



Adrian D. Finlay

@thewiprogrammer. Writer @hackernoon. Code, LOTS of it. Mangos, LOVE THEM! Barbering. Health. Travel. Business. & more! Network w/ me @ [adriandavid.me/network](https://adriandavid.me/network)

Jun 26 · 13 min read

## New Java 11 Feature: Launch Single-File Source-Code Programs

This november, the way you view scripting in Java will change. This change is the consequence of JEP 330: Launch Single-File Source-Code Programs. But first, some necessary context.



RON FRICKE LOOP GIF — GIPHY

## What is Scripting?

The truth is **we don't know**. I can hear some of you screaming from the mountain tops already. Some in strong agreement, others in disdain, and yet others, perhaps, in rebuke.

Why don't we know? Because like many tech terms (I'm looking at you "strongly typed"), **it does not have a consistent, universally accepted and clear definition.**

Some definitions of *script* include:

- "An **automated** series of instructions carried out in a specific order."—Oxford Dictionary
- "An executable section of code that **automates** a task"—Dictionary.com
- "Scripts are lists of commands executed by certain programs or scripting engines. They are usually text documents with instructions written using a scripting language. They are used to generate Web pages and to **automate** computer processes."—Techopedia

Noticed a commonality? *Automation*. This is perhaps the most common shared attribute among the various definitions of *script*. This most common shared attributes takes it's most practical significance, perhaps, in automating operating system tasks in fields like systems administration and the management of systems like web servers, which require periodic maintenance. For the purpose of this article, and to further establish context, I will take the liberty of providing my own definition of *script* as follows:

*A series of instructions designed to automate a task or series of tasks that is typically executed dynamically by an interpreter in a virtual environment.*

My definition looks an awful lot the like aforementioned definitions, save for a few additional details.

Firstly, unlike Oxford and Techopedia, I did not specify that the instructions need manifest as code written by a programming language. This is because scripting can be done without turing complete languages, which is generally considered the major qualifying criteria

in categorizing a language as a programming language. For example, think of CRON jobs, macros, and bash / korn shell scripts.

Secondly, you will notice that I also included the context of automation, which I believe to be the core idea and purpose in defining scripting as a unique term, separate from programming or some other term.

Thirdly, I go further than the aforementioned dictionaries in including that scripts are *typically* executed A) dynamically, that is to say that they are not compiled, B) by an interpreter that handles said dynamic execution, and C) within a virtual environment, which I believe to be more aligned with the modern context of scripting. These are, however, what I consider to be the typical case and as such is not expressly required by my definition of scripting.

*The clearest distinction between compilation and interpretation is that interpretation always implies execution whereas compilation is simply translation without execution.*

Furthermore, notice that there is some additional lingering vagueness involving scripting, although this part is perhaps not so hotly contested. This is the notion of *interpretation versus compilation*. Typically, Java is considered to be a hybrid of the two, although it is more often considered to be compiled. As evidence of this, notice the depiction of Java on the Wikipedia page for scripting:

“For example, it is uncommon to characterise Java as a scripting language because of its lengthy syntax and rules about which classes exist in which files, and it is not directly possible to execute Java interactively, because source files can only contain definitions that must be invoked externally by a host application or application launcher.”  
[2].

This is, however, as we will discover, not quite accurate anymore. Before JEP 330, the advent of Just-In-Time (JIT) compilation, Ahead-of-Time (AOT) compilation, the evolution of virtual machines, and the fact that java does not need to be translated into object code and executed immediately, are part of what made java tend to be considered as a compiled language as opposed to interpreted, in my

opinion. The clearest distinction between compilation and interpretation is that interpretation always implies execution whereas compilation is simply translation without the need for execution. Both processes, do, however, involve translating an abstraction into a lesser abstraction. There is a camp of thought that believes the difference between interpreted and compiled code is somewhat meaningless—After all, If Python is scripting C then is C scripting assembler?

Lastly, notice that I have made no statements regarding the length of a script. This is a sticky area. Scripts are generally thought of as short in nature, although scripts are rarely ever quantified by those referring to scripts as such and there is great danger in qualifying terms with other terms like “short” and “long” as they are intensely subjective and would only serve to make the definition of script all the more vague and inconsistent.

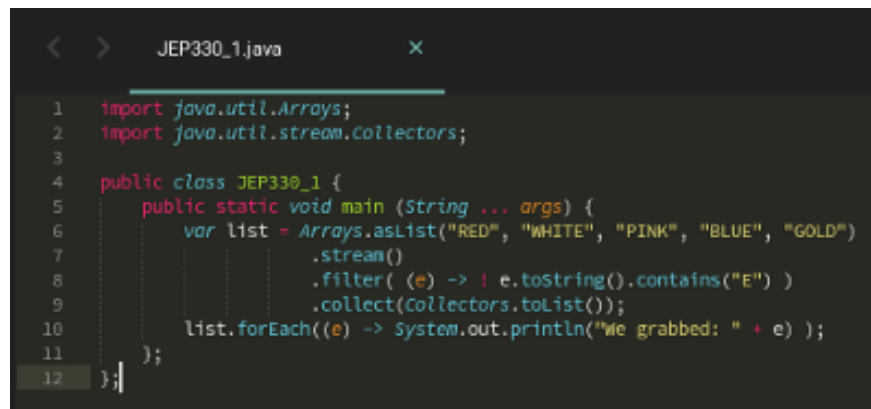
These debates aside, it is not, however, necessary to accept my definition, nor to form an opinion on the meaningfulness (or lack thereof) of the differences between compilation and interpretation. Suffice that one understand the basic context of scripting and the commonalities shared by these definitions, one may happily follow along. My definition is provided as the basis of what I will mean when I will later refer to scripting.

