly important role. The G1 Garbage Collector can use NUMA (Non-Uniform Memory Access) data to allocate objects on the memory node that is closest to the CPU core on which the object-generating thread is running. NUMA-aware memory allocation is only available for Linux and must be activated via the VM option `+XX:+UseNUMA`.

**JFR Event Streaming (JEP 349)** – The Java Flight Recorder (JFR) collects data about the JVM during the execution of an application and saves it in a file. The data can then be visualized with JDK Mission Control. As of Java 14, continuous monitoring is also possible. You can find out how this works in the [Java 14 article](#).

**Deprecate the Solaris and SPARC Ports (JEP 362)** – Solaris and the SPARC processor architecture are no longer up to date. In order to be able to use development resources elsewhere, the corresponding ports are marked as "deprecated".

**Remove the Concurrent Mark Sweep (CMS) Garbage Collector (JEP 363)** – The Concurrent Mark Sweep (CMS) garbage collector was marked as "deprecated" in Java 9. Development resources are to be redistributed in favor of more modern garbage collectors such as G1GC (standard since Java 9) and ZGC. CMS is completely removed from the JDK in Java 14.

**Deprecate the ParallelScavenge + SerialOld GC Combination (JEP 366)** – The rarely used combination of GC algorithms, parallel for the young generation and serial for the old generation, activated by `-XX:+UseParallelGC -XX:-UseParallelOldGC`, is rarely used and requires a high maintenance effort. It is therefore marked as "deprecated".

**Remove the Pack200 Tools and API (JEP 367)** – The "Pack200" compression method was introduced in Java 5 in order to save as much bandwidth as possible in the early 2000s through more effective compression. In times of 100 Mbit lines, the development costs are no longer in proportion to the benefits. Pack200 is therefore being removed in Java 14.



# The Latest Features in Java 13

You can find more details and the latest version of this PDF at [www.happycoders.eu/java-versions](http://www.happycoders.eu/java-versions).

The preview features published in Java 13 – Switch Expressions (JEP 354), Text Blocks (JEP 355), and ZGC: Uncommit Unused Memory (351) – will all be finalized in Java 15.

## Dynamic CDS Archives – JEP 350

*Application Class-Data Sharing* (see Java 10) was previously quite complicated to set up. In addition, not only the classes of the application itself, but also all JDK classes in use were included in the shared archive file.

In Java 13, only the classes of the application are stored in the shared archive; the JDK classes are loaded from the basic archive *classes.jsa*, which has been supplied with the JDK since Java 12. The shared archive of a simple "Hello World" program can thus be reduced from 9 MB to 256 KB.

A shared archive for a "Hello World" application compiled to *target/helloworld.jar* with a main method in the class *eu.happycoders.helloworld.Main* can be created as follows, for example:

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

To start the program using the shared archive, start it as follows:

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

## Reimplement the Legacy Socket API – JEP 353

The *java.net.Socket* and *java.net.ServerSocket* APIs have existed since Java 1.0. The underlying code, a mixture of Java and C code, is difficult to maintain and extend – particularly with regard to [Project Loom](#), which will introduce virtual threads in Java 21.

In Java 13, the old implementation is replaced by a more modern, more maintainable, and extendable implementation, which should be able to be adapted to virtual threads without further refactoring.

## Buffer.slice()

You can use the new *Buffer.slice(int index, int length)* method to create a view of a subsection of a buffer (e.g. *ByteBuffer*, *IntBuffer*, *DoubleBuffer*) that starts at position *index* and contains *length* bytes.

## New XxxBuffer.get() and put() Methods

Similarly, there are two new *get()* and *put()* methods in each of the concrete buffer classes *ByteBuffer*, *IntBuffer*, *DoubleBuffer*, etc., with which the data is not read/written at the current position of the buffer as before, but at the explicitly specified position. The following is an example of the new methods in the concrete class *ByteBuffer*:

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

The position of the buffer remains unchanged with all four methods.

## FileSystems.newFileSystem()

You can use the *FileSystems.newFileSystem(Path path, ClassLoader loader)* method to create a pseudo file system whose content is mapped to a file (such as a ZIP or JAR file).

The method was overloaded in Java 13 with a variant that makes it possible to pass a provider-specific configuration of the file system via the *env* parameter: *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 is to be loaded via the specified class loader. This is not necessary for common file types such as ZIP or JAR.



