OpenJDK

Installing
Contributing
Sponsoring
Developers' Guide
Vulnerabilities
JDK GA/EA Builds

Mailing lists
Wiki · IRC

Bylaws · Census
Legal

Workshop

JEP Process

Source code
Mercurial
GitHub

Tools
Git
jtreg harness

Groups
(overview)
Adoption
Build
Client Libraries
Compatibility & Specification Review
Compiler
Conformance
Core Libraries
Governing Board
HotSpot
IDE Tooling & Support
Internationalization
JMX
Members
Networking
Porters
Quality
Security
Serviceability
Vulnerability
Web

Projects
(overview, archive)
Amber
Babylon
CRaC
Caciocavallo
Closures
Code Tools
Coin
Common VM Interface
Compiler Grammar
Detroit
Developers' Guide
Device I/O
Duke
Galahad
Graal
IcedTea
JDK 7
JDK 8
JDK 8 Updates
JDK 9
JDK (..., 21, 22, 23)
JDK Updates
JavaDoc.Next
Jigsaw
Kona
Kulla
Lambda
Lanai
Leyden
Lilliput
Locale Enhancement
Loom
Memory Model Update
Metropolis
Mission Control
Multi-Language VM
Nashorn
New I/O
OpenJFX
Panama
Penrose
Port: AArch32
Port: AArch64
Port: BSD
Port: Haiku
Port: Mac OS X
Port: MIPS
Port: Mobile
Port: PowerPC/AIX
Port: RISC-V
Port: s390x
Portola
SCTP
Shenandoah
Skara
Sumatra
Tiered Attribution
Tsan
Type Annotations
Valhalla
Verona
VisualVM
Wakefield
Zero
ZGC

ORACLE

JEP 458: Launch Multi-File Source-Code Programs

Owner	Ron Pressler
Type	Feature
Scope	JDK
Status	Closed / Delivered
Release	22
Component	tools / launcher
Discussion	compiler dash dev at openjdk dot org
Effort	S
Relates to	JEP 330: Launch Single-File Source-Code Programs
Reviewed by	Alex Buckley, Brian Goetz
Endorsed by	Brian Goetz
Created	2023/03/17 10:17
Updated	2023/12/05 18:43
Issue	8304400

Summary

Enhance the `java` application launcher to be able to run a program supplied as multiple files of Java source code. This will make the transition from small programs to larger ones more gradual, enabling developers to choose whether and when to go to the trouble of configuring a build tool.

Non-Goals

- It is not a goal to launch multi-file source-code programs via the "shebang" mechanism. Only single-file source-code programs can be launched via that mechanism.
- It is not a goal to ease the use of external library dependencies in source-code programs. That may be the subject of a future JEP.

Motivation

The Java programming language excels for writing large, complex applications developed and maintained over many years by large teams. Still, even large programs start small. In the early stages, developers tinker and explore and do not care about deliverable artifacts; the project's structure may not yet exist, and once it emerges, it changes frequently. Fast iteration and radical change are the order of the day. Several features to assist with tinkering and exploration have been added to the JDK in recent years, including JShell (an interactive shell for playing with snippets of code) and a simple web server (for quick prototyping of web apps).

In JDK 11, JEP 330 enhanced the `java` application launcher to be able to run `.java` source files directly, without an explicit compilation step. For example, suppose the file `Prog.java` declares two classes:

```
class Prog {
    public static void main(String[] args) { Helper.run(); }
}

class Helper {
    static void run() { System.out.println("Hello!"); }
}
```

Then running

```
$ java Prog.java
```

compiles both classes in memory and executes the `main` method of the first class declared in that file.

This low-ceremony approach to running a program has a major limitation: All of the source code of the program must be placed in a single `.java` file. To work with more than one `.java` file, developers must return to compiling source files explicitly. For experienced developers, this often entails creating a project configuration for a build tool, but shifting from amorphous tinkering to formal project structure is irksome when trying to get ideas and experiments to flow smoothly. For beginning developers, the transition from a single `.java` file to two or more files requires an even starker phase change: They must pause their learning of the language and learn to operate `javac`, or learn a third-party build tool, or learn to rely on the magic of an IDE.

It would be better if developers could defer the project-setup stage until they understand more about the shape of the project, or even avoid setup altogether when quickly hacking and then throwing away a prototype. Some simple programs might remain in source form forever. This motivates enhancing the `java` launcher to be able to run programs that have grown beyond a single `.java` file, but without forcing an explicit compilation step. The traditional edit/build/run cycle becomes simply edit/run. Developers can decide for themselves when it is time to set up a build process rather be forced to do so by the limitations of the tooling.

