0-language-and-features

Contents

Release

Java 9 (Released in September 2017). This language release adds many improvements to the libraries and `APIs` but does not introduce any new significant changes to the base language itself unlike future iterations which do, like java 10 or 11. Java 10 Released in March 2018, and Java 11 Released September 2018, are more incremental changes and improvements to the Java 9 release. Below are presented the most notable features from each release between java 9 and 11

General

1. `JShell` (Java Shell) - Interactive command-line tool for evaluating code snippets. Ideal for learning, experimentation, and quick prototyping. One can use the java shell to REPL to quickly test and iterate over small pieces of code, for example the shell script below is used to start the jshell with a default set of import statements which allows the user to quickly use library features which are not immediately accessible.

   The `startup.jsh` file could contain a set of `pre-defined` initialization steps, in this example it simply contains a set of import statements which are to be loaded when the `jshell` is started

   ```
   import java.util.*;
   import java.time.*;
   import java.io.*;
   import java.math.*;
   import java.nio.file.*;
   import java.net.*;
   import java.text.*;
   import java.util.stream.*;
   ```

The command below starts the shell, and passes the file defined above, with the import statements to initialize the shell environment outright

```
jshell --startup startup.jsh
```

2. `@SafeVarargs` on Private Methods - The `@SafeVarargs` annotation, previously allowed only on final and static methods, can now be used with private methods. That annotation is used to mark method's variable arguments as being used in a safe manner by the method's implementation, in the example below, the actual usage is not safe, even though marked, it is showing that the complier will not emit any warnings since the method is annotated with the `SafeVarargs` annotation however, during run-time the implementation will fail with `ClassCastException`

```
The body of the method or constructor declaration performs potentially unsafe
    operations, such as an assignment to an
element of the variable arity parameter's array that generates an unchecked
   warning. Some unsafe operations do not
trigger an unchecked warning. For example, the aliasing in
```

```
@SafeVarargs // Not an actually safe implementation!
private static void method(List<String>... stringLists) {
    Object[] array = stringLists;
    List<Integer> tmpList = Arrays.asList(42);
    array[0] = tmpList; // Semantically invalid, but compiles without
       warnings
    String s = stringLists[0].get(0); // Oh no, ClassCastException at runtime
       !
}
```

3. `try-with-resources` Improvements - You can now use final or effectively final variables with `try-with-resources` without explicitly defining them inside the block. Note that this variable has to be considered final by the compiler, since if it is re-assigned it would be hard to detect which resource has to be released, and the value or resource reference before the re-assignment will leak

   Prior to java 9 the following had to be done in order to use the resource within the block,

```
FileInputStream fileInputStream = new FileInputStream("test.txt");
try (FileInputStream input = fileInputStream) {
    // use input
}
```

   However with the new addition the final variable can be outside of the scope and be re-used within the try-with-resources without being redefined. The variable can be effectively final, meaning that as long as it is not re-assigned anywhere, the compiler will not complain about it. However if it is re-assigned compile time error will occur.

```
FileInputStream fileInputStream = new FileInputStream("test.txt");
// fileInputStream is effectively final (not reassigned anywhere)
try (fileInputStream) {
    // use fileInputStream variable directly, can not be re-assigned
}
```

Language

1. `var keyword for local variables and type inference`: Allows for local variable declarations without explicitly specifying the type. Java infers the type based on the right hand side expression.

However there are some specific use cases where the usage of var is restricted in the language

- Can `not` be used to define class members and fields - `public final class Test { private var test = "string"; }`
- Can `not` be used as return type of functions or methods - `public var method(){ return "" }`
- Can `not` be used to assign null - `var invalid = null;`
- Can `not` be used to infer lambda reference results - `var lambda = ()-> "i-return-string";`
- Can `not` be used to create compound assignments - `var n = 1, f = 10;`
- Can not be used for method parameters - `public void method(var param){ return "" }`

Besides the restrictions above, the `var` keyword can be used in every other context, it is mostly a syntax sugar around verbose types and names, when a variable is assigned through var it will infer the type of the right hand expression, never the super or sub-type of the given expression - `var arrayList = new ArrayList<>()`, the type of `arrayList` will be `ArrayList`. Explicit casts are permitted where they are valid - `var nilString = (String)null;`

Library

1. `Stream` API Enhancements - new methods `takeWhile`, `dropWhile`, and iterate for more fluent functional-style programming. These are very similar to `filter` however there is a key difference, while `filter` goes through all elements - not stopping, possibly not in order, the new methods always go over elements in order and stop at the first element that does not match the predicate (takeWhile) or start processing in order and remove elements as long as the predicate is fulfilled (dropWhile)

   The new methods make sense when they are used on ordered/sorted streams of elements, such as the list below, since they will execute in order starting from the beginning, stopping at the first element that does not meet the predicate condition

   ```
   List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);
   List<Integer> result = numbers.stream()
                                 .takeWhile(n -> n < 5)
                                 .collect(Collectors.toList());

   System.out.println(result); // Output: [1, 2, 3, 4]
   ```

2. `Optional` API Enhancements - methods `ifPresentOrElse`, or, and stream added to improve the usability of Optional. The new method gives the ability for one to run two actions on optional one when the optional is present, the other when the optional is empty, both arguments use lambda function reference and the general signature of the method looks like that

   ```
   ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)
   ```

3. `CompletableFuture` - new methods `newIncompleteFuture`, `defaultExecutor`, `completeAsync`, and `orTimeout`. The most notable is `completeAsync`, which allows one to modify the completion state of the future, after it was created but before it was started or completed. It is similar to `supplyAsync` which is a static method which produces a new instance of the `CompletedFuture`, while `completeAsync` is a member method which mutates the current instance's state. Another one `newIncompleteFuture` is a companion to the new `completeAsync` it basically is a replacement for calling `supplyAsync` without running the future, to run it call `thenApply`

