Java Updates

Java 9

Module System

The Java Module System allows you to set up your JAR as a module, where that module can…

- define dependencies to other modules (other JARs that are modules).
- expose its classes to other modules (classes aren't exposed by default).

Up until Java8, nearly all core JDK classes were kept in rt.jar. The problem was that some JDK classes had to have public visibility but weren't intended to be used outside of the JDK -- they were internal JDK classes that other JDK classes (in different packages) would use. For example, sun.misc.Unsafe was an internal JDK class that was never intended for use by the outside world. Because it was public, several tools/frameworks/libraries ended up relying on it for some of their functionality/performance.

Starting with Java9, Java uses the Java Module System to organize core JDK classes and prevent access to internal classes. rt.jar has been split up into multiple separate JAR modules with dependencies between each other, and only those classes intended for use by the outside world are whitelisted/exposed.

NOTE: Taken from https://blog.codefx.org/java/dev/javaone-2015-introduction-to-modular-development/

To define a module, you need to include a module-info.java file in the default package of your JAR. In there you can define …

```
module MyJavaModule {
    // requires <module> -- depends on some other module
    // requires transitive <module> -- depends on some other module + all
    //                                 of its dependencies
    // requires static <module> -- depends on some other module at
    //                             compile-time but is optional at
    //                             runtime
    // exports <package> -- make a package visible to all other modules
    // exports <package> to <module/package> -- make package visible to
    //                                          only specific modules
    // opens <package> -- make a package visible to all other modules but
    //                    only at runtime (not compile-time)
    // opens <package> to <module/package> -- make a package visible to
    //                                        only specific modules but
    //                                        only at runtime (not
    //                                        compile-time)
    //   NOTE if you want all packages in a module to be open…
    //     open module MyJavaModule {
    //         requires ...
    //         exports ...
    //         exports ...
    //      }
    // uses <interface> -- specifies a service used by this module
    // provides <interface> with <implementation> -- provides a service
    //                                               implementation
    requires java.xml;
    exports com.offbynull.project.packageA;
    exports com.offbynull.project.packageB;
}
```

NOTE: Remember that the default package is the root of your source folder. It's where all the code goes that isn't in some explicit package.

NOTE: Just like classes, modules can be marked as deprecated and end up being removed in later versions of the JDK. In addition, new modules that are in beta may be added under the jdk.incubator namespace

To list all modules available in the JDK, run java --list-modules…

```
$ java --list-modules
java.activation@9
java.base@9
java.compiler@9
java.corba@9
...
```

To describe a module in the JDK, run java --describe-module <name>...

```
$ java --describe-module java.logging
java.logging@9
exports java.util.logging
requires java.base mandated
provides jdk.internal.logger.DefaultLoggerFinder with
sun.util.logging.internal.LoggingProviderImpl
contains sun.net.www.protocol.http.logging
contains sun.util.logging.internal
contains sun.util.logging.resources
```

Since Java9+ removes public access to internal JDK classes, you may need to forcefully allow access for legacy old code to run. You can do this by using the --add-exports flag. This flag works with both the Java compiler and Java itself...

```
$ javac --add-exports java.base/sun.misc.Unsafe=ALL-UNNAMED Main.java
$ java --add-exports java.base/sun.misc.Unsafe=ALL-UNNAMED Main
```

> **NOTE**: What is ALL-UNNAMED? By default, all exported classes in all modules are a part of the 'unnamed' module. With this flag, we're forcefully exposing/whitelisting a class that was never meant to be exposed to the outside.

But, you shouldn't do this because it'll likely be removed in the future. The best course of action is to use the new --illegal-access=deny flag to force strong encapsulation (will never expose non-exported classes, will likely be the default in the future). Once that's done, use the new jdeps tool to scan classes/JARs for usage of internal JDK classes. If it finds an internal JDK class being used, it'll attempt to provide you with a replacement class…

```
$ jdeps kryo-2.20.jar
kryo-2.20.jar -> java.base
kryo-2.20.jar -> java.desktop
kryo-2.20.jar -> not found
   com.esotericsoftware.kryo                          ->
com.esotericsoftware.kryo.io
...
```

```
JDK Internal API                          Suggested Replacement
----------------                          ---------------------
...
```

NOTE: Currently, the default for the --illegal-access flag is permit, which lets you break encapsulation.