Description

We enhance the `java` launcher's source-file mode to be able to run a program supplied as multiple files of Java source code.

For example, suppose a directory contains two files, `Prog.java` and `Helper.java`, where each file declares a single class:

```
// Prog.java
class Prog {
    public static void main(String[] args) { Helper.run(); }
}

// Helper.java
class Helper {
    static void run() { System.out.println("Hello!"); }
}
```

Running `java Prog.java` compiles the `Prog` class in memory and invokes its `main` method. Because code in this class refers to the class `Helper`, the launcher finds the `Helper.java` file in the filesystem and compiles its class in memory. If code in class `Helper` refers to some other class, e.g., `HelperAux`, then the launcher finds `HelperAux.java` and compiles that, too.

When classes in different `.java` files refer to each other, the `java` launcher does not guarantee any particular order or timing for the compilation of the `.java` files. It is possible, for example, for the launcher to compile `Helper.java` before `Prog.java`. Some code may be compiled before the program starts executing while other code may be compiled lazily, on the fly. (The process of compiling and executing source-file programs is described in detail below.)

Only `.java` files whose classes are referenced by the program are compiled. This allows developers to play with new versions of code without worrying that old versions will be compiled accidentally. For example, suppose the directory also contains `OldProg.java`, whose older version of the `Prog` class expects the `Helper` class to have a method named `go` rather than `run`. The presence of `OldProg.java`, with its latent error, is immaterial when running `Prog.java`.

Multiple classes can be declared in one `.java` file, and are all compiled together. Classes co-declared in a `.java` file are preferred to classes declared in other `.java` files. For example, suppose the file `Prog.java` above is expanded to declare a class `Helper`, despite a class of that name already being declared in `Helper.java`. When code in `Prog.java` refers to `Helper`, the class that is co-declared in `Prog.java` is used; the launcher will not search for the file `Helper.java`.

Duplicate classes in source-code programs are prohibited. That is, two declarations of a class with the same name in either the same `.java` file, or across different `.java` files that form part of the program, are not permitted. Suppose that, after some edits, `Prog.java` and `Helper.java` end up as shown below, with the class `Aux` accidentally declared in both:

```
// Prog.java
class Prog {
    public static void main(String[] args) { Helper.run(); Aux.cleanup(); }
}
class Aux {
    static void cleanup() { ... }
}

// Helper.java
class Helper {
    static void run() { ... }
}
class Aux {
    static void cleanup() { ... }
}
```

Running `java Prog.java` compiles the `Prog` and `Aux` classes in `Prog.java`, invokes the `main` method of `Prog`, and then — due to `main`'s reference to `Helper` — finds `Helper.java` and compiles its classes `Helper` and `Aux`. The duplicate declaration of `Aux` in `Helper.java` is not permitted, so the program stops and the launcher reports an error.

The source-file mode of the `java` launcher is triggered by passing it the name of a single `.java` file. If additional filenames are given, they become arguments to its `main` method. For example, `java Prog.java Helper.java` results in an array containing the string "`Helper.java`" being passed as an argument to the `main` method of the `Prog` class.

Using pre-compiled classes

Programs that depend on libraries on the class path or the module path can also be launched from source files. For example, suppose a directory contains two small programs plus a helper class, alongside some library JAR files:

```
Prog1.java
Prog2.java
Helper.java
library1.jar
library2.jar
```

You can quickly run these programs by passing `--class-path '*'` to the `java` launcher:

```
$ java --class-path '*' Prog1.java
$ java --class-path '*' Prog2.java
```

Here the `'*'` argument to the `--class-path` option puts all the JAR files in the directory on the class path; the asterisk is quoted to avoid expansion by the shell.

As you continue to experiment, you might find it more convenient to put the JAR files in a separate `libs` directory, in which case `--class-path 'libs/*'` will make them available. You can start thinking about producing a packaged deliverable, probably with the help of a build tool, only later, as the project takes shape.

How the launcher finds source files

The `java` launcher requires the source files of a multi-file source-code program to be arranged in the usual directory hierarchy in which directory structure follows package structure, starting with a root directory that is computed as described below. This means that:

- Source files in the root directory must declare classes in the unnamed package, and
- Source files in a directory `foo/bar` under the root directory must declare classes in the named package `foo.bar`.

