Java 11 Features

**Date 22/03/2023**

# Overview of Java 11 (Sept 25, 2018)

1. Collections toArray (int Function)
2. Local Variable update for Lambda expression
3. Single command Java File run
4. String API Updates
5. Optional Enhancement isEmpty
6. File Read and Write Updates
7. HTTP Client API Update
8. Nested Based Access Control

# JEP 323 - Local-Variable Syntax for Lambda Parameters

The var is a keyword in Java that is used to declare a local variable. It was introduced in Java 10 to improve type inference in Java.

In Java 11, we can use var with lambda expression parameters to avoid using the type name with the variable name. In earlier versions of Java, while working with the lambda expression, the var keyword was not allowed.

## Example

public class **LocalVarInLambda\_Demo1** {

public static void main (String[] args) {

calc c = (var x, var y) -> {

return (x + y);

};

int sum = c.add(2, 3);

System.out.println(sum);

}

}

interface **Calc** {

public int add (int a, int b);

}

# String API Enhancements

In Java 11, the String class has been extended with some helpful methods.

## String.repeat(int)

Repeats a string given number of times. Returns the concatenated string. The repeat method simply repeats the string that many numbers of times as mentioned in the method in the form of an int.

## String.isBlank()

Checks if a string is empty or have white spaces only. It returns a boolean value. Empty Strings and Strings with white spaces are only treated as blank

## String.strip()

Removes the leading and trailing whitespaces.

## String.stripLeading()

Removes the leading whitespaces. It is used to remove the white space at beginning of string.

## String.stripTrailing()

Removes the trailing whitespaces. It is used to remove the white space at end of the string.

## String.lines()

Return the stream of lines of multi-line string. This method returns a stream of strings, which is a collection of all substrings split by lines.

## Trim () vs Strip ()

Trim () removes all characters with a code point U+0020 or smaller. This includes, for example, "space", "tab", "newline", and "carriage return".

Strip () removes those characters that Character.isWhitespace() classifies as whitespaces. On the one hand, these are some (but not all) characters with code point U+0020 or smaller. And on the other hand, characters defined in the Unicode Standard as spaces, line breaks, and paragraph separators (e.g., U+2002 - a space as wide as the letter 'n').

In short: strip() is "Unicode-aware" evolution of trim(). Meaning trim() removes only characters <= U+0020 (space); strip() removes all Unicode whitespace characters (but not all control characters, such as \0)

The definition of space used by String::trim is any code point less than or equal to the space code point (\u0020), commonly referred to as ASCII or ISO control characters.

Unicode-aware trimming routines should use Character::isWhitespace(int).

Additionally, developers have not been able to specifically remove indentation white space or to specifically remove trailing white space.

Introduce trimming methods that are Unicode white space aware and provide additional control of leading only or trailing only.

The current Javadoc for String::trim does not make it clear which definition of "space" is being used in the code. With additional trimming methods coming in the near future that use a different definition of space, clarification is imperative. String::trim uses the definition of space as any codepoint that is less than or equal to the space character codepoint (\u0020.) Newer trimming methods will use the definition of (white) space as any codepoint that returns true when passed to the Character::isWhitespace predicate.

## Reference

<https://stackoverflow.com/questions/51266582/difference-between-string-trim-and-strip-methods-in-java-11>

<https://stackoverflow.com/questions/53640184/why-is-string-strip-5-times-faster-than-string-trim-for-blank-string-in-java>

## Example

public class **String \_Demo1** {

public static void main(String[] args) {

var topic = "Java String Repeat ";

System.out.println(topic.repeat(3));

var whiteSpace = " ";

var blank = "";

var topic = "String";

System.out.println(whiteSpace.isBlank());

System.out.println(blank.isBlank());

System.out.println(topic.isBlank());

var stripDemo = " strip ";

System.out.println("Before Strip Operation");

System.out.print("|");

System.out.print(stripDemo);

System.out.println("|");

System.out.println("After Strip Operation");

System.out.print("|");

System.out.print(stripDemo.strip());

System.out.print(stripDemo. stripLeading ());

System.out.print(stripDemo. stripTrailing() ());

System.out.println("|");

**// With lines()**

var lines1 = linesDemo.lines();

lines1.forEachOrdered(System.out::println);

var lines2 = linesDemo.lines().collect(Collectors.toList());

System.out.println(lines2);

}

}

# File API Enhancements

Reading and writing text files has been continuously simplified since Java 6. In Java 6, we had to open a FileInputStream, wrap it with an InputStreamReader and a BufferedReader, then load the text file line by line into a StringBuilder (alternatively, omit the BufferedReader and read the data in char [] blocks) and close the readers and the InputStream in the finally block.

