Java 18

***UTF-8 by Default*** *- UTF-8 is a variable-width character encoding widely used on the web for electronic communication. In Java 18, the default charset of the platform is now UTF-8. This is a change from Java’s previous default charset.* *The main reason for this update is that UTF-8 has become the most widely used charset in the world, so making it an explicit choice in Java simplifies things quite a bit.*

***JEP 408: Simple Web Server*** *- This JEP provides a command-line tool, jwebserver to start a simple web server to serve static files, suitable for prototyping, ad-hoc coding, testing, and educational purpose, not for a production server.*

*By default, the jwebserver starts a simple web server at port 8000, and serves the static files from the current directory that start the command. Furthermore, it serves only HEAD and GET requests, and other requests will receive a 501 – Not Implemented or a 405 – Not Allowed response.*

*Command to starts a static web server at port 8888. - ./jwebserver -p 8888*

*Command starts a static web server at port 8888 and serves files from the directory - ./jwebserver -p 8888 -d /Users/mkyong*

*Code Snippets in Java API Documentation - Before Java 18, we used {@code ...} to include the source code snippets in the documentation like this:*

*/\*\**

*\* <pre>{@code*

*\* int sum = widgets.stream()*

*\* .filter(w -> w.getColor() == RED)*

*\* .mapToInt(w -> w.getWeight())*

*\* .sum();*

*\* }</pre>*

*\*/*

*The javadoc tool will render the body of the {@code ...} tag as HTML code. This method has some disadvantages, like the inability to do syntax highlighting, can’t to contain HTML markups, indentation issues, etc.*

*This JEP introduces the @snippet tag to allow developers to be more flexible in including source code snippets in the documentation.*

*Inline snippets*

*The generated documentation will render the body of the {@snippet ...} tag as HTML code. There is no need to escape special characters like ’ <’ and ‘>` with HTML entities.*

***Reimplement Core Reflection with Method Handles -*** *The existing core reflection has three internal mechanisms for invoking methods and constructors. Adding new features to the core reflection may modify all three code paths. Which is costly.*

* *VM native methods*
* *Dynamically generated bytecode stubs*
* *Method handles*

*This JEP reimplements core reflection with method handles as the underlying reflective mechanism. There is no change to the java.lang.reflect API; solely an implementation change.*

*We can enable back the old implementation via -Djdk.reflect.useDirectMethodHandle=false*

***Vector API -*** *Java coders can use the new Vector API in Java 18 to perform vector computations.*

***Internet-Address Resolution SPI*** *- By default, the java.net.InetAddress API uses the operating system’s built-in resolver to resolve host names to Internet Protocol (IP) addresses.*

*InetAddress ip = InetAddress.getByName("google.com");*

*This JEP redesign java.net.InetAddress API to use service loader to find the resolver instead of using the operating system’s built-in resolver.*

*InetAddress.java*

*private static InetAddressResolver loadResolver() {*

*return ServiceLoader.load(InetAddressResolverProvider.class)*

*.findFirst()*

*.map(nsp -> nsp.get(builtinConfiguration()))*

*.orElse(BUILTIN\_RESOLVER);*

*}*

***Foreign Function & Memory API (Second Incubator)*** *- This Foreign Function & Memory API allows the developer to access the code outside the JVM (foreign functions), data stored outside the JVM (off-heap data), and accessing memory not managed by the JVM (foreign memory). This JEP improves the Foreign Function & Memory API and other enhancements in response to feedback.*

***Pattern Matching for switch (Second Preview) -*** *This JEP is the second preview of pattern matching for the switch, with the following enhancements since the first preview:*

*8.1. Dominance checking of the same type.*

*Review the below switch pattern matching, every value that matches the String s also matches the CharSequence cs, which makes String s unreadable and will cause a compile-time error.*

static void error (Object o){

switch (o) {

case CharSequence cs -> System.out.println("A sequence of length " + cs.length());

case String s -> *// Error - pattern is dominated by previous pattern*

System.out.println("A string: " + s);

default -> {

break;

}

}

}

*If we run the above code with Java 18, it will cause a compile-time error.*

java: this case label is dominated by a preceding case label

*The IntelliJ IDE will highlight the error label and show a more user-friendly error message when you hover over it.*

Label is dominated by a preceding case label 'CharSequence cs'

*8.2 Exhaustiveness of switch expressions and statements*

*The switch expression requires all possible values to be handled in the switch block, else prompts a compile-time error.*

*Review the below code:*

static int coverage(Object o) {

return switch (o) { *// Error - not exhaustive*

case String s -> s.length();

case Integer i -> i;

};

}

*If we run the above code with Java 18, it will cause a compile-time error.*

*java: the switch expression does not cover all possible input values*

*The below code is fine because the default will handle all the possible types.*

static int coverage(Object o) {

return switch (o) {

case String s -> s.length();

case Integer i -> i;

default -> 0;

};

}

***Deprecate Finalization for Removal -*** *The object finalization is unpredictable and dangerous. The JDK has deprecated finalizers for some time now, but in JDK 18 the deprecation will become a warning. This move signals that finalizers might be removed from Java SE in a future release. This JEP deprecates (again?) finalization for removal in the future release. In Java 18, we can use the command-line option --finalization=disabled to disable finalization. The finalization remains enabled by default;*