# The Latest Features in Java 12

You can find more details and the latest version of this PDF at [www.happycoders.eu/java-versions](http://www.happycoders.eu/java-versions).

The preview features published in Java 12 – Switch Expressions (JEP 325) and Shenandoah: A Low-Pause-Time Garbage Collector (JEP 189) – will be finalized in Java 14 and Java 15.

## New String and Files Methods

In Java 12, the following methods have been added to the *String* and *Files* classes:

- *String.indent(int n)* – indents the string on which *indent(...)* is called line by line by *n* spaces.
- *String.transform(Function<...> f)* – calls the function *f* with this string as a parameter and returns the result.
- *Files.mismatch()* – indicates the position of the first byte in which the files differ, or -1 for identical files.

## Teeing Collector

The new "Teeing Collector" allows you to specify two stream collectors and a merge function that combines the results of the collectors. The following example determines the difference between the maximum and minimum from the stream elements:

```
long width = numbersStream.collect(
    Collectors.teeing(Collectors.minBy(Integer::compareTo), Collectors.maxBy(Integer::compareTo),
        (min, max) -> (long) max.orElseThrow() - min.orElseThrow()));
```

## Support for Compact Number Formatting

Using *NumberFormat.getCompactNumberInstance(...)*, we can create a formatter for so-called "compact number formatting". This is a human-readable form, such as "1 thousand" or "3 billion". Here is an example:

```
NumberFormat nfShort = NumberFormat.getCompactNumberInstance(Locale.US, NumberFormat.Style.SHORT);
NumberFormat nfLong  = NumberFormat.getCompactNumberInstance(Locale.US, NumberFormat.Style.LONG);
System.out.println(nfShort.format(1_000));           // → 1K
System.out.println(nfShort.format(3_456_789_000L));  // → 3B
System.out.println(nfLong.format(1_000));            // → 1 thousand
System.out.println(nfLong.format(3_456_789_000L));  // → 3 billion
```

## Default CDS Archives – JEP 341

To activate *class-data sharing* (see Java 11), you previously had to execute the *java -Xshare:dump* command for each Java installation to generate the shared archive file *classes.jsa*.

As of Java 12, the JDK is delivered with this file on all 64-bit architectures, so that the execution of the above command is no longer necessary, and Java applications use the default CDS archive by default.

## Abortable Mixed Collections for G1 – JEP 344

One of the goals of the G1 Garbage Collector is to adhere to specified maximum pause times for "stop-the-world" phases – i.e., not to stop the application for longer than the specified time. G1 uses a heuristic for this, which can lead to the application being paused for longer than intended, especially if the application's behavior changes.

In Java 12, "mixed collections" (i.e., cleanup work that affects both the young and the old generation) were divided into a mandatory and an optional part. The mandatory part is executed uninterruptibly – and the optional part in incremental steps until the maximum pause time is reached. Meanwhile, G1 tries to adapt the heuristic so that it again determines collection sets that can be fully processed in the maximum pause time.

## Promptly Return Unused Committed Memory from G1 – JEP 346

In cloud environments where you pay for the memory you actually use, it is desirable for the garbage collector to quickly return unused memory to the operating system. The G1 Garbage Collector can return memory, but only during the garbage collection phase. However, if the heap allocation or the current rate of object allocation does not trigger a garbage collection cycle, no memory is returned.

As of Java 12, you can use the VM option *-XX:G1PeriodicGCInterval=<milliseconds>* to specify an interval at which GC cycles are to be executed, regardless of whether they are necessary or not.

## Other Changes in Java 12

**Microbenchmark Suite (JEP 230)** – Microbenchmarks, which regularly measure the performance of the JDK class library, were previously managed as a separate project. In Java 12, the code for these benchmarks was moved to the JDK source code.

**JVM Constants API (JEP 334)** – The constant pool of a *.class* file stores strings and the names of referenced classes and methods, among other things. The JVM Constants API is intended to make it easier to write Java programs that manipulate JVM bytecode. To this end, the API provides new interfaces and classes with which the constant pool can be managed.

**One AArch64 Port, Not Two (JEP 340)** – There are two ports for 64-bit ARM CPUs in the JDK: "arm64" from Oracle and "aarch64" from Red Hat. In Java 12, Oracle's port is removed in order to consolidate development resources.