## **Now that that is over....the real stuff!**

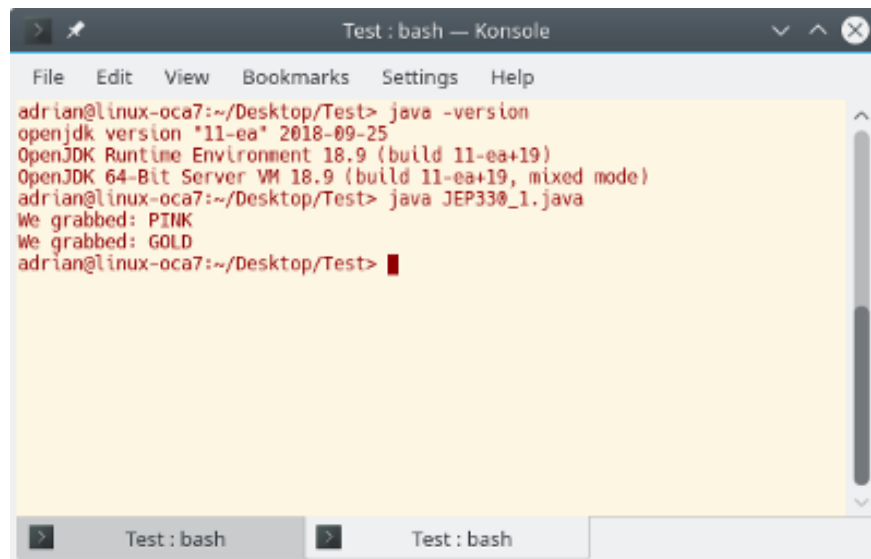
So what’s the hubbub all about in Java 11? Well, for the first time in Java history, we will know longer need the ceremony associated with compiled languages of compiling/linking and then running executable code. Now, one can run their code simply through invoking the java command! The basic syntax is as follows:

```
java MyScript.java
```

Sounds to good to be true? Try a basic hello world! Note that, you will, of course, need a Java SE 11 runtime environment. I will be using SUSE Linux and the openJDK EA Builds for this tutorial. This feature has been integrated into the JDK as of JDK 11 EA Build 17. You may grab it [here](#).



```
< > JEP330_1.java X
1  import java.util.Arrays;
2  import java.util.stream.Collectors;
3
4  public class JEP330_1 {
5      public static void main (String ... args) {
6          var list = Arrays.asList("RED", "WHITE", "PINK", "BLUE", "GOLD")
7              .stream()
8              .filter( (e) -> ! e.toString().contains("E") )
9              .collect(Collectors.toList());
10         list.forEach((e) -> System.out.println("We grabbed: " + e) );
11     };
12 }
```



```
> ✎ Test : bash — Konsole
File Edit View Bookmarks Settings Help
adrian@linux-oca7:~/Desktop/Test> java -version
openjdk version "11-ea" 2018-09-25
OpenJDK Runtime Environment 18.9 (build 11-ea+19)
OpenJDK 64-Bit Server VM 18.9 (build 11-ea+19, mixed mode)
adrian@linux-oca7:~/Desktop/Test> java JEP330_1.java
We grabbed: PINK
We grabbed: GOLD
adrian@linux-oca7:~/Desktop/Test> █
```

## This is nice and all, but why did they add it?....

Well, here are some hypotheses. If you follow the openJDK mailing list as well as community discussion (blogs, twitter, etc.) you will notice that one of the biggest barriers to using java as an introductory language is the baggage associated with syntactic ceremony. This makes sense. After all—which “hello world” program is easier to explain?

*In a way, the early learning stage is somewhat recursive—we begin learning a concept within the context of another concept we don't understand en route to understanding the originally unfamiliar*

*surrounding context, and we often repeat this process.*

```
print("Hello World")  
#Run in a python3 interpreter with the command "python"
```

or....

```
public class HelloWorld {  
    public static void main (String[] args) {  
        System.out.println("Hello World")  
    }  
}  
  
//javac HelloWorld.java  
//java HelloWorld
```

What's public? class? static? void? println? And so on.....

When instructing beginners, you want to get right down to the bottom of what you are trying to explain, without having to be surrounded by unnecessary details. In a way, the early learning stage is somewhat recursive—we begin learning a concept within the context of another concept we don't understand en route to understanding the originally unfamiliar surrounding context, and we often repeat this process.

Having to explain these identifiers distracts from the goal of simply teaching an individual how to print text to the console. While this syntactic baggage has not been relieved us (and probably never will), the ceremony of manually compiling and running has. This allows java to behave more like an interpreted language, although we will soon see that this is not exactly the case.