For example, suppose a directory contains `Prog.java`, which declares classes in the unnamed package, and a subdirectory `pkg`, where `Helper.java` declares the class `Helper` in the package `pkg`:

```
// Prog.java
class Prog {
    public static void main(String[] args) { pkg.Helper.run(); }
}

// pkg/Helper.java
package pkg;
class Helper {
    static void run() { System.out.println("Hello!"); }
}
```

Running `java Prog.java` causes `Helper.java` to be found in the `pkg` subdirectory and compiled in memory, resulting in the class `pkg.Helper` that is needed by code in class `Prog`.

If `Prog.java` declared classes in a named package, or `Helper.java` declared classes in a package other than `pkg`, then `java Prog.java` would fail.

The `java` launcher computes the *root* of the source tree from the package name and filesystem location of the initial `.java` file. For `java Prog.java`, the initial file is `Prog.java` and it declares a class in the unnamed package, so the root of the source tree is the directory containing `Prog.java`. On the other hand, if `Prog.java` declares a class in the named package `a.b.c` then it must be placed in the corresponding directory in the hierarchy:

```
dir/
    a/
      b/
        c/
          Prog.java
```

It must also be launched by running `java dir/a/b/c/Prog.java`. In this case, the root of the source tree is `dir`.

If `Prog.java` had declared its package to be `b.c`, then the root of the source tree would have been `dir/a`; if it had declared the package `c`, then the root would have been `dir/a/b`, and if it had declared no package, then the root would have been `dir/a/b/c`. The program will fail to launch if `Prog.java` declares some other package, e.g. `p`, that does not correspond to a suffix of the file's path in the filesystem.

A minor but incompatible change

If, in the above example, `Prog.java` declared classes in a different named package then `java a/b/c/Prog.java` would fail. This is a change in the behavior of the `java` launcher's source-file mode.

In past releases, the launcher's source-file mode was permissive about which package, if any, was declared in a `.java` file at a given location: `java a/b/c/Prog.java` would succeed as long as `Prog.java` was found in `a/b/c`, regardless of any package declaration in the file. It is unusual for a `.java` file to declare classes in a named package without that file residing in the corresponding directory in the hierarchy, so the impact of this change is likely to be limited. If the package name is not important then the fix is to remove the package declaration from the file.

Modular source-code programs

In the examples shown thus far, the classes compiled from `.java` files have resided in the unnamed module. If the root of the source tree contains a `module-info.java` file, however, then the program is considered to be modular and the classes compiled from `.java` files in the source tree reside in the named module declared in `module-info.java`.

Programs that make use of modular libraries in the current directory can be run like so:

```
$ java -p . pkg/Prog1.java
$ java -p . pkg/Prog2.java
```

Alternatively, if the modular JAR files are in a `libs` directory then `-p libs` will make them available.

Launch-time semantics and operation

Since JDK 11, the launcher's source-file mode has worked as if

```
java <other options> --class-path <path> <.java file>
```

is informally equivalent to

```
javac <other options> -d <memory> --class-path <path> <.java file>
java  <other options> --class-path <memory>:<path> <first class in .java file>
```

With the ability to launch multi-file source-code programs, source-file mode now works as if

```
java <other options> --class-path <path> <.java file>
```

is informally equivalent to

```
javac <other options> -d <memory> --class-path <path> --source-path <root> <.java file>
java <other options> --class-path <memory>:<path> <launch class of .java file>
```

where `<root>` is the computed root of the source tree as defined earlier and `<launch class of .java file>` is the *launch class* of the `.java` file as defined below. (The use of `--source-path` indicates to `javac` that classes mentioned in the initial `.java` file may refer to classes declared in other `.java` files in the source tree. Classes co-located in a `.java` file are preferred to classes located in other `.java` files; for example, invoking `javac --source-path dir dir/Prog.java` will not compile `Helper.java` if `Prog.java` declares the class `Helper`.)

When the `java` launcher runs in source-file mode (e.g., `java Prog.java`) it takes the following steps:

1. If the file begins with a "shebang" line, that is, a line that starts with `#!`, then the source path passed to the compiler is empty so that no other source files will be compiled. Proceed to step 4.
2. Compute the directory which is the root of the source tree.
3. Determine the module of the source-code program. If a `module-info.java` file exists in the root then its module declaration is used to define a named module that will contain all the classes compiled from `.java` files in the source tree. If `module-info.java` does not exist then all the classes compiled from `.java` files will reside in the unnamed module.
4. Compile all the classes in the initial `.java` file, and possibly other `.java` files which declare classes referenced by code in the initial file, and store the resulting `class` files in an in-memory cache.
5. Determine the *launch class* of the initial `.java` file. If the first top level class in the initial file declares a standard `main` method (`public static void main(String[])` or other standard `main` entry points as defined in JEP 463), then that class is the launch class. Otherwise, if another top level class in the initial file declares a standard `main` method and has same name as the file, that class is the launch class. Otherwise, there is no launch class, and the launcher reports an error and stops.
6. Use a custom class loader to load the launch class from the in-memory cache, then invoke the standard `main` method of that class.