JShell

JShel is a REPL (Read Eval Print Loop) for Java, similar to the REPL in Python. It comes as a standalone program called jshell...

```
$ jshell
|  Welcome to JShell -- Version 11.0.1
|  For an introduction type: /help intro

jshell> int a = 5
a ==> 5

jshell> String b = "hello!"
b ==> "hello!"

jshell> b+a
$3 ==> "hello!5"

jshell> $3 + "helloagain!"
$4 ==> "hello!5helloagain!"

jshell> class Test1 {
   ...>     public String value;
   ...> }
|  created class Test1
```

The syntax isn't exactly Java, but it's close. For example, …

- semicolons aren't always required (example).
- results not going into a variable may get implicitly assigned to one ($3/$4 in example).
- import statements can appear anywhere.

In addition, it provides you with some common shell functions...

- ctrl-L clears the screen.
- auto-completing for commands, method invocations, types, variables, etc…

The JShell API provides an API to access JShell functionality directly. If you wanted to, you can expose the less pedantic JShell coding style to users of your application to give them the ability to easily script functionality. See the package summary for how you'd do this: https://docs.oracle.com/javase/9/docs/api/jdk/jshell/package-summary.html.

Process API Changes

The process API has been extended to support any native process currently running on the OS via the ProcessHandle class. ProcessHandle provides methods to get details of the process (e.g. get parent processes, get child processes, get CPU usage, etc..). It also lets you perform operations on the process (e.g. destroy).

Use ProcessHandle.of() to get a ProcessHandle from a pid.
Use ProcessHandle.curent().pid() to get your own pid.
Use ProcessHandler.allProcesses() to get a snapshot of all currently running processes.
Use ProcessHandler.onExit() to get a Future that gets completed once the proc exits

Remember that Java uses the Process/ProcessBuilder class to launch new processes on the OS. Process.toHandle() has been added to get a ProcessHandle for a Process that you launched.

The ProcessHandle allows you to forceibly kill processes, but it won't let you forceibly kill your own Java process -- it'll throw an IllegalStateException.

Collection/Stream API Changes

Collection Factories

You can now create immutable Lists, Maps, and Sets using collection factory methods.For example, ...

- List.of("a", "b", "c");
- Set.of("a", "b", "c");
- Map.of("a", 0, "b", 1, "c", 0)

Note that…

- null values aren't allowed (exception thrown).
- set values and map keys must be unique (exception thrown).

Stream.takeWhile() / Stream.dropWhile()

The method...

- Stream.takeWhile(Predicate p) → passes elements forward from an ORDERED stream until p returns false, at which point no more elements are passed forward. In other words, stream elements are passed through until one of them fails p.
- Stream.dropWhile(Predicate p) → ignores elements from an ORDERED stream until p returns true, at which point it starts passing elements forward. In other words, stream elements are dropped until one of them fails p.

Note that these method should only be used on…

- ordered streams - using them on parallel streams will result in non deterministic behaviour.
- sequential streams - using them on parallel ordered streams will result in very bad performance.

See https://blog.indrek.io/articles/whats-new-in-java-9-streams/ for more information

Stream.ofNullable() / Optional.stream()

The Stream.ofNullable() method creates a single element stream, if that element is not null. If it's null then it'll be an empty stream.

Similarly, the Otional.stream() method will return a single element stream if the optional is present, or an empty stream if it's not present. This is useful for flattening out streams of optionals (options not present will be empty streams, so they'll automatically get ignored)…

```
optionalStream
    .flatMap(o -> o.stream())
    .collect(toList());
```

Stream.iterator() overload (Iterator to Stream)

A new method has been added to Stream that makes it easier to convert an iterator to a stream…

```
static Stream<T> iterate(
    T seed,
    Predicate<? super T> hasNext,
    UnaryOperator<T> next)
```

The old version of this method didn't take in the hasNext parameter -- it was for generating infinite streams. This is for generating finite streams.