In my opinion, this is the second among recent efforts to try to maintain and grow the choice of java as an introductory language in education systems. For the first time, in many years, several institutions have switched away from java in favor of languages like python. It seems to me that the java camp is feeling the recent pressure of migration

towards python. There are several reasons for these shifts by institutions, and a lot of it has to do with ease of explanation. You can read more about this, [here](#). In addition, things like applets that were fancied by many academics have also gone out of style (deprecated) and JNLPS/Java WebStart have not picked up the traction that applets once had for educational use, so far.

In the end, nothing beats the horse's mouth: "Single-file programs—where the whole program fits in a single source file—are common in the early stages of learning Java, and when writing small utility programs. In this context, it is pure ceremony to have to compile the program before running it. In addition, a single source file may compile to multiple class files, which adds packaging overhead to the simple goal of "run this program"." [1]

## How it works, and Some Gotchas

JEP 330 adds a fourth mode of execution to the java command line tool—the ability to execute a class from source code. This execution mode is determined with the consideration of the presence of the `--source` option and the result of parsing the first non-option argument passed to the java tool.

In other words, java looks for the first `<FileName>.java`. **Notice we said file name and not class name!**(More on this later).

The source option is considered in the context of terminal/console scripts, as we will later examine.

There are two basic ways to script in Java:

1. Using the java command tool directly
2. Terminal/Command Line Scripts (bash, ksh, DOS, and so on.)

We have seen the first already, now let us examine the second.

## Shell Scripting with Java 11

Java 11 introduces support for scripting with traditional \*nix shebang files. My environment for this example will be BASH running on openSUSE. First, some rules.

1. You may not mix java code with your desired shell scripting language. (Imagine the mess this would create!)
2. Should you include VM options, you must specify the `--source` as the first option following the filename in the shebang file.
3. If you need to specify the source version of the file (obvious).
4. The shebang must be the first line of the file, of which the first two characters must strictly be ASCII character 0x23 followed by ASCII character 0x21 (`#!`).
5. “The name of the shebang file does not follow the standard naming conventions for Java source files.”[1] (**Note:** This does not appear to be implemented as of EA build +18, **please correct me if I am wrong!**)
6. Files ending in `.java` are not permitted to include the `#!` shebang.
7. java will split arguments for the `--source` option should it contain whitespace. You may not override this by using quotes to preserve whitespace.
8. The shebang line (the first line) is ignored when evaluating source code against the JLS.

Now that we’ve got the rules out of the way, let’s dig into some syntax. To invoke a bash script programmatically, we use the following syntax.

```
#The file should be saved as follows
<FileName>.sh #Or just <FileName>

#The file needs to be marked as executable as follows
chmod +x <Filename>.<Extension>

#The file can be invoked as follows
./FileName.sh #Or just ./<FileName>

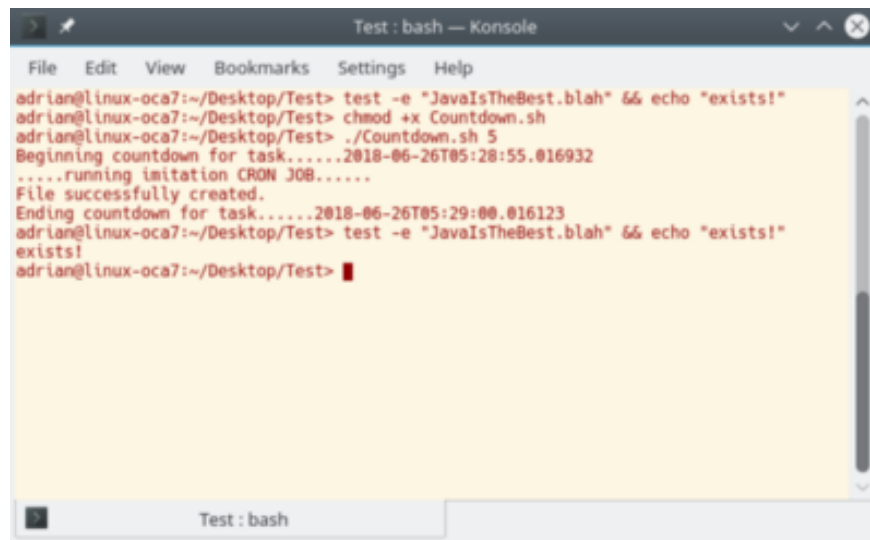
#The file can also be invoked as follows
java -Dtrace=true --source XX -<Options> <FileName>
<Argument List>

#For example
java -Dtrace=true --source 11 Countdown.sh
```