The procedure in step 5 for choosing the launch class preserves compatibility with JEP 330 and ensures that the same `main` method is used when a source program grows from one file to multiple files. It also ensures that "shebang" files continue to work, since the name of the class declared in such a file might not match the name of the file. Finally, it maintains an experience as close as possible to that of launching a program compiled with `javac`, so that when a source program grows to the point that it is desirable to run `javac` explicitly and execute the `class` files, the same launch class can be used.

When the custom class loader in step 6 is invoked to load a class — either the launch class or any other class that needs to be loaded while running the program — the loader performs a search that mimics the order of `javac`'s `-Xprefer:source` option at compile time. In particular, if a class exists both in the source tree (declared in a `.java` file) and on the class path (in a `.class` file) then the class in the source tree is preferred. The loader's search algorithm for a class named `C` is:

1. If a class file for `C` is found in the in-memory cache then the loader defines the cached class file to the JVM, and loading of `C` is complete.
2. Otherwise, the loader delegates to the application class loader to search for a class file for `C` that is exported by a named module which is read by the module of the source-code program and, also, is present on the module path or in the JDK run-time image. (The unnamed module, in which the source-code program may reside, reads a default set of modules in the JDK run-time image.) If found, loading of `C` is completed by the application class loader.
3. Otherwise, the loader searches for a `.java` file whose name matches the name of the class (or the enclosing class if the requested class is a member class), i.e. `C.java`, located in the directory corresponding to the package of the class. If found, all the classes declared in the `.java` file are compiled. If compilation succeeds then the resulting class files are stored in the in-memory cache, the loader defines the class `C` to the JVM using the cached class file, and loading of `C` is complete. If compilation fails then the launcher reports the error and terminates with a non-zero exit status.

   When compiling `C.java`, the launcher may choose to eagerly to compile other `.java` files that declare classes referenced by `C.java`, and store the resulting class files in the in-memory cache. This choice is based on heuristics that may change between JDK releases.
4. Otherwise, if the source-code program resides in the unnamed module, the loader delegates to the application class loader to search for a class file for `C` on the class path. If found then loading of `C` is completed by the application class loader.
5. Otherwise, a class named `C` cannot be found, and the loader throws a `ClassNotFoundException`.

Classes loaded from the class path or the module path cannot reference classes that are compiled in memory from `.java` files. That is, when class references in pre-compiled classes are encountered, the source tree is never consulted.

Differences between compilation at compile time versus launch time

There are some major differences between how the Java compiler compiles code on the source path when using `javac` and how it compiles code when using the `java` launcher in source-file mode.

- In source-file mode, the class declarations that are found in `.java` files may be compiled incrementally and on demand, during program execution, rather than being compiled all at once before execution starts. This means that if a compilation error occurs then the launcher will terminate after the program has already started executing. This behavior is different than prototyping with explicit compilation via `javac`, but it works effectively in the fast-moving edit/run cycle enabled by source-file mode.
- Classes that are accessed via reflection are loaded in the same manner as classes that are accessed directly. For example, if the program calls `Class.forName("pkg.Helper")` then the launcher's custom class loader will attempt to load the class `Helper` in the package `pkg`, potentially causing compilation of `pkg/Helper.java`. Similarly, if a package's annotations are queried via `Package::getAnnotations` then an appropriately-placed `package-info.java` file in the source tree, if present, will be compiled in memory and loaded.
- Annotation processing is disabled, similar to when `--proc:none` is passed to `javac`.
- It is not possible to run a source-code program whose `.java` files span multiple modules.

The last two limitations may be removed in the future.

Alternatives

- We could keep source-code programs restricted to single files and continue to require a separate compilation step for multi-file programs. While that does not impose significantly more work on the developer, the reality is that many Java developers have grown unfamiliar with the direct use of `javac` and prefer to rely on build tools when compilation to class files is required. Using the `java` command is less intimidating than using `javac`.
- We could make `javac` easier to use, with convenient defaults for compiling complete source trees. However, the need to set up a directory for the generated class files, or else have them pollute the source tree, is a speed bump to rapid prototyping. Developers often place their `.java` files under version control even at the tinkering stage, and would thus need to set up their version control repository to exclude the class files generated by `javac`.