In Java 7, we could create the nested stream/reader or stream/writer combinations much easier using Files.newBufferedWriter() and Files.newBufferedReader(). And With the help of try-with-resources, we didn't need a finally block anymore.

**Read and Write File in Java 7**

public static void **writeStringJava7**(Path path, String text) throws IOException {

try (BufferedWriter writer = Files.newBufferedWriter(path, StandardCharsets.UTF\_8)) {

writer.write(text);

}

}

private static String **readFileJava7**(Path path) throws IOException {

StringBuilder sb = new StringBuilder();

try (BufferedReader reader = Files.newBufferedReader(path, StandardCharsets.UTF\_8)) {

String line;

while ((line = reader.readLine()) != null) {

sb.append(line).append('n');

}

}

return sb.toString();

}

Java 11 has introduced an easy way to read and write files by providing new overloaded methods without writing much boiler plate code.

## WriteString()

This method is used to write some content in a file. Using this method, we can insert some text into file at the location.

## ReadString()

This method is used to read the contents of a file.

## Example

public class **File\_ReadWrite\_Demo1** {

public static void main(String[] args) throws IOException {

Files.writeString(Path.of("Java11File.txt"), "This is Java11 File", StandardCharsets.UTF\_8,

StandardOpenOption.WRITE);

String fileContent = Files.readString(Path.of("Java11File.txt"));

System.out.println(fileContent);

}

}

# Collection Enhancements

Until Java 11, the Collection interface provided two toArray() methods to convert collections to arrays. The following example shows these two methods (and two different usages of the second method) using a String list as an example:

**List<String> list = List.of("foo", "bar", "baz");**

**Object[] strings1 = list.toArray();**

**String[] strings2a = list.toArray(new String[list.size()]);**

**String[] strings2b = list.toArray(new String[0]);**

The first toArray() method (without parameters) returns an Object array, because due to Type Erasure, the type information of list is no longer known at runtime.

The second toArray() method expects an array of the requested type. If this array is at least as large as the collection, the elements are stored in this array (strings2a). Otherwise, a new array of the needed size is created (strings2b).

**String[] strings = list.toArray(String[]::new);**

The above method allows the Collection classes to create an array of the necessary size using the passed array constructor reference.

## Example

public class **ToArray\_Demo1** {

public static void main(String[] args) {

// List Type 1

var list1 = List.of("Java", "JavaScript");

var list1Array = list1.toArray(String[]::new);

for (var data : list1Array) {

System.out.println(data);

}

}

}

# Optional Enhancements

Optional is a container object which may or may not contain a non-null value. If no value is present, the object is considered empty. Previously existing method isPresent() returns true if a value is present, otherwise false. Sometimes, it forces us to write negative conditions which are not readable.

Java 11 introduced new method to Optional class as isEmpty() to check if value is present. isEmpty() returns false if value is present otherwise true. It can be used as an alternative of isPresent() method which often needs to negate to check if value is not present.

## Example

public class **Optional\_IsEmpty\_Demo1** {

public static void main(String[] args) {

String name = null;

System.out.println(Optional.ofNullable(name).isEmpty());

System.out.println(!Optional.ofNullable(name).isPresent());

var topic = "Java";

System.out.println(Optional.ofNullable(topic).isEmpty());

System.out.println(!Optional.ofNullable(topic).isPresent());

Optional<?> string = Optional.empty();

System.out.println(string.isEmpty());

}

}

# Stream API Enhancements

## Predicate Not

Java 11 introduced a new static method to Predicate interface as not () to negate an existing predicate similar to negate method.

## Example

While not(isBlank) reads more naturally than isBlank.negate(), the big advantage is that we can also use not with method references, like not(String::isBlank).

public class **Predicate\_Not\_Demo1** {

public static void main(String[] args) {

var sampleList = Arrays.asList("java", "\n", "javascript", " ");

var listWithoutBlankValue = sampleList.stream().filter(Predicate.not(String::isBlank))

.collect(Collectors.toList());

System.out.println(listWithoutBlankValue);

}

}

## Pattern Recognition

The asMatchPredicate() method will create a predicate if the pattern matches with the input string. It returns true if the string matches else returns false.