A typical cron job might perhaps be reimplemented like this:

```
1  #!/home/adrian/java/openjdk-11/bin/java --source 11
2
3  //Simple CRON Job reimplemented using Java APIs!
4
5  import java.io.File;
6  import java.util.Date;
7  import java.util.Timer;
8  import java.util.Calendar;
9  import java.util.TimerTask;
10 import java.time.LocalDateTime;
11
12 public class Countdown {
13
14     private static class DeleteClassFiles extends TimerTask {
15         public void run () {
16             System.out.println(".....running");
17
18             try {
19                 //CRON JOB - Create File
20                 if (new File("JavaIsTheBest").exists()) {
21                     System.out.println("File exists");
22                 } else {
23                     if (new File("JavaIsTheBest").createNewFile()) {
24                         System.out.println("File created");
25                     } else {
26                         System.out.println("File not created");
27                     }
28                 }
29             } catch (java.io.IOException ioe) {
30                 System.out.println("Ending count");
31                 System.exit(0);
32             }
33         }
34     }
35 }
```



```
Test : bash — Konsole
File Edit View Bookmarks Settings Help
adrian@linux-oca7:~/Desktop/Test> test -e "JavaIsTheBest.blah" && echo "exists!"
adrian@linux-oca7:~/Desktop/Test> chmod +x Countdown.sh
adrian@linux-oca7:~/Desktop/Test> ./Countdown.sh 5
Beginning countdown for task.....2018-06-26T05:28:55.016932
.....running imitation CRON JOB.....
File successfully created.
Ending countdown for task.....2018-06-26T05:29:00.016123
adrian@linux-oca7:~/Desktop/Test> test -e "JavaIsTheBest.blah" && echo "exists!"
exists!
adrian@linux-oca7:~/Desktop/Test>
```

## Gotcha #1—No outside classes, Single-File Programs only!

This feature single-file programs, that means that you may not invoke other classes in files other than the file you are executing. Therefore if we added `new SampleCustomClass();`, where `SampleCustomClass` is a class which we defined and is visible to our program, to the `main()` method of our original source file, execution would fail.

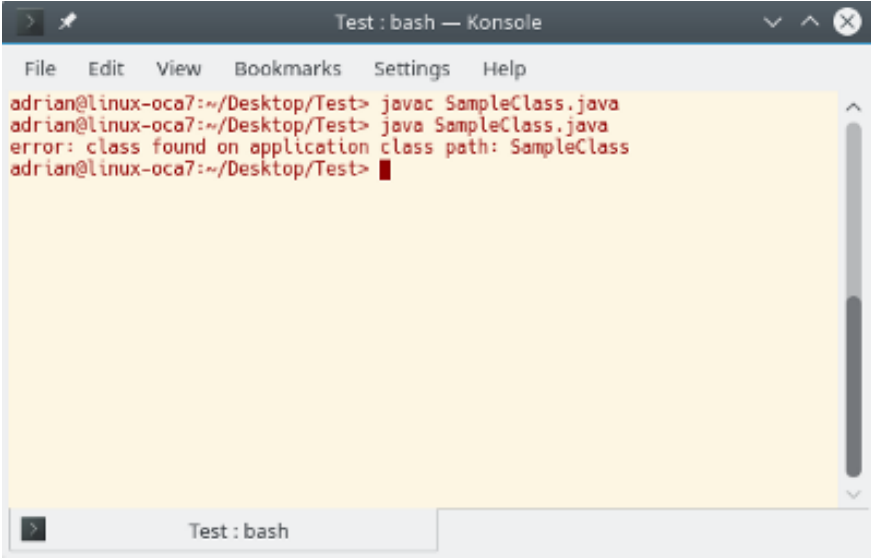
Remember, the goal of this feature is not to replace `javac`. It is merely to make it more convenient, especially in the context of beginning programming to achieve the goal of “run this program” [1]. This also eliminates the production of multiple classes in the root (or specified) directory.

## Gotcha #2—No class files available for you!

Notice that the invocation of the `java` tool in this manner does not provide any class files you can see in your working directory. That is because, “the effect is as if the source file is compiled into memory”, per JEP 3330[1].

## Gotcha #3—If existing class file exists in classpath, you’re forced to use it

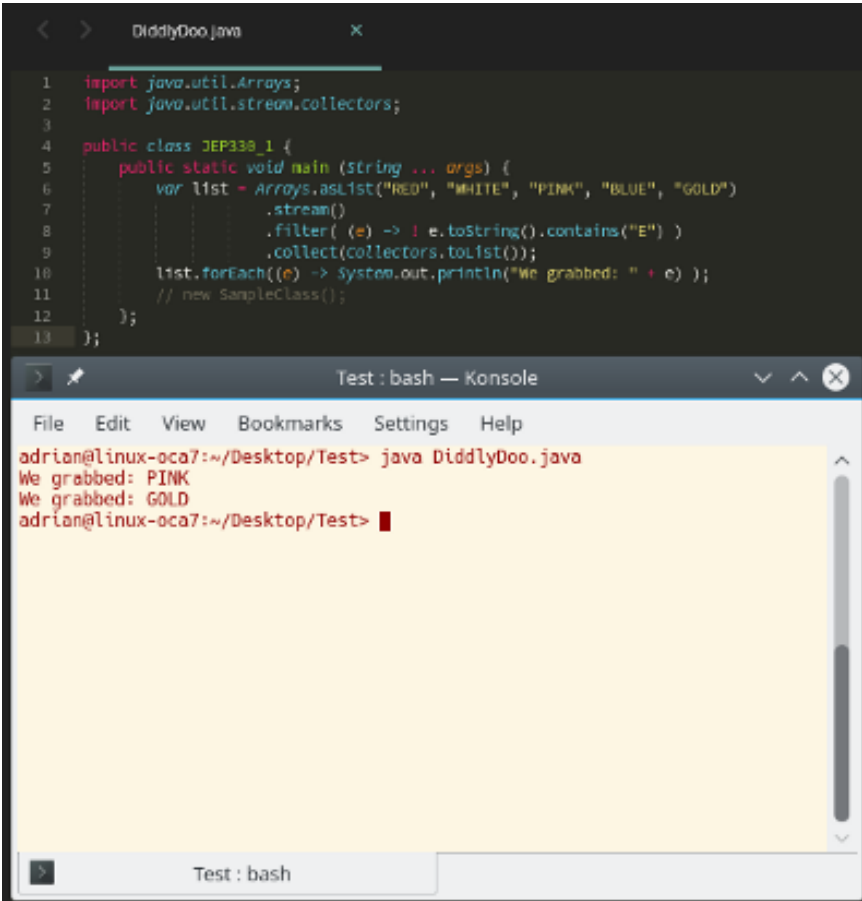
For a file, say `SampleClass.java`, should an existing class file, say, `SampleClass.class` exist, you may not invoke the `java` tool to run your source file in source-file mode. You can still, of course, invoke the class via the standard method, as you always have.



```
Test : bash — Konsole
File Edit View Bookmarks Settings Help
adrian@linux-oca7:~/Desktop/Test> javac SampleClass.java
adrian@linux-oca7:~/Desktop/Test> java SampleClass.java
error: class found on application class path: SampleClass
adrian@linux-oca7:~/Desktop/Test>
```

