Java 11 Tutorial

September 24, 2018

Java 11 is around the corner and many people still use Java 8 in production. This example-driven tutorial covers the most important language and API features from Java 9 to 11. No walls of text, so let's dive right into the code. Enjoy!

Local Variable Type Inference

Java 10 has introduced a new language keyword `var` which optionally replaces the type information when declaring local variables (*local* means variable declarations inside method bodies).

Prior to Java 10 you would declare variables like this:

```
String text = "Hello Java 9";
```

Now you can replace `String` with `var`. The compiler infers the correct type from the assignment of the variable. In this case `text` is of type `String`:

```
var text = "Hello Java 10";
```

types to such variables. This code snippet does not compile:

```
var text = "Hello Java 11";
text = 23;  // Incompatible types
```

You can also use `final` in conjunction with `var` to forbid reassigning the variable with another value:

```
final var text = "Banana";
text = "Joe";   // Cannot assign a value to final variable 'text'
```

Also `var` is not allowed when the compiler is incapable of infering the correct type of the variable. All of the following code samples result in compiler errors:

```
// Cannot infer type:
var a;
var nothing = null;
var lambda = () -> System.out.println("Pity!");
var method = this::someMethod;
```

Local variable type inference really shines with generics involved. In the next example `current` has a rather verbose type of `Map<String, List<Integer>>` which can be reduced to a single `var` keyword, saving you from typing a lot of boilerplate:

```
var myList = new ArrayList<Map<String, List<Integer>>>();

for (var current : myList) {
    // current is infered to type: Map<String, List<Integer>>
    System.out.println(current);
}
```

As of Java 11 the `var` keyword is also allowed for lambda parameters which enables you to add annotations to those parameters:

```
Predicate<String> predicate = (@Nullable var a) -> true;
```

> *Tip: In Intellij IDEA you can hover on a variable while holding `CMD/CTRL` to reveal the infered type of the variable (for keyboard junkies press `CTRL + J`).*

Java 9 introduced a new incubating `HttpClient` API for dealing with HTTP requests. As of Java 11 this API is now final and available in the standard libraries package `java.net`. Let's explore what we can do with this API.

The new `HttpClient` can be used either synchronously or asynchronously. A synchronous request blocks the current thread until the reponse is available. `BodyHandlers` define the expected type of response body (e.g. as string, byte-array or file):

```
var request = HttpRequest.newBuilder()
    .uri(URI.create("https://winterbe.com"))
    .GET()
    .build();
var client = HttpClient.newHttpClient();
HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers
System.out.println(response.body());
```

The same request can be performed asynchronously. Calling `sendAsync` does not block the current thread and instead returns a `CompletableFuture` to construct asynchronous operation pipelines.

```
var request = HttpRequest.newBuilder()
    .uri(URI.create("https://winterbe.com"))
    .build();
var client = HttpClient.newHttpClient();
client.sendAsync(request, HttpResponse.BodyHandlers.ofString())
    .thenApply(HttpResponse::body)
    .thenAccept(System.out::println);
```

> *We can omit the `.GET()` call as it's the default request method.*

The next example sends data to a given URL via `POST`. Similiar to `BodyHandlers` you use `BodyPublishers` to define the type of data you want to send as body of the request such as strings, byte-arrays, files or input-streams:

```
var request = HttpRequest.newBuilder()
    .uri(URI.create("https://postman-echo.com/post"))
    .header("Content-Type", "text/plain")
    .POST(HttpRequest.BodyPublishers.ofString("Hi there!"))
    .build();
var client = HttpClient.newHttpClient();
```

The last sample demonstrates how to perform authorization via `BASIC-AUTH`:

```
var request = HttpRequest.newBuilder()
    .uri(URI.create("https://postman-echo.com/basic-auth"))
    .build();
var client = HttpClient.newBuilder()
    .authenticator(new Authenticator() {
        @Override
        protected PasswordAuthentication getPasswordAuthentication() {
            return new PasswordAuthentication("postman", "password".toCharArra
        }
    })
    .build();
var response = client.send(request, HttpResponse.BodyHandlers.ofString());
System.out.println(response.statusCode());      // 200
```

Collections

Collections such as `List`, `Set` and `Map` have been extended with new methods. `List.of` created a new immutable list from the given arguments. `List.copyOf` creates an immutable copy of the list.

```
var list = List.of("A", "B", "C");
var copy = List.copyOf(list);
System.out.println(list == copy);   // true
```