Example,

public class **PatternMatchAsPredicate\_Demo1** {

public static void main(String[] args) {

var pattern = Pattern.compile("^[a-zA-Z0-9+\_.-]+@[a-zA-Z0-9.-]+$").asMatchPredicate();

System.out.println(pattern.test("john.alex@gmail.com"));

}

}

# JEP 181 - Nested Based Access Control

Java 11 introduced a concept of nested class where we can declare a class within a class. This nesting of classes allows to logically group the classes to be used in one place, making them more readable and maintainable. Nested class can be of four types.

1. Static nested classes
2. Non-static nested classes
3. Local classes
4. Anonymous classes

Java 11 also provide the concept of nestmate to allow communication and verification of nested classes.

## Use Case 1

The Java Language Specification (JLS) allows access to private fields and methods of inner classes. The Java Virtual Machine (JVM), on the other hand, does not (yet) allow this.

To resolve this contradiction, the Java compiler (up to Java 10) inserts so-called "synthetic accessor methods" when accessing these private fields and methods – with default "package-private" visibility.

These additional methods result in seemingly private fields and methods being accessible from the entire package. Accordingly, the warning occurs.

Until now, you could solve this problem by either making the corresponding members package-private yourself or - at least in Eclipse - annotating the code with @SuppressWarnings("synthetic-access").

JEP 181 extends the JVM's access control mechanisms to allow access to private members of inner classes without synthetic accessors.

Should you have made methods and fields of inner classes package-private or used @SuppressWarnings for the above reason, you can undo this after upgrading to Java 11.

## Use Case 2

Java 11 introduced nest-based access control that allows classes to access each other’s private members without the need for bridge methods created by the compiler. These methods are called accessibility-broadening bridge methods and the compiler inserts these into the code during the program execution.

Before Java 11, if we have private members in our code then the compiler creates accessibility-broadening bridge methods that increase the size of the deployed applications and may lead to confusion. That’s why Java improved nest-based access control.

Java 11 allows classes and interfaces to be nested within each other. This nested type can be private fields, methods, and constructors.

Example,

public class **NestBasedAccessControl\_Demo1** {

public static void main(String[] args) {

var isNestMate = \_01\_NestBasedAccessControl\_Demo1.class

.isNestmateOf(\_01\_NestBasedAccessControl\_Demo1.InnerClass.class);

var isNestHost = \_01\_NestBasedAccessControl\_Demo1.InnerClass.class

.getNestHost() == \_01\_NestBasedAccessControl\_Demo1.class;

System.out.println("NestMate ==> " + isNestMate);

System.out.println("NestHost ==> " + isNestHost);

Set<String> nestedMembers = Arrays.stream(\_01\_NestBasedAccessControl\_Demo1.InnerClass.class.getNestMembers())

.map(Class::getName).collect(Collectors.toSet());

System.out.println(nestedMembers);

}

public class InnerClass {

}

}

# JEP 321 - HttpClient Enhancement

The new HTTP client from the java.net.http package was introduced in Java 9. It has now become a standard feature in Java 11. JDK 11 includes the new HttpClient class, which significantly simplifies working with HTTP.

HttpClient also supports HTTP/2 and WebSocket, unlike the previous solution.

Furthermore, in addition to the synchronous programming model shown above, HttpClient provides an asynchronous model. The HttpClient.sendAsync() method returns a CompletableFuture, which we can then use to continue working asynchronously.

## Reference

<https://openjdk.org/groups/net/httpclient/>

## Example

public class **Http2Client\_Demo1** {

public static void main(String[] args) {

HttpClient httpClient = HttpClient.newBuilder().connectTimeout(Duration.ofSeconds(10))

.version(HttpClient.Version.HTTP\_2).build();

HttpResponse<String> response = null;

try {

String urlEndPoint = "https://reqres.in/api/users";

URI uri = URI.create(urlEndPoint + "?id=2");

HttpRequest request = HttpRequest.newBuilder().uri(uri).build();

response = httpClient.send(request, HttpResponse.BodyHandlers.ofString());

} catch (IOException ex) {

ex.printStackTrace();

} catch (InterruptedException ex) {

ex.printStackTrace();

}

System.out.println("Status Code ===> " + response.statusCode());

System.out.println("Headers ===> " + response.headers().allValues("content-type"));

System.out.println("Body ===> " + response.body());

}

}

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

With JDK 11, we got a new garbage collector: Epsilon GC. It manages the allocation of objects on the heap - but it has no garbage collection process to release the objects again. A new garbage collector called Epsilon is available for use in Java 11 as an experimental feature.

It's called a No-Op (no operations) because it allocates memory but does not actually collect any garbage. Thus, Epsilon is applicable for simulating out of memory errors.

What is the purpose of a garbage collector that does not collect garbage?