Collectors.filtering()

A new method has been added to Collectors that allows you to filter items when you're collecting a stream…

```
public static <T,A,R> Collector<T,?,R> filtering(
    Predicate<? super T> predicate,
    Collector<? super T,A,R> downstream)
```

Note that this collector requires a downstream collector -- (it doesn't produce a final output by self). An example usage can be found directly in the JavaDocs for this method…

```
 Map<Department, Set<Employee>> wellPaidEmployeesByDepartment
   = employees.stream().collect(
     groupingBy(Employee::getDepartment,
                filtering(e -> e.getSalary() > 2000,
                          toSet())));
```

Collectors.flatMapping()

A new method has been added to Collectors that allows you to flatten out a stream of streams when you're collecting…

```
public static <T,U,A,R> Collector<T,?,R> flatMapping(
    Function<? super T,? extends Stream<? extends U>> mapper,
    Collector<? super U,A,R> downstream)
```

Note that this collector requires a downstream collector -- (it doesn't produce a final output by self). An example usage can be found directly in the JavaDocs for this method (modified to make it easier to read/understand)…

```
Map<String, Set<LineItem>> itemsByCustomerName
   = orders.stream().collect(
     groupingBy(Order::getCustomerName,
                flatMapping(o -> o.lineItems(), toSet())));
```

In the above example, orders is probably…

```
List<Order> orders = ...;
public static final class Order {
    public void getCustomerName() {...};
    publicStream<LineItem> lineItems() {...};
}
```

So basically flatMapping() just takes whatever you're working with and tries to derive a stream out of it, at which point it'll flatten that stream and pass it to the downstream collector. It's useful when your collector requires other collectors ("multi-level reduction"), such as groupingBy and partitioningBy().

How is the above example any different than doing...

```
Map<String, Set<LineItem>> itemsByCustomerName
   = orders.stream().collect(
     groupingBy(Order::getCustomerName,
            o -> o.stream().collect(toSet())));
```

The above won't work because the 2nd parameter to groupingBy() expects a Collector, not a Collection.

Time API Changes

Several new methods have been added:

- Clock.systemUTC() // gets the most accurate system clock
- Duration.dividedBy() // divides one duration by another
- Duration.truncatedTo() // truncates to a specific time unit (e.g. minutes)
- LocalDate.datesUntil() // stream of dates until some end date

Language Changes

_ Identifier

Underscore (_) is now an identifier -- you can't use it for names (e.g. param/class name)

Try-with-resources Using Variables

Try-with-resources can now use variables directly, as long as they're "effectively final" (as long as they're set once and never changed, as if it were a final variable). For example…

```
public void test(InputStream is) {
  try (is) {
     fis.read(...)
  }
}
```

Before this, you had to alias whatever you were working with as a new variable in the try block...

```
public void test(InputStream is) {
  try (InputStream is2 = is) {
     is2.read(...)
  }
}
```

Diamond Operator in Anonymous Inner Classes

Using the diamond operator when you're declaring an anonymous inner class now works...

```
MyImplementation<String> obj = new MyImplementation<> () {
    @Override
    public void do() { ...}
}
```

Private Interface Methods

You can now declare private methods in interfaces, where your default methods can access these private methods to reduce duplicate code…

```
public interface MyInterface() {
    default public String getX() {
        return getDefaultName() + "X";
    }

    default public String getY() {
        return getDefaultName() + "Y";
    }

    private String getDefaultName() {
        return "name";
    }
}
```

Deprecations / Removals

- VisualVM is gone.
- hprof/jhat is gone (use jmap and Eclipse Memory Analyzer instead).
- javac now can only target Java 1.6+.
- rt.jar is gone (replaced by the new module system).
- Most internal APIs are gone (sun.misc.Unsafe is still around but will be removed).
- JavaDB is gone (use Apache Derby directly, JavaDB was a clone of Derby).
- Java EE modules have been deprecated and no longer visible by default.
- Applets are gone.

Other Changes

- StackWalker is a new niche API to inspect the stack (too obscure to cover in detail).
- G1 is now the default garbage collector,
- JavaDoc HTML5 compliant output.

- JavaDoc output searchable (there's a search bar now).
- Unicode 8.0 support.
- Properties files default to UTF-8 instead of ISO-8859-1.
- JDK locale data now comes from Common Locale Data Repository files.
  (number formats, date formates, etc..)
- HiDPI support.
- TIFF image suppoer via TIFFImage.
- Multi-resolution image support via MultiResolutionImage.
- Renderer used for GUIs has been changed to Marlin (FOSS).
- Font renderer has been changed from ICU to Harfbuzz (FOSS).
- java.awt.Desktop has added new methods to interface with the underlying OS's desktop.
  (e.g. displaying taskbar notifications, show the native file explorer)
- JavaFX API changes to support skinning.
  (some internal APIs have been hidden and new APIs have been added)
- String performance improvements.
- TLS security improvements.