## Gotcha #4—File name, not class name!

Breaking with over twenty years of tradition, the java tool considers the **name of the file** and **NOT** the name of the class in executing source files. Notice the following example:

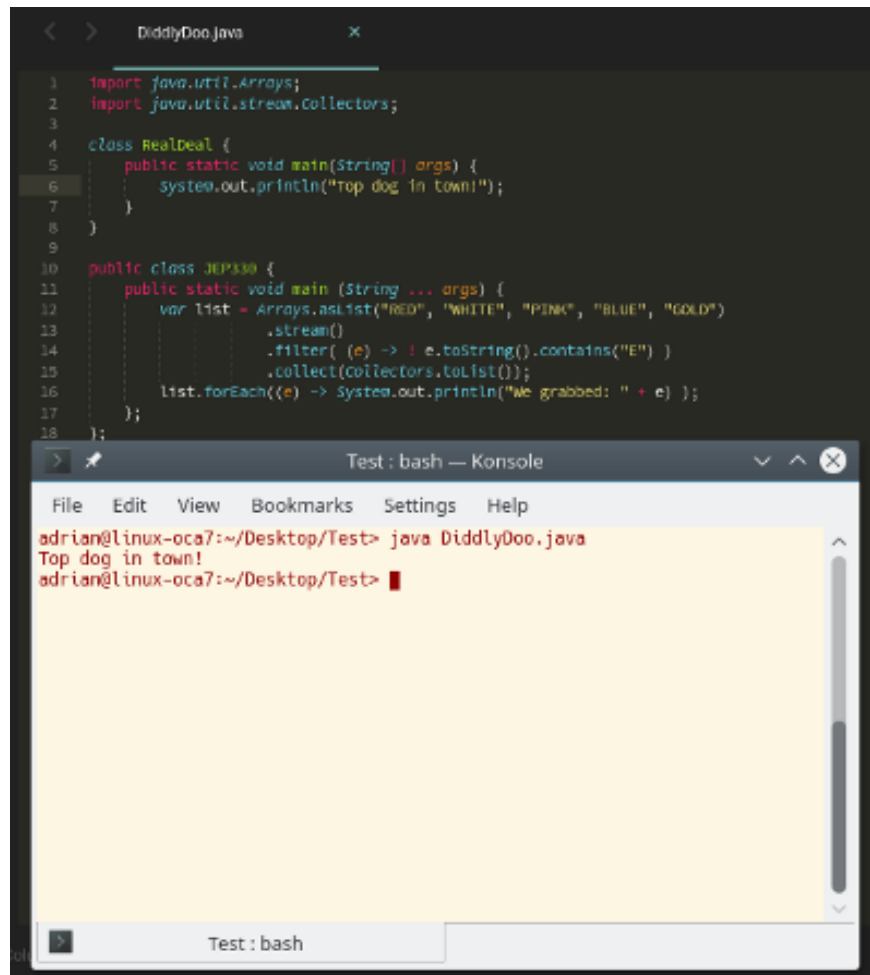


```
DiddlyDoo.java
1 import java.util.Arrays;
2 import java.util.stream.Collectors;
3
4 public class JEP338_1 {
5     public static void main (String ... args) {
6         var list = Arrays.asList("RED", "WHITE", "PINK", "BLUE", "GOLD")
7             .stream()
8             .filter( (e) -> ! e.toString().contains("E") )
9             .collect(Collectors.toList());
10         list.forEach((e) -> System.out.println("We grabbed: " + e ));
11         // new SampleClass();
12     }
13 }

Test : bash — Konsole
File Edit View Bookmarks Settings Help
adrian@linux-oca7:~/Desktop/Test> java DiddlyDoo.java
We grabbed: PINK
We grabbed: GOLD
adrian@linux-oca7:~/Desktop/Test>
```

## Gotcha #5—First class in the file, not matching file-class names!

In tandem with Gotcha #4, the class loader no longer determines the class to be executed by matching file name and class name. **The first class in the file is the one that will be run!**