The following scenarios are convincible:

**Performance tests:** In micro benchmarks, for example, where you compare different implementations of algorithms with each other, a regular garbage collector is a hindrance, as it can influence the execution times and thus falsify the measurement results. By using Epsilon GC, you can exclude such influences.

**Extremely short-lived applications:** The applications such as those developed for AWS Lambda, should be terminated as quickly as possible. A garbage collection cycle would be a waste of time if the application was terminated a few milliseconds later anyway.

**Eliminating latencies:** If developers have a good understanding of the memory requirements of their application and entirely (or almost entirely) dispense with object allocations, Epsilon GC enables them to implement a latency-free application.

Now Epsilon is good only for test environments. It will lead to OutOfMemoryError in production and crash the applications. The benefit of Epsilon is no memory clearance overhead. Hence, it will give an accurate test result of performance and we can no longer GC for stopping it. Note: This is an experimental feature.

You can activate Epsilon GC with the following option in the java command line:

**-XX:+UseEpsilonGC**

**-XX:+UnlockExperimentalVMOptions -XX:+UseEpsilonGC flag.**

# JEP 330 - Single File Source Code Program

Java 11 onwards, now a single java file can be tested easily without compiling as well. In Earlier versions of java we have to compile java file first and then we can execute code but from java11 we can directly execute code without compiling it. Suppose if we have Sample.java file we can directly run java Sample.java for execution of code in JAVA11 without compiling it.

# JEP 328 - Java Flight Recorder

Numerous tools help us analyze and fix errors during the development process. However, certain problems only occur at the runtime of an application. Analyzing them is often difficult or impossible, as we are often unable to reproduce such errors.

Java Flight Recorder (JFR) can assist us by recording JVM data at runtime and making it available in a file for subsequent analysis.

Flight Recorder has already existed for several years as a commercial feature in Oracle's JDK. With JDK Enhancement Proposal 328, it becomes part of the OpenJDK and can thus be used freely.

## To start Flight Recorder

You can start Flight Recorder in two ways. Firstly, you can activate it at the start of an application using the following option on the java command line:

**-XX:StartFlightRecording=filename=<file name>**

Secondly, you can use jcmd to activate Flight Recorder in a running Java application:

**jcmd JFR.start filename=<file name>**

To start a 120 seconds JFR recording, we can use the following parameter:

**-XX:StartFlightRecording=duration=120s,settings=profile,filename=java-demo-app.jfr**

You can specify numerous options; for example, you can use "duration" to specify how long the recorder should run. To present all options in detail would go beyond the scope of this article. You can find them in Oracle's official Flight Recorder documentation.

## Example

In the following example, let 31100 be the process ID of the Java application to be analyzed. You start the recording as follows (we specify a name for the recording via the optional "name" parameter):

$ jcmd 31100 JFR.start filename=myrecording.jfr name=myrecording

31100:

Started recording 1. No limit specified, using maxsize=250MB as default.

Use jcmd 31100 JFR.dump name=myrecording to copy recording data to file.

Code language: plaintext (plaintext)

Normally Flight Recorder saves the recording to the specified file only at certain intervals and when you exit the application. However, you can also save the recording manually in between by executing the dump command that was displayed at startup:

$ jcmd 31100 JFR.dump name=myrecording

31100:

Dumped recording "myrecording", 344.8 kB written to:

<path>/myrecording.jfr

Code language: Mizar (mizar)

If you did not specify a "name" parameter at startup, you can specify the recording number (in the example above "1") as the name.

You can stop Flight Recorder as follows:

$ jcmd 31100 JFR.stop name=myrecording

31100:

Stopped recording "myrecording".

## JDK Mission Control

To view the collected data, you need another tool: JDK Mission Control. On the project's GitHub page, you can find links to several distributors where you can download Mission Control for Windows, Mac, and Linux.

<https://github.com/openjdk/jmc>

Click "File / Open File…" to load the analysis file. Mission Control first shows you an overview of the collected data:

Using the navigation on the left, you can then dive deeper into specific areas, such as threads, memory usage, locks, etc… In "Threads", for example, you can see which threads ran from when to when.

# References

<https://lovetolearn-anil.blogspot.com/2023/02/java11-features-simply-explained.html>

<https://www.happycoders.eu/java/java-11-features/>

<https://www.baeldung.com/java-11-new-features>

<https://www.tutorialspoint.com/java11/index.htm>

<https://howtodoinjava.com/java11/features-enhancements/>

<https://www.digitalocean.com/community/tutorials/java-11-features>