# The Latest Features in Java 11 (Part 1)

You can find more details and the latest version of this PDF at [www.happycoders.eu/java-versions](http://www.happycoders.eu/java-versions).

## Launch Single-File Source-Code Programs – JEP 330

One of the most useful enhancements in Java 11 is that Java programs consisting of only a single file can be started without explicit compilation. Save the program on the right in the file *Hello.java*. You can then run it as follows without having to compile it first with *javac*:

```
$ java Hello.java
```

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

## Local-Variable Syntax for Lambda Parameters – JEP 323

A popular example for the use of a lambda expression is:

```
map.computeIfAbsent(key, (String k) -> new ArrayList<>()).add(value);
```

Java can derive the type of the lambda parameter *k* from the context, so you can omit it:

```
map.computeIfAbsent(key, k -> new ArrayList<>()).add(value);
```

As of Java 11, you can also specify *var* instead of a concrete type:

```
map.computeIfAbsent(key, (var k) -> new ArrayList<>()).add(value);
```

What is the point of specifying *var* if you can do without it? The reason for this is that only concrete types and *var* may be annotated, e.g., as follows:

```
map.computeIfAbsent(key, (@NotNull String k) -> new ArrayList<>()).add(value);
```

```
map.computeIfAbsent(key, (@NotNull var k) -> new ArrayList<>()).add(value);
```

We can replace a particularly long, annotated type name with *var* – but we must not omit it.

## HTTP Client (Standard) – JEP 321

With the new *HttpClient* class, Java 11 enables more convenient access to HTTP APIs, e.g., as follows:

```
try (HttpClient client = HttpClient.newHttpClient()) {
    HttpRequest request = HttpRequest.newBuilder().uri(URI.create(url)).GET().build();
    HttpResponse<String> response = client.send(request, BodyHandlers.ofString());
    if (response.statusCode() == 200) { String responseBody = response.body(); . . . }
    else throw new IOException("HTTP error " + response.statusCode());
}
```

## New Collection.toArray() Method

To convert a collection into an array, we previously had to pass an array of the target type to the *toArray(...)* method. If this array is sufficiently large, it is filled with the values of the collection; otherwise a new array of the same type is created. Since Java 11, we can also pass a method reference to the array's constructor to the method, e.g., like this:

```
String[] stringArray = stringList.toArray(String[]::new);
```

## New String Methods

The *String* class has been extended in Java 11 to include the following methods:

- *String.strip()*, *stripLeading()*, *stripTrailing()* – remove all leading and trailing or only the leading or only the trailing whitespaces from a string. In contrast to *trim()*, which only removes characters whose code point is U+0020 or less, *strip()* removes all characters marked as "whitespace" in the Unicode character set.
- *String.isBlank()* – returns *true* if a string consists only of characters marked as "whitespace".
- *String.repeat(int count)* – creates a new string by repeating the string on which the method is called *count* times.
- *String.lines()* – splits a string at all line breaks and returns a stream of all lines.

## Files.readString() and writeString()

Previously, several lines of Java code (or a third-party library) were required to read a file into a string or write a string into a file. As of Java 11, you can use the *Files.writeString(...)* and *Files.readString(...)* methods.

## Path.of()

As of Java 11, you can create a *Path* object using the static default method *of(...)* of the *Path* interface, e.g., like this:

```
Path.of("foo/bar/baz");           // ← Relative path foo/bar/baz
Path.of("foo", "bar", "baz");     // ← Relative path foo/bar/baz
Path.of("/", "foo", "bar", "baz"); // ← Absolute path /foo/bar/baz
```

# The Latest Features in Java 11 (Part 2)

You can find more details and the latest version of this PDF at [www.happycoders.eu/java-versions](http://www.happycoders.eu/java-versions).

## Remove the Java EE and CORBA Modules – JEP 320 ★

The following modules were removed from the JDK in Java 11: *java.xml.ws* (JAX-WS), *java.xml.bind* (JAXB), *java.activation* (JAF), *java.xml.ws.annotation* (common annotations), *java.corba* (CORBA), *java.transaction* (JTA), *java.se.ee* (aggregator module for the six modules mentioned before), *jdk.xml.ws* (tools for JAX-WS), and *jdk.xml.bind* (tools for JAXB).