Java 10

Language Changes

Local Variable Type Inference

Java 10 introduces the var ~~keyword~~ reserved type name. Just like other languages, var does local variable type inference. For example, instead of…

```
MyClass test = new MyClass();
```

, you can write…

```
var test = new MyClass();
```

The type of variable is inferred based on what's on the right-hand side of the assignment. The right hand-side could be a constructor or an expression or a method invocation.

```
var test = new MyClass();
var test2 = test.getName();
var test3 = 4 + 5L;
var var = "var";// REMEMBER: var isn't a keyword, it's a reserve type
                // name. That's why you can have a variable named var.
```

In certain cases, there isn't enough information to make inference isn't possible. For example…

```
var t = m -> m == 5;// no way to figure out which interface class this
```

```
                    // lambda maps to
var x = null;
```

In certain cases, it looks like there isn't enough information to make inference possible, but the Java compiler hacks in what it thinks is appropriate. For example, with diamond operations…

```
var t = new HashSet<>(); //t will be of type HashSet<Object>
var x = new HashSet() { }; //x type is HashSet$1 (NOT HashSet), which is
                           //the internally generated name for the
                           //anonymous inner class
x = new HashSet(); //won't work because type of x is HashSet$1, not HashSet
var y = Set.of(0.1, 2, 3L, "4");//type of y is
                                //Set<? extends Serializable&Comparable<>>
                                //because that's what the common super
                                //types are (intersection type)
```

The takeaway here is to only use var when it's painfully obvious what the type is (or maybe just don't use it at all).

Note that var is for local types only, you can't use it for method parameters, fields, return types, etc..

Performance Changes

Class Data Sharing

Class Data Sharing (CDS) is a way to reduce the JVM startup time by sharing metadata across multiple instances, such that each run won't have to parse classes, validate/verify them, etc..

To use CDS on only system classes…

1. create CDS archive using java -Xshare:dump
2. launch JVMs using java -jar app.jar -Xshare:on

To use CDS on your application classes…

1. launch a JVM with java -jar app.jar -XX:DumpLoadedClassList=classes.txt and kill it.
2. create CDS archive using java --class-path <path> -Xshare:dump -XX:+UseAppCDS -XX:SharedClassListFile=classes.xt -XX:SharedArchiveFile=myapp.jsa
3. launch JVMs using -Xshare:on -XX:+UseAppCDS -XX:SharedArchiveFile=myapp.jsa

Note that classes.txt refers to a text file that contains all the classes you want shared, including your application classes and third-party classes. The JVM fills up this file during step 1 as classes are accessed, so if all you want to do is boost the startup time, launch your application

once and kill it right away. But, if you want to include classes for features touched later on, you should make use of those features at least once before killing the JVM.

Note that -XX:DumpLoadedClassList may not do the right thing if you're using custom class loaders (e.g. in containers like tomcat or jetty). The material recommends using https://github.com/simonis/cl4cds for these cases.

G1 Changes

In certain cases, G1 can't free enough memory using its normal incremental garbage collector approach. If this happen, G1 does a full stop-the-world garbage collection run. In Java 10, a full garbage collection run uses all cores on the CPU instead of a single thread.

To control the number of threads used for a full garbage collector run, use the -XX:ParallelGCThreads JVM arg.

Deprecations / Removals

- javah is gone -- (use javac -h instead to generate native C++ headers).
- policytool is gone (edit your policy files directly using a text editor).
- java -X:prof is gone (use jmap to get profiling information or a 3rd party profiler).
- java.security.acl is deprecated (replacements are in java.security)
- java.security.Certificate/Identity/IdentityScope/Signer are deprecated.
- javax.security.auth.Policy (replace with java.security.policy)

Other Changes

- OracleJDK and OpenJDK are converging.
  (they're going to be the exact same, so no more closed source code in OracleJDK)

Java 11

Single Source Files

You can now run java files directly without first compiling them first…

```
$ java /home/user/Hello.java
Hello world!
```

You can target different source-code compliance levels by using the --source flag...

```
$ java --source 11 /home/user/Hello.java
```

```
Hello world!
```

The syntax is slightly different for Java source files run directly vs compiled. Your source code...

