Java 10

1.. Local Variable Type Inference

Until Java 9, we had to mention the type of the local variable explicitly and ensure it was compatible with the initializer used to initialize it.

```
String message = "Good bye, Java 9";
```

In Java 10, this is how we could declare a local variable.

```
@Test
public void whenVarInitWithString_thenGetStringTypeVar() {
    var message = "Hello, Java 10";
    assertTrue(message instanceof String);
}
```

We don't provide the data type of message. Instead, we mark the message as a var, and the compiler infers the type of message from the type of the initializer present on the right-hand side. In above example, the type of message would be String.

Note that this feature is available only for local variables with the initializer. It cannot be used for member variables, method parameters, return types, etc – the initializer is required as without which compiler won't be able to infer the type. This enhancement helps in reducing the boilerplate code. Example:

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

This can now be rewritten as:

```
var idToNameMap = new HashMap<Integer, String>();
```

This also helps to focus on the variable name rather than on the variable type. Another thing to note is that **var** is not a keyword – this ensures backward compatibility for programs using var say, as a function or variable name. **var** is a reserved type name, just like int.

Finally, note that there is no runtime overhead in using var nor does it make Java a dynamically typed language. The type of the variable is still inferred at compile time and cannot be changed later.

As mentioned earlier, var won't work without the initializer and nor would it work if initialized with null and It won't work for non-local variables.

```
var n; // error: cannot use 'var' on variable without initializer
```

```
var emptyList = null; // error: variable initializer is 'null'
```

```
public var = "hello"; // error: 'var' is not allowed here
```

Lambda expression needs explicit target type, and hence var cannot be used.

```
var p = (String s) -> s.length() > 10; // error: lambda expression needs an explicit target-type
```

Same is the case with the array initializer.

```
var arr = { 1, 2, 3 }; // error: array initializer needs an explicit target-type
```

There are situations where var can be used legally, but may not be a good idea to do so. For example, in situations where the code could become less readable.

```
var result = obj.prcoess();
```

Here, although a legal use of var, it becomes difficult to understand the type returned by the process()making the code less readable. **java.nethas** a dedicated article on Style Guidelines for Local Variable Type Inference in Java which talks about how we should use judgment while using this feature. Another situation where it's best to avoid var is in streams with long pipeline.

```
var x = emp.getProjects.stream()
  .findFirst()
  .map(String::length)
  .orElse(0);
```

Usage of var may also give unexpected result. For example, if we use it with the diamond operator introduced in Java 7:

```
var empList = new ArrayList<>();
```

The type of **empList** will be ArrayList<Object>and not List<Object>. If we want it to be ArrayList<Employee>, we will have to be explicit:

```
var empList = new ArrayList<Employee>();
```

Using var with non-denotable types could cause unexpected error. For example, if we use var with the anonymous class instance.

```
@Test
public void whenVarInitWithAnonymous_thenGetAnonymousType() {
    var obj = new Object() {};
    assertFalse(obj.getClass().equals(Object.class));
}
```

Now, if we try to assign another Object to obj, we would get a compilation error:

```
obj = new Object(); // error: Object cannot be converted to <anonymous Object>
```

This is because the inferred type of obj isn't Object.

2.. Unmodifiable Collection Enhancements

There are a couple of changes related to unmodifiable collections in Java 10.

2.1 copyOf()

java.util.List, **java.util.Map** and **java.util.Set** each got a new static method **copyOf(Collection).** It returns the unmodifiable copy of the given Collection. Any attempt to modify such a collection would result in **java.lang.UnsupportedOperationException** runtime exception.

```
@Test(expected = UnsupportedOperationException.class)
public void whenModifyCopyOfList_thenThrowsException() {
    List<Integer> copyList = List.copyOf(someIntList);
    copyList.add(4);
}
```

2.2 toUnmodifiable*()

java.util.stream.Collectors get additional methods to collect a Stream into unmodifiable List, Map or Set. Any attempt to modify such a collection would result in **java.lang.UnsupportedOperationException** runtime exception.

```
@Test(expected = UnsupportedOperationException.class)
public void whenModifyToUnmodifiableList_thenThrowsException() {
    List<Integer> evenList = someIntList.stream()
      .filter(i -> i % 2 == 0)
      .collect(Collectors.toUnmodifiableList());
    evenList.add(4);
}
```

3.. Optional class Enhancement

java.util.Optional, java.util.OptionalDouble, java.util.OptionalInt and java.util.OptionalLong each got a new method orElseThrow() which doesn't take any argument and throws **NoSuchElementException** if no value is present.

```
@Test
public void whenListContainsInteger_OrElseThrowReturnsInteger() {
    Integer firstEven = someIntList.stream()
      .filter(i -> i % 2 == 0)
      .findFirst()
      .orElseThrow();
    is(firstEven).equals(Integer.valueOf(2));
}
```

It's synonymous with and is now the preferred alternative to the existing get()method.

4.. Performance Improvements

The performance improvements that come along with the latest Java 10 release which is applied to all applications running under JDK 10, with no need for any code changes to leverage them.

