(/)

Java 11 Single File Source Code

Last modified: May 7, 2019

by Ganesh Pagade (https://www.baeldung.com/author/ganesh-pagade/)

Java (https://www.baeldung.com/category/java/) •

Java 11 (https://www.baeldung.com/tag/java-11/)

I just announced the new *Learn Spring* course, focused on the fundamentals of Spring 5 and Spring Boot 2:

>> CHECK OUT THE COURSE (/ls-course-start)

1. Introduction

JDK 11 (https://openjdk.java.net/projects/jdk/11/), which is the implementation of Java SE 11, released in September 2018.

In this tutorial, we'll cover the new Java 11 feature of launching single-file source-code programs.

2. Before Java 11

A single-file program is one where the program fits in a single source file.

Before Java 11, even for a single-file program, we had to follow a two-step process to run the program.

For example, if a file called *HelloWorld.java* contains a class called *HelloWorld* with a *main()* method, **we would have to first compile it:**

We use cookies to improve your experience with the site. To find out more, you can read the full Privacy and Cookie Policy (/privacy-policy)

Ok

```
$ javac HelloWorld.java
```

This would generate a class file that **we would have to run using the command:**

```
$ java HelloWorld
Hello Java 11!
```

Such programs are standard in the early stages of learning Java or when writing small utility programs. In this context, it's a bit ceremonial to have to compile the program before running it.

But, wouldn't it be great to just have a one-step process instead? Java 11 tries to address this, by allowing us to run such programs directly from the source.

3. Launching Single-File Source-Code Programs

Starting in Java 11, we can use the following command to execute a single-file program:

```
$ java HelloWorld.java
Hello Java 11!
```

Notice how we passed the Java source code file name and not the Java class to the *java* command.

The JVM compiles the source file into memory and then runs the first public *main()* method it finds.

We'll get compilation errors if the source file contains errors, but otherwise, it will run just as if we'd already compiled it.

4. Command-Line Options

The Java launcher introduced a new *source-file mode* to support this feature. The source-file mode is enabled if one of the following two conditions are true:

1. The first item on the command line followed by the JVM options is a file name with the *.java* extension
2. The command line contains the *–source* version option

If the file does not follow the standard naming conventions for Java source files, we need to use the *–source* option. We'll talk more about such files in the next section.

Any arguments placed after the name of the source file in the original command line are passed to the compiled class when it is executed.

For example, we have a file called *Addition.java* that contains an *Addition* class. This class contains a *main()* method that calculates the sum of its arguments:

We use cookies to improve your experience with the site. To find out more, you can read the full Privacy and Cookie Policy (/privacy-policy)

Ok

```
$ java Addition.java 1 2 3
```

Also, we can pass options likes *–class-path* before the file name:

```
$ java --class-path=/some-path Addition.java 1 2 3
```

Now, **we'll get an error if there is a class on the application classpath with the same name as the class we are executing**.

For example, let's say at some point during development, we compiled the file present in our current working directory using *javac*:

```
$ javac HelloWorld.java
```

We now have both *HelloWorld.java and HelloWorld.class* present in the current working directory:

```
$ ls
HelloWorld.class  HelloWorld.java
```

But, if we try to use the source-file mode, we'll get an error:

```
$ java HelloWorld.java
error: class found on application class path: HelloWorld
```

5. Shebang Files

It's common in Unix-derived systems, like macOS and Linux to use the "#!" directive to run an executable script file.

For example, a shell script typically starts with:

```
#!/bin/sh
```

We can then execute the script:

```
$ ./some_script
```

Such files are called "shebang files".

We can now execute Java single-file programs using this same mechanism.

If we add the following to the beginning of a file:

```
#!/path/to/java --source version
```

For example, let's add the following code in a file named *add*:

We use cookies to improve your experience with the site. To find out more, you can read the full Privacy and Cookie Policy (/privacy-policy)

Ok

```
#!/usr/local/bin/java --source 11

import java.util.Arrays;

public class Addition
{
    public static void main(String[] args) {
        Integer sum = Arrays.stream(args)
          .mapToInt(Integer::parseInt)
          .sum();

        System.out.println(sum);
    }
}
```

And mark the file as executable:

```
$ chmod +x add
```

Then, we can execute the file just like a script:

```
$ ./add 1 2 3
6
```

We can also explicitly use the launcher to invoke the shebang file:

```
$ java --source 11 add 1 2 3
6
```

The *–source* option is required even if it's already present in the file. The shebang in the file is ignored and is treated as a normal java file without the *.java* extension.

However**, we can't treat a *.java* file as a shebang file, even if it contains a valid shebang.** Thus the following will result in an error:

```
$ ./Addition.java
./Addition.java:1: error: illegal character: '#'
#!/usr/local/bin/java --source 11
^
```

One last thing to note about shebang files is that the directive makes the file platform-dependent. **The file will not be usable on platforms like Windows, which does not natively support it.**

6. Conclusion

In this article, we saw the new single file source code feature introduced in Java 11.

We use cookies to improve your experience with the site. To find out more, you can read the full Privacy and Cookie Policy (/privacy-policy)
Ok

As usual, code snippets can be found over on GitHub (https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-11).

I just announced the new *Learn Spring* course, focused on the fundamentals of Spring 5 and Spring Boot 2:

>> CHECK OUT THE COURSE (/ls-course-end)

Learning to "Build your API
with Spring"?

Enter your email address

>> Get the eBook

Comments are closed on this article!

We use cookies to improve your experience with the site. To find out more, you can read the full Privacy and Cookie Policy (/privacy-policy)

Ok

CATEGORIES

SPRING (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
HTTP CLIENT-SIDE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE COURSES (HTTPS://COURSES.BAELDUNG.COM)
JOBS (/TAG/ACTIVE-JOB/)
THE FULL ARCHIVE (/FULL_ARCHIVE)
WRITE FOR BAELDUNG (/CONTRIBUTION-GUIDELINES)
EDITORS (/EDITORS)
OUR PARTNERS (/PARTNERS)
ADVERTISE ON BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)

We use cookies to improve your experience with the site. To find out more, you can read the full Privacy and Cookie Policy (/privacy-policy)

Ok