Because `list` is already immutable there's no practical need to actually create a copy of the list-instance, therefore `list` and `copy` are the same instance. However if you copy a mutable list, `copy` is indeed a new instance so it's garanteed there's no side-effects when mutating the original list:

```
var list = new ArrayList<String>();
var copy = List.copyOf(list);
System.out.println(list == copy);   // false
```

When creating immutable maps you don't have to create map entries yourself but instead pass keys and values as alternating arguments:

```
var map = Map.of("A", 1, "B", 2);
```

> *Immutable collections in Java 11 still use the same interfaces from the old Collection API. However if you try to modify an immutable collection by adding or removing elements, a `java.lang.UnsupportedOperationException` is thrown. Luckily Intellij IDEA warns via an inspection if you try to mutate immutable collections.*

Streams

Streams were introduced in Java 8 and now receive three new methods. `Stream.ofNullable` constructs a stream from a single element:

```
Stream.ofNullable(null)
    .count()   // 0
```

The methods `dropWhile` and `takeWhile` both accept a predicate to determine which elements to abandon from the stream:

```
Stream.of(1, 2, 3, 2, 1)
    .dropWhile(n -> n < 3)
    .collect(Collectors.toList());  // [3, 2, 1]

Stream.of(1, 2, 3, 2, 1)
    .takeWhile(n -> n < 3)
    .collect(Collectors.toList());  // [1, 2]
```

> *If you're not yet familiar with Streams you should read my Java 8 Streams Tutorial.*

Optionals

Optionals also receive a few quite handy new methods, e.g. you can now simply turn optionals into streams or provide another optional as fallback for an empty optional:

```
Optional.of("foo").orElseThrow();      // foo
Optional.of("foo").stream().count();  // 1
Optional.ofNullable(null)
    .or(() -> Optional.of("fallback"))
    .get();                           // fallback
```

One of the most basic classes `String` gets a few helper methods for trimming or checking whitespace and for streaming the lines of a string:

```
" ".isBlank();                // true
" Foo Bar ".strip();          // "Foo Bar"
" Foo Bar ".stripTrailing();  // " Foo Bar"
" Foo Bar ".stripLeading();   // "Foo Bar "
"Java".repeat(3);             // "JavaJavaJava"
"A\nB\nC".lines().count();    // 3
```

InputStreams

Last but not least `InputStream` finally gets a super useful method to transfer data to an `OutputStream`, a usecase that's very common when working with streams of raw data.

```
var classLoader = ClassLoader.getSystemClassLoader();
var inputStream = classLoader.getResourceAsStream("myFile.txt");
var tempFile = File.createTempFile("myFileCopy", "txt");
try (var outputStream = new FileOutputStream(tempFile)) {
    inputStream.transferTo(outputStream);
}
```

Other JVM features

These are the - in my opinion - most interesting language new API features when moving from Java 8 to 11. But the feature list doesn't end here. There's a lot more packed into the latest Java releases:

- Flow API for reactive programming
- Java Module System
- Application Class Data Sharing
- Dynamic Class-File Constants
- Java REPL (JShell)
- Flight Recorder
- Unicode 10
- G1: Full Parallel Garbage Collector
- ZGC: Scalable Low-Latency Garbage Collector

- ...

What's your favorite features? Let me know!

Where to go from here?

Many people (including me) are still using Java 8 in production. However as of the beginning of 2019 free support for JDK 8 ends. So this is a good time to migrate to Java 11 now. I wrote a migration guide how to move from Java 8 to 11 which hopefully helps you with your migration. You should also read my Java 8 and Stream API tutorials to learn more modern Java fundamentals. The source code of this tutorial is published on GitHub so feel free to play around with it (and leave a star if you like). You should also follow me on Twitter for more Java- and development-related stuff. Cheers!

Benjamin is Software Engineer, Full Stack Developer at Pondus, an excited runner and table foosball player. Get in touch on Twitter and GitHub.

Read More

Recent | All Posts | Java | JavaScript | Tutorials

Migrate Maven Projects to Java 11

Kotlin Sequence Tutorial

Integrating React.js into Existing jQuery Web Applications

Java 8 Concurrency Tutorial: Atomic Variables and ConcurrentMap