1. may begin with a shebang, incase you want to run directly it from the shell.
   (e.g. #!/usr/bin/java --source 11)
2. may include multiple top-level classes.
3. doesn't have to have a top-level class with a name that matches the file name.
   (first declared class is the one that gets run)

New Garbage Collectors

G1 has had improvements to increase performance. It's still the default garbage collector in JDK11, but 2 new garbage collectors have also been added:

- Epsilon → Performs no GC at all. Used for applications where…
  - memory usage is strictly bounded.
  - application runtime is incredibly short.
  - you're performance testing your code and you don't want GC overhead screwing up your metrics.
- Z → GC with consistent pause times (under 10ms) scalable to multi-terabyte heaps. Pause times shouldn't increase with heap increases.

To use Epsilon, use args -XX:+UnlockExperimnalVMOptions -XX:+UseEpsilonGC.
To use Z, us args -XX:+UnlockExperimnalVMOptions -XX:+UseZGC.

HTTPClient API

The HTTPClient API is an HTTP client implementation similar to Apache HttpClient/HttpAsyncClient. It's meant as a more controllable/flexible replacement for HttpURLConnection.

The HTTPClient API supports both HTTP/1.1 and HTTP/2 (and websockets?) and lets you set options that you were normally only available set on third-party clients: follow redirects, custom authenticators, proxy settings, timeouts, etc…

Full coverage of the API is out of scope for this document, but the following example should be enough to get started…

```
HttpClient client = HttpClient.newBuilder()
        .version(HttpClient.Version.HTTP_2)
        .followRedirects(HttpClient.Redirect.ALWAYS)
        .build();
```

```
HttpRequest request = HttpRequest
        .newBuilder(URI.create("https://www.google.com"))
                .header("User-Agent", "Java")
                .timeout(Duration.ofMillis(500))
                .GET()
                .build();

HttpResponse<String> response = client.sendAsync(
        Request, BodyHandlers.ofString()).get();

if (response.statusCode() == 200) {
    System.out.println(response.headers().map());
}

System.out.println("Version: " + response.version());
System.out.println(response.body());
```

String API Changes

String.repeat()

Repeats a string up to n times.

String.strip() / String.stripLeading() / String.stripTrailing() / String.isBlank()

String.strip() removes leading and trailing whitespace from a string. This is different from String.trim() in that strip() takes out any whitespace character while trim() only focuses on a normal space character (U+0020).

Similarly, String.stripLeading() and String.stripTrailing() removes leading and trailing whitespace from a string respectively.

String.isBlank() checks to see if a string only consists of whitespace characters only. This is different from doing String.trim().equals(“”) because trim() doesn’t work for all whitespace, just the space character.

String.lines()

Breaks up a string by lines: \n, \r, or \r\n.

IO API Changes

Files.readString() / Files.writeString()

These are convenience methods that read an entire file to a string / write a string out to a file.

Language Changes

Local Variable Type Inference Updates

It's now possible to use var when declaring lambda parameters…

```
(String a, String b) -> a.concat(b);
(a, b) -> a.concat(b);
(var a, var b) -> a.concat(b); // wasn't possible in 10
```

If you're going to use var on a parameter, you need to use it on all parameters and you need to have encapsulating brackets…

```
(var a, var b) -> a.concat(b); // allowed
(var a, b) -> a.concat(b); // NOT ALLOWED
(var a, String b) -> a.concat(b); // NOT ALLOWED
var a -> a.concat(b); // NOT ALLOWED
```

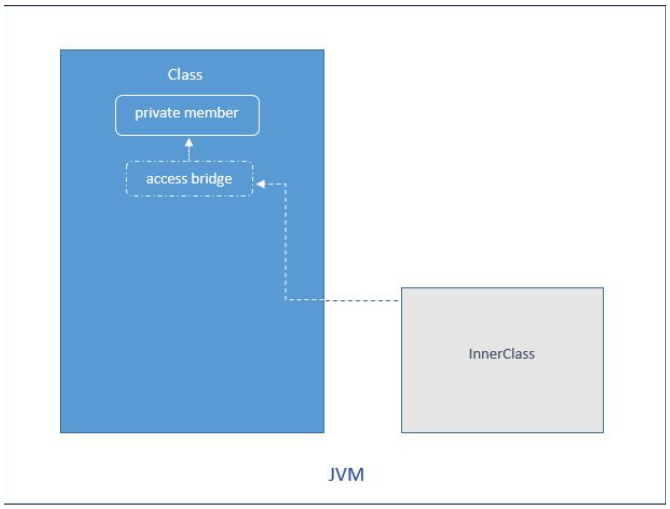
The material says that you'd want to use var in cases where you don't want to explicitly write out the type, but you do want have annotations on it…

```
(@NotNull var a, @NotNull var b) -> a.concat(b);
```