The technologies mentioned were originally developed for the Java EE platform and were integrated into the Standard Edition "Java SE" in Java 6. In Java 9, they were marked as "deprecated", and in Java 11, they were removed for good.

Should you miss these libraries when upgrading to Java 11, you can pull them back into your project using Maven, for example.

## Flight Recorder – JEP 328 ★

[Java Flight Recorder](#) (JFR for short) can record numerous JVM data during the execution of an application and make it available in a file for subsequent analysis by [JDK Mission Control](#). JFR has long existed as a commercial feature in the Oracle JDK. From Java 11 onwards, it is part of the OpenJDK and can be freely used.

## Epsilon: A No-Op Garbage Collector – JEP 318 ★

Like any other garbage collector, the Epsilon garbage collector manages the allocation of objects. However, unlike all other GCs, it never releases the objects again. This can be helpful for performance tests, extremely short-lived applications, and the full elimination of latencies. Epsilon GC is activated via the VM option `-XX:+UseEpsilonGC`.

## Nest-Based Access Control – JEP 181

The Java Language Specification (JLS) has always allowed access to private fields and methods of inner classes. As the Java Virtual Machine (JVM) did *not* previously allow this, the Java compiler (up to Java 10) inserted so-called "synthetic access methods" with the standard visibility "package-private" when accessing these private fields and methods. As a result, these fields and methods could then be accessed from the entire package. The compiler issued a warning accordingly.

In JDK 11, access control of the JVM was extended so that access to private members of inner classes is possible without those synthetic accessors.

## Low-Overhead Heap Profiling – JEP 331

Heap dumps are an important debugging tool for analyzing the objects located on the heap. Until now, however, such tools have not been able to determine where in the program code such an object was created.

In Java 11, the Java Virtual Machine Tool Interface (JVMTI) – the interface through which the tools obtain data about the running application – has been expanded to include the option of collecting stack traces of all object allocations. Heap analysis tools can display this additional information, making it much easier for us to analyze problems.

## Other Changes in Java 11

**Dynamic Class-File Constants (JEP 309)** – The *.class* file format is extended by the *CONSTANT\_Dynamic* constant, "which gives language developers and compiler implementers broader options for expressiveness and performance."

**Improve AArch64 Intrinsics (JEP 315)** – Intrinsics are used to execute architecture-specific assembler code instead of Java code. Intrinsics for the AArch64 platform have been improved, and new intrinsics have been added.

**Key Agreement with Curve25519 and Curve448 (JEP 324)** – Key exchange protocols with elliptical curves, such as "Curve25519" and "Curve448", enable particularly fast encryption and decryption of symmetric keys.

**Unicode 10 (JEP 327)** – Unicode support has been upgraded to version 10.0, i.e., classes such as *String* and *Character* can handle the newly introduced code blocks and characters.

**ChaCha20 and Poly1305 Cryptographic Algorithms (JEP 329)** – The JDK is extended by the two cryptographic algorithms mentioned above. These are used, for example, by security protocols such as TLS.

**Transport Layer Security (TLS) 1.3 (JEP 332)** – In addition to Secure Socket Layer (SSL) 3.0, Transport Layer Security (TLS) 1.0 to 1.2, and Datagram Transport Layer Security (DTLS) 1.0 and 1.2, Java also supports TLS 1.3 from version 11 onwards.

**ZGC: A Scalable Low-Latency Garbage Collector (Experimental) (JEP 333)** – A new GC that will be finalized in Java 15.

**Deprecate the Nashorn JavaScript Engine (JEP 335)** – The JavaScript engine "Nashorn" introduced in Java 8 is marked as "deprecated for removal" in Java 11. The reason for this is the rapid development speed of ECMAScript (the standard behind JavaScript) and the *node.js* engine, which makes further development of Nashorn too costly.

**Deprecate the Pack200 Tools and API (JEP 336)** – The "Pack200" compression algorithm was introduced in Java 5 to save as much bandwidth as possible in the early 2000s through more effective compression. In times of 100 MBit lines, the development costs are no longer in proportion to the benefits. Pack200 is therefore marked as "deprecated for removal".