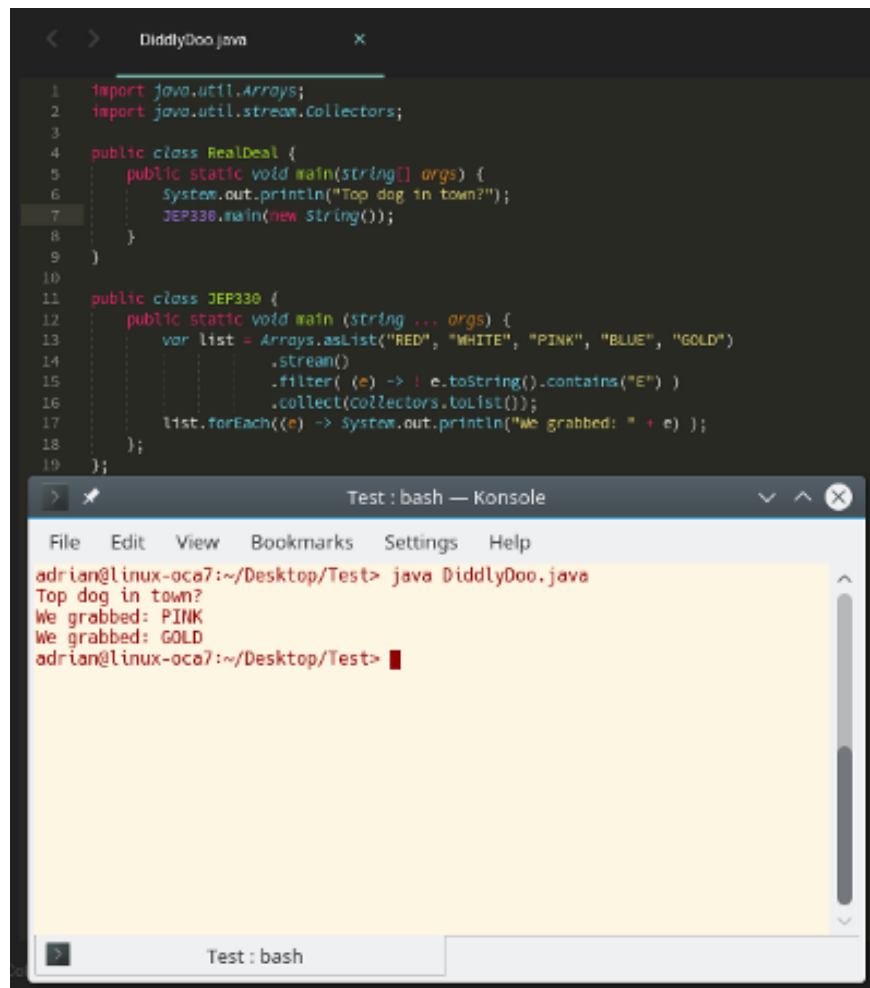


```
< > DiddlyDoo.java x
1 import java.util.Arrays;
2 import java.util.stream.Collectors;
3
4 class RealDeal {
5     public static void main(String[] args) {
6         System.out.println("Top dog in town!");
7     }
8 }
9
10 public class JEP130 {
11     public static void main (String ... args) {
12         var list = Arrays.asList("RED", "WHITE", "PINK", "BLUE", "GOLD")
13             .stream()
14             .filter( (e) -> ! e.toString().contains("E") )
15             .collect(Collectors.toList());
16         list.forEach((e) -> System.out.println("We grabbed: " + e ));
17     };
18 };
```

```
Test: bash — Konsole
File Edit View Bookmarks Settings Help
adrian@linux-oca7:~/Desktop/Test> java DiddlyDoo.java
Top dog in town!
adrian@linux-oca7:~/Desktop/Test>
```

## Gotcha #6—No limits on public files in a source file 🤖

While javac cares about the amount of public classes in a source file, java couldn't care less! Notice that we know have two public classes and a totally irrelevant file name.



The screenshot shows an IDE window titled 'DiddlyDoo.java'. The code defines two classes: `RealDeal` and `JEP338`. `RealDeal` has a `main` method that prints 'Top dog in town?' and calls `JEP338.main`. `JEP338` has a `main` method that uses `Arrays.asList` to create a list of colors, filters out those containing 'E', and prints each remaining color. Below the code is a terminal window titled 'Test: bash — Konsole'. It shows the command `java DiddlyDoo.java` being executed, with the output: 'Top dog in town?', 'We grabbed: PINK', and 'We grabbed: GOLD'.

```
1 import java.util.Arrays;
2 import java.util.stream.Collectors;
3
4 public class RealDeal {
5     public static void main(String[] args) {
6         System.out.println("Top dog in town?");
7         JEP338.main(new String());
8     }
9 }
10
11 public class JEP338 {
12     public static void main (String ... args) {
13         var list = Arrays.asList("RED", "WHITE", "PINK", "BLUE", "GOLD")
14             .stream()
15             .filter( (e) -> ! e.toString().contains("E") )
16             .collect(Collectors.toList());
17         list.forEach((e) -> System.out.println("We grabbed: " + e));
18     };
19 }
```

```
adrian@linux-oca7:~/Desktop/Test> java DiddlyDoo.java
Top dog in town?
We grabbed: PINK
We grabbed: GOLD
adrian@linux-oca7:~/Desktop/Test>
```

## Gotcha #7—You may not pass certain compiler specific arguments

Note that the standard command line options such as `--source` and most command line arguments that you pass to your application are still available for use in the fashion you are accustomed to. That is, preceding the file name. Command line argument files work in the expected fashion as well.