Platform

1. `Multi-Release JARs` - allows for packaging multiple versions of classes optimized for different Java versions within the same JAR. The structure of the jar is such that by default for java 8 and below, the

compiled class files are still contained at the root of the jar, however, the META-INF folder can contain sub-directories for different java versions (9 and above). To compile the jar first call `javac -d out/base src/com/myapp/*.java` and to add additional release versions for different java run-times simply run `javac --release 9 -d out/9 src9/com/myapp/*.java`, that will compile the classes, but will not package a jar, to package the jar - `jar --create --file library.jar --main-class com.myapp.Main -C out/base .` to package the base implementation, and `jar --update --file mylibrary.jar --release 9 -C out/9 .` to update the created jar with the new release classes.

```
mylibrary.jar
|-- com/myapp/Main.class                  # Default (Java 8 and below)
|-- META-INF
|   -- versions
|       -- 9
|           -- com/myapp/Main.class      # Java 9+ version of Main.class
|       -- 11
|           -- com/myapp/Main.class      # Java 11+ version of Main.class
```

2. `Java Platform Module System (Project Jigsaw)`: Modularized the JDK into smaller modules, reducing memory usage and improving startup time by only loading necessary modules. Introduced the module-info.java file. This feature allows jar packages to package only the required modules which are being used. Decreases the general footprint of the jar significantly, reduces friction, and improves performance. Inside each package, or on package level, a special `module-info.java` file can be provided which specifies what the module exports, what it depends or requires and defines the name of the module. What is more interesting is the fact that the JDK itself was modularized, meaning that the end-user has the ability to package their own JDK, which would include only the required/used modules by the specific application. This is also useful since this packages jdk can be distributed along with the application package, providing a 100% reproduce able environment and run-time.

```
module com.example.myapp {
    requires java.sql;             // Requires another module (from the JDK)
    requires com.external.library; // Requires a custom module, for example,
       an external library

    exports com.example.myapp.api;           // Exports the API package
    exports com.example.myapp.utils;         // Exports a utilities package

    // Internal packages are not exported and remain hidden by default
}
```

The module-info file provides meta information for package exports, it does not export individual classes, in the example above, the root package `com.example.myapp`, exports a specific number of modules, it also specifies dependencies, Unlike typical `fat-jars` - jars built by systems like `maven` or `gradle`, that contain all dependencies for a given application, the `module` approach works differently, the final jar contains only the actual application, however when the application is started the modules are provided on the `module-path` the module path is like the class path, it is a list of modules which have to be loaded by the run-time, those are the dependent modules for the application. Below is a short script which shows how a module based application is run, the module-path specifies a directory with dependencies, directory full of `jars` and the module application itself. The module approach is an alternative to provide a more fine grained control over traditional build-systems, like `gradle` or `maven`, which are third party tools, unrelated to java itself, now the language provides its own idiomatic specification on how to manage and control dependencies, through a higher level module system, instead of specifying class-paths

Module way of building, packaging & running, during packaging, modules are used only for compiling, during running the modules have to be again specified

```
javac -d $BIN_DIR --module-path "$LIBS_DIR/external-library.jar" $SRC_DIR/
   module-info.java $SRC_DIR/MainApp.java $SRC_DIR/utils/Utils.java # Package
    the application
java --module-path "$LIBS_DIR/external-library.jar:$BIN_DIR" --module com.
   example.myapp/com.example.myapp.MainApp # Run the application
# Note that the 3rd party libraries are present during compilation and
   running of the application, two stages, imply one can upgrade them
```

Traditional way of building, packaging & running, everything is included on the `classpath`, 3rd party and actual application classes

```
javac -d $BIN_DIR -cp "$LIBS_DIR/external-library.jar" $SRC_DIR/MainApp.java
   $SRC_DIR/utils/Utils.java # Package the application
java -cp "$BIN_DIR:$LIBS_DIR/external-library.jar" MainApp # Run the
   application
# Note libraries are packaged in the final jar, meaning that there is no easy
    way to provide new versions without re-packaging everything
```

```
Note that the module approach gives the power of upgrading the 3rd party libraries
during run-time / running the application without having to re-package the application
 with new versions of the libraries, as long as the API provided by the libraries
 is still compatible, which is quite powerful, since one could upgrade only the
application dependencies without having to re-compile the application
```

The module approach bears very strong resemblance to how DLLs (dynamic link libraries) or Shared Objects work, the general premise is the same, the application is compiled against the API - contract of a given library, however the implementation itself is not packaged within the application, it is linked dynamically during run-time, hence the name, The old approach of bundling resembles the static linking in a way

3. `Java Linker`: New tool to create custom `runtime` images with only required modules, reducing application size. As a continuation of the `module` approach, the `jlink` now gives one the ability to package a java runtime along with the application itself, in a single jar. This is like a fat-jar, but instead of 3-rd party libraries in the jar, the jar contains the `JDK` runtime modules, only those used / required by the application, along with the application source itself, the 3rd party libraries are still specified outside during `runtime`

```
# Compile the application with required modules and 3rd party dependencies
javac -d mods/com.example.myapp --module-path libs src/com/example/MyApp.java
    src/module-info.java

# Create and package a minified version of the JDK along with the application
    itself
jlink   --module-path "$JAVA_HOME/jmods:mods" \
        --add-modules com.example.myapp \
        --output my-custom-runtime \
        --strip-debug \
        --compress 2 \
        --no-header-files \
        --no-man-pages
```

```
# Run The minified JDK, which contains its own binaries - java/javac
./my-custom-runtime/bin/java -m com.example.myapp/com.example.MyApp
```