# The Latest Features in Java 10

You can find more details and the latest version of this PDF at [www.happycoders.eu/java-versions](http://www.happycoders.eu/java-versions).

## Local-Variable Type Inference – JEP 268 🌟

Since Java 10, we can use the *var* keyword to declare local variables ("local" means: within methods), as in the example on the right.

The extent to which *var* is used will probably lead to heated discussions. I use it when a) it is significantly shorter and b) I can clearly recognize the type in the code.

```
var i = 10;
var hello = "Hello world!";
var list = List.of(1, 2, 3, 4, 5);
var httpClient = HttpClient.newBuilder().build();
var status = getStatus();
var map = new HashMap<String, String>();
```

## Immutable Collections 🌟

From Java 10 onwards, you can use *List.copyOf(...)*, *Set.copyOf(...)*, and *Map.copyOf(...)* to create immutable copies of lists, sets, and maps. Attempting to modify collections created in this way results in an *UnsupportedOperationException*.

Similarly, you can use *Collectors.toUnmodifiableList()*, *toUnmodifiableSet()*, and *toUnmodifiableMap()* to collect streams in immutable lists, sets, and maps.

## Optional.orElseThrow() 🌟

The *Optional.get()* method throws a *NoSuchElementException* if an *Optional* is empty; it should therefore always be used in combination with *Optional.isPresent()*. IDEs and static code analysis tools generate a warning if you don't do this. However, if such an exception is desired, this warning is annoying.

This is why the method *Optional.orElseThrow()* was introduced, which does exactly the same as *Optional.get()*, but is not reported by static code analysis tools, as its name clearly describes its behavior.

## Parallel Full GC for G1 – JEP 307

The Garbage-First (G1) garbage collector has replaced the parallel collector as the standard garbage collector in Java 9. While the parallel GC could perform a complete garbage collection ("full GC") in parallel to the running application, this was so far not possible with G1 – it had to stop the application ("stop-the-world"), which could result in noticeable latencies.

In Java 10, full garbage collection of the G1 Garbage Collector has now also been parallelized.

## Application Class-Data Sharing – JEP 310

*Class-Data Sharing* (CDS for short) means that once loaded, *.class* files are cached in an architecture-specific binary form so that the JVM start is accelerated, and the total memory requirement is reduced by sharing access to this cache from several JVMs running in parallel.

In Java 10, CDS was extended so that not only classes of the JDK class library can be cached, but also the classes of the application ("Application CDS", "AppCDS" for short).

## Other Changes in Java 10

**Consolidate the JDK Forest into a Single Repository (JEP 296)** – The JDK source, which was previously stored in eight separate repositories, has been consolidated into a [monorepo](#), which greatly simplifies work through atomic commits, branches, and PRs.

**Garbage-Collector Interface (JEP 304)** – A clean garbage collector interface was introduced in the JDK source code to isolate the GC algorithms from the interpreter and the compilers and to simplify the development of new GCs.

**Thread-Local Handshakes (JEP 312)** – Thread-local handshakes are an optimization to improve virtual machine performance on x64 and SPARC-based architectures.

**Remove the Native-Header Generation Tool (javah) (JEP 313)** – The *javah* tool was used to create native header files for JNI. The functionality was integrated into the Java compiler *javac* in Java 10.

**Additional Unicode Language-Tag Extensions (JEP 314)** – So-called "language tag extensions" allow additional information about the currency, the first day of the week, the region, and the time zone to be stored in a *Locale* object.

**Heap Allocation on Alternative Memory Devices (JEP 316)** – As of Java 10, the Java heap can also be created on an alternative memory device, e.g., on NV-DIMM (non-volatile memory), instead of on conventional RAM.

**Experimental Java-Based JIT Compiler (JEP 317)** – Since Java 9, the [Graal compiler](#) (a Java compiler written in Java) has been included as an experimental ahead-of-time (AOT) compiler. From Java 10 onwards, Graal can also be used as a just-in-time (JIT) compiler – initially on the Linux/x64 platform.

**Root Certificates (JEP 319)** – Root certificates previously only included in the Oracle JDK are now available in the OpenJDK, too.

**Time-Based Release Versioning (JEP 322)** – Version information now includes the release date and whether the version is a long-term support release. Examples: *java version "10.0.2" 2018-07-17*, *java version "11.0.11" 2021-04-20 LTS*