Bytecode Changes

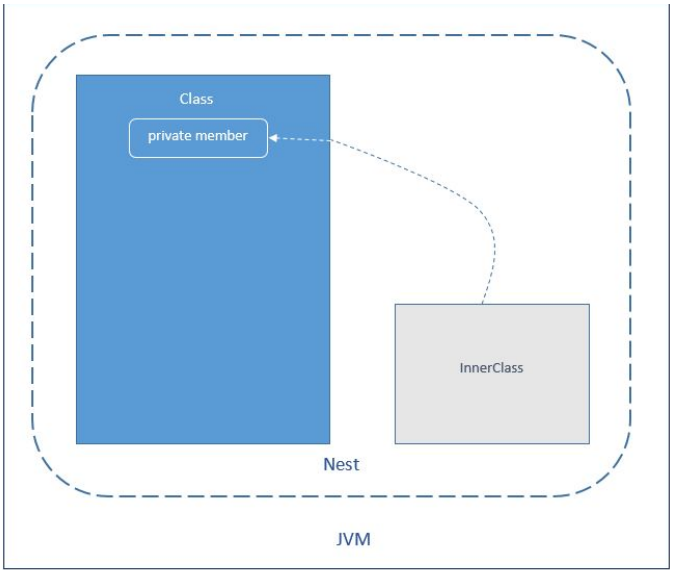
Nest-based Access Control

Nest-based access control is a JVM-level class change that bundles specific classes together such that they can access each other's private variables. For example, in the past when an inner class accessed a private field in its parent class, the Java compiler built a public bridge method to access that variable...



(https://medium.com/@kraghavendrarao1/java-11-a-first-look-2d447e00926c)

Starting with Java 11, that's no longer required. The inner class will be able to directly access the private field in the parent class..



(https://medium.com/@kraghavendrarao1/java-11-a-first-look-2d447e00926c)

More information on the spec is here:
https://docs.oracle.com/javase/specs/jvms/se11/html/jvms-4.html#jvms-4.7.28 and
https://docs.oracle.com/javase/specs/jvms/se11/html/jvms-4.html#jvms-4.7.29.

> **NOTE**: It's unlikely coroutines will need changes to support this?

CONSTANT_Dynamic

CONSTANT_Dynamic is a JVM-level feature that's like the INVOKEDYNAMIC opcode, but for the class's constant pool instead. It isn't a opcode but a constant pool type that invokes some code to determine what the actual constant is.

More information about the spec is here: https://docs.oracle.com/javase/specs/jvms/se11/html/jvms-4.html#jvms-4.4.10.

> **NOTE**: It's unlikely coroutines will need changes to support this?

Deprecations / Removals

- Java EE modules are gone (still available on Maven or application servers like JBoss)
  - java.corba (CORBA)
  - java.transaction (JTA)
  - java.xml.ws (JAX-WS)
  - java.xml.ws.annotation
  - java.xml.bind (JAXB)
  - java.activation (JavaBeans Activation)
- Thread.destroy() is gone.
- Thread.stop() is gone.
- System.runFinalizersOnExit() is gone.
- Runtime.runFinalizersOnExit() is gone.
- SecurityManager AWT checks are gone (replace with checkPermission(Permission p))
  - SecurityManager.checkAwtEventQueueAccess() is gone.
  - SecurityManager.checkSystemClipboardAccess() is gone.
  - SecurityManager.checkTopLevelWindow() is gone.
  - SecurityManager.checkMemberAccess() is gone.
- JavaFX is split off from OpenJDK (use OpenJFX instead)
- javapackager is gone (no alternative, find a different way to package your app)
- Java Web Start is gone (no alternatives)
- Nashorn is deprecated (Graal.JS is Oracle's new pure Java JS implementation)

Other Changes

- TLS updated to 1.3
- Unicode 10 support.