The real kicker here is that arguments like `-Werror` or `-nowarn` that you can pass to `javac`, may not be passed (or recognized for that matter) by the java tool.

## Gotcha #8—Beware of funky package conflicts!!

One should never write code like this. Seriously! However, who am I to say what may happen. Notice the following directory structure:

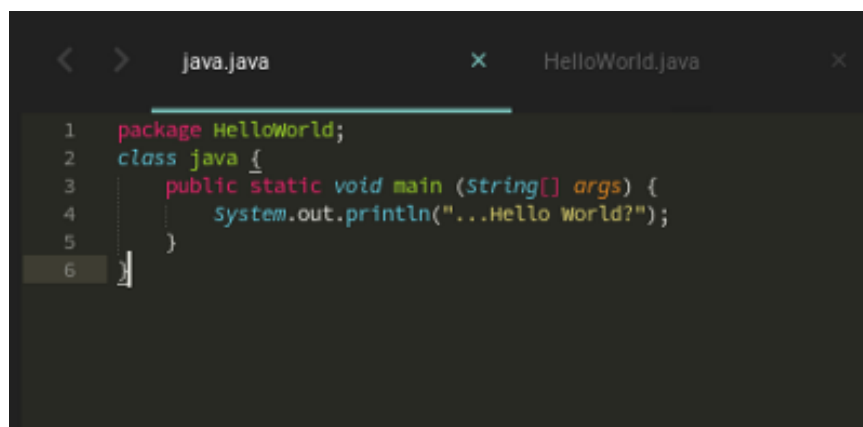
A terminal window titled "Test : bash — Konsole" with a menu bar (File, Edit, View, Bookmarks, Settings, Help). The prompt is "adrian@linux-oca7:~/Desktop/Test>". The command "tree" has been executed, showing a directory tree. The files listed are: Countdown, Countdown.sh, DiddlyDoo.java, HelloWorld (a directory containing java.class and java.java), HelloWorld.java, JavaCompiler.java, JEP330, JEP\_330, JEP330\_2.java, and JEP330\_3.java.

```
> ✂ Test : bash — Konsole
File Edit View Bookmarks Settings Help
adrian@linux-oca7:~/Desktop/Test> tree
.
├── Countdown
├── Countdown.sh
├── DiddlyDoo.java
├── HelloWorld
│   ├── java.class
│   └── java.java
├── HelloWorld.java
├── JavaCompiler.java
├── JEP330
├── JEP_330
├── JEP330_2.java
└── JEP330_3.java
```

Now notice these two files.

A code editor window with two tabs: "java.java" and "HelloWorld.java". The "HelloWorld.java" tab is active. It contains the following code:

```
1 class HelloWorld {
2     public static void main (String[] args) {
3         System.out.println("...Hello World!?");
4     }
5 }
```

A code editor window with two tabs: "java.java" and "HelloWorld.java". The "java.java" tab is active. It contains the following code:

```
1 package HelloWorld;
2 class java {
3     public static void main (String[] args) {
4         System.out.println("...Hello World!?");
5     }
6 }
```

See the problem? It's a nasty one. What happens when you run `java HelloWorld.java` ? Is the first file run or the second file?

**The answer: The one in the current directory is run.** In other words, java is **no longer referencing the class file in the HelloWorld package**, instead, it will load the `HelloWorld.java` file from source. You were warned, beware!

## Some Notes on behind-the-scenes details

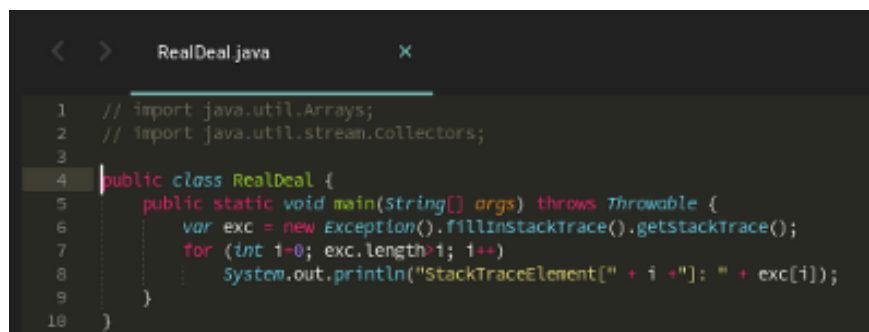
### How java does the magic

Behind the scenes, java will perform as if the following is invoked[1]:

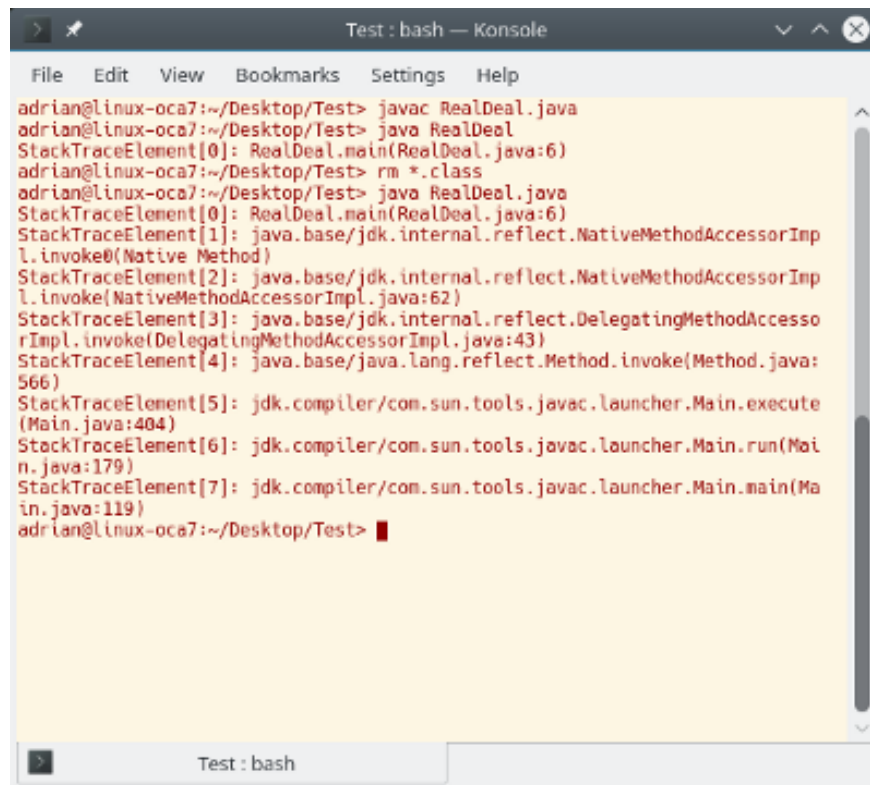
```
java [VM args] \
  -m jdk.compiler/<source-launcher-implementation-
  class> \
  Foo.java [program args]
```

Using the classes available in the `jdk.Compiler` module, java will bootstrap the application in memory by invoking the `javac` compiler programmatically and using a custom classloader to run the class files persisted in memory in accordance with the aforementioned rules.

In the event that an exception is thrown from your code, the exception is passed back to the launcher and displayed as if the launcher threw the exception itself. In other words, in the implementation, the stack trace leading up to your application's source code is removed! This is to accommodate the more natural expectation of a stack trace that you would be used to had you compiled the class and ran it with the traditional method. To demonstrate this, look at the differences below.



```
< > RealDeal.java x
1 // import java.util.Arrays;
2 // import java.util.stream.Collectors;
3
4 public class RealDeal {
5     public static void main(String[] args) throws Throwable {
6         var exc = new Exception().fillInStackTrace().getStackTrace();
7         for (int i=0; exc.length>i; i++)
8             System.out.println("StackTraceElement[" + i + "]: " + exc[i]);
9     }
10 }
```



```
Test : bash — Konsole
File Edit View Bookmarks Settings Help
adrian@linux-oca7:~/Desktop/Test> javac RealDeal.java
adrian@linux-oca7:~/Desktop/Test> java RealDeal
StackTraceElement[0]: RealDeal.main(RealDeal.java:6)
adrian@linux-oca7:~/Desktop/Test> rm *.class
adrian@linux-oca7:~/Desktop/Test> java RealDeal.java
StackTraceElement[0]: RealDeal.main(RealDeal.java:6)
StackTraceElement[1]: java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
StackTraceElement[2]: java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
StackTraceElement[3]: java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
StackTraceElement[4]: java.base/java.lang.reflect.Method.invoke(Method.java:566)
StackTraceElement[5]: jdk.compiler/com.sun.tools.javac.launcher.Main.execute(Main.java:404)
StackTraceElement[6]: jdk.compiler/com.sun.tools.javac.launcher.Main.run(Main.java:179)
StackTraceElement[7]: jdk.compiler/com.sun.tools.javac.launcher.Main.main(Main.java:119)
adrian@linux-oca7:~/Desktop/Test>
```

## Miscellaneous implementation details.....for the curious

I will not list them all, after all, there is the JEP 330 docs for that, however I will list the ones that deviate from standard behavior or that we have not touched on earlier.

- Irrelevant options passed to the VM are ignored or rejected.
- Annotation processing is disabled. (**Note:** This does not appear to be implemented as of EA build +19 as I have got annotation processing work).
- Source files other than the single file are ignored and not visible in the classpath.
- The value passed with `--source` argument are passed on to the compiler for `--release` as well.
- The source file is enclosed in an unnamed module.
- Classes in the Application Classpath cannot refer to classes declared in the source file. (**Note:** This does not appear to be implemented as of EA build +18.)



## Closing Thoughts

Did you notice something? Did you notice that the ceremony of compiling was merely done in the background? Does this still qualify as **real** scripting? Is the difference that profound?

Like most interpretations of vague terms, it will depend on who you ask. The only thing that is likely commonly agreed among minds is that Java has come *closer* to possessing scripting abilities.



Looney Tunes Ending

## Works Cited

[1]—JEP 330: Local-Variable Type Inference