4.1 Parallel Full GC for G1

The G1 garbage collector is the default one since JDK 9. However, the full GC for G1 used a single threaded mark-sweep-compact algorithm. This has been changed to the parallel mark-sweep-compact algorithm in Java 10 effectively reducing the stop-the-world time during full GC.

4.2 Application Class-Data Sharing

Class-Data Sharing, introduced in JDK 5, allows a set of classes to be pre-processed into a shared archive file that can then be memory-mapped at runtime to reduce startup time which can also reduce dynamic memory footprint when multiple JVMs share the same archive file. CDS only allowed the bootstrap class loader, limiting the feature to system classes only. Application CDS (AppCDS) extends CDS to allow the built-in system class loader (a.k.a., the "app class loader"), the built-in platform class loader, and custom class loaders to load archived classes. This makes it possible to use the feature for application classes.

We can use the following steps to make use of this feature:

1. Get the list of classes to archive

The following command will dump the classes loaded by the HelloWorld application into **hello.lst**

```
$ java -Xshare:off -XX:+UseAppCDS -XX:DumpLoadedClassList=hello.lst \
    -cp hello.jar HelloWorld
```

2. Create the AppCDS archive

Following command creates hello.js a using the **hello.lst** as input:

```
$ java -Xshare:dump -XX:+UseAppCDS -XX:SharedClassListFile=hello.lst \
    -XX:SharedArchiveFile=hello.jsa -cp hello.jar
```

3. Use the AppCDS archive

Following command starts the HelloWorld application with **hello.jsa** as input:

```
$ java -Xshare:on -XX:+UseAppCDS -XX:SharedArchiveFile=hello.jsa \
    -cp hello.jar HelloWorld
```

AppCDS was a commercial feature in Oracle JDK for JDK 8 and JDK 9. Now it is open sourced and made publicly available.

4.3 Experimental Java-Based JIT Compiler

Graal is a dynamic compiler written in Java that integrates with the Hotspot JVM; it's focused on high performance and extensibility. It's also the basis of the experimental Ahead-of-Time (AOT) compiler introduced in JDK 9.

JDK 10 enables the Graal compiler, to be used as an experimental JIT compiler on the Linux/x64 platform.

To enable Graal as the JIT compiler, use the following options on the java command line:

```
-XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler
```

Note that this is an experimental feature and we may not necessarily get better performance than the existing JIT compilers.

5.. Container Awareness

JVMs are now aware of being run in a Docker container and will extract container-specific configuration instead of querying the operating system itself – it applies to data like the number of CPUs and total memory that have been allocated to the container. However, this support is only available for Linux-based platforms. This new support is enabled by default and can be disabled in the command line with the JVM option.

```
-XX:-UseContainerSupport
```

Also, this change adds a JVM option that provides the ability to specify the number of CPUs that the JVM will use.

```
-XX:ActiveProcessorCount=count
```

Also, three new JVM options have been added to allow Docker container users to gain more fine-grained control over the amount of system memory that will be used for the Java Heap.

```
-XX:InitialRAMPercentage
-XX:MaxRAMPercentage
-XX:MinRAMPercentage
```

6.. Root Certificates

The **cacerts** keystore, which was initially empty so far, is intended to contain a set of root certificates that can be used to establish trust in the certificate chains used by various security protocols. As a result, critical security components such as TLS didn't work by default under OpenJDK builds. With Java 10, Oracle has open-sourced the root certificates in Oracle's Java SE Root CA program in order to make OpenJDK builds more attractive to developers and to reduce the differences between those builds and Oracle JDK builds.

7.. Deprecations and Removals

7.1 Command Line Options and Tools

Tool javah has been removed from Java 10 which generated C headers and source files which were required to implement native methods – now, javac -h can be used instead. Policy tool was the UI based tool for policy file creation and management. This has now been removed. The user can use simple text editor for performing this operation.

Removed java **-Xprof** option. The option was used to profile the running program and send profiling data to standard output. The user should now use **jmap** tool instead.

7.2 APIs

Deprecated **java.security.acl** package has been marked forRemoval=true and is subject to removal in a future version of Java SE. It's been replaced by **java.security.Policy** and related classes. Similarly, java.security.{Certificate,Identity,IdentityScope,Signer} APIs are marked forRemoval=true.

8.. Time-Based Release Versioning

Starting with Java 10, Oracle has moved to the time-based release of Java. This has following implications. A new Java release every six months. The March 2018 release is JDK 10, the September 2018 release is JDK 11, and so forth. These are called feature releases and are expected to contain at least one or two significant features Support for the feature release will last only for six months, i.e., until next feature release. Long-term support release will be marked as LTS. Support for such release will be for three years. Java 11 will be an LTS release. java -version will now contain the GA date, making it easier to identify how old the release is.

```
$ java -version
openjdk version "10" 2018-03-20
OpenJDK Runtime Environment 18.3 (build 10+46)
OpenJDK 64-Bit Server VM 18.3 (build 10+46, mixed mode)
```