Modules and javac

a.k.a. Making javac *module-aware*
(Not making javac *modular;* that comes later)

Alex Buckley, Jonathan Gibbons
Sun Microsystems

Module usage at compile-time

Scenario

- Compile code that belongs to the HelloWorld module
- The HelloWorld module depends on the Quux module

```
module HelloWorld { requires Quux; }
```

- javac can easily locate the Quux module and set an internal "magic ClassPath" containing Quux's classes

Problem

- Where does HelloWorld's module-info come from?

Compiling a single module

- HelloWorld's module-info comes from root of ClassPath:

```
src/classes/com/foo/HelloWorld.java
            /com/bar/Baz.java
            /module-info.java
```

ClassPath value shown in blue box

- Familiar
- Reuses existing structure of source trees
- Easy to find complete module content

Compiling multiple modules together

- Difficult but necessary
  - Module A requires module B which requires module A
  - To start with, all we have is the source of modules A and B
- Given:

```
src/classes/com/foo/HelloWorld.java
           /com/bar/Baz.java
           /module-info.java
```

  it's impossible to put `com/foo/HelloWorld.java` in a different module than `com/bar/Baz.java`

- What to do?

Compiling multiple modules together

- *If module name is in source*, derive a "deep" location from it:

```
src/classes/com/foo/HelloWorld.java    module com.foo.app;
            /module-info               module com.foo.app @ ..
        /com/bar/Baz.java              module com.bar.app;
            /module-info               module com.bar.app @ ..
```

- javac can find a module-info from a given module name
- But hard to find complete module content
  - Classes in a module could be anywhere under the ClassPath
- Module names overload the package hierarchy
  - No guarantee that module names will be similar to package names; some directories may hold just a module-info
- Makes the hard case (multiple modules) easy, and the easy case (one module) hard, as membership is repeated everywhere

Compiling multiple modules together

- *If module name is in source*, change path semantics to pick the 'right' module-info on the ClassPath?

```
src/classes1/com/foo/HelloWorld      module com.foo.app;
             /module-info            module com.foo.app @ ..
src/classes2/com/bar/Baz.java        module com.bar.app;
             /module-info            module com.bar.app @ ..
```

- Can *read* source for multiple modules but cannot *write* their classfiles, as -d sets **one** output directory
  - Module name in source doesn't help

Overcoming the -d limitation

- Read module-info.java from multiple top-level locations:

```
src/classes1/com/foo/HelloWorld.java
            /module-info.java
src/classes2/com/bar/Baz.java
            /module-info.java
```

- Write module-info.class to "deep" locations under -d:

```
build/classes/com/foo/HelloWorld.class
                      /module-info.class
              /com/bar/Baz.class
                      /module-info.class
```

-d value shown in green box

- Destroys input:output isomorphism required by many tools

From ClassPath to ModulePath

- Instead of putting many locations on the ClassPath:

```
src/classes1/com/foo/HelloWorld.java
            /module-info.java
src/classes2/com/bar/Baz.java
            /module-info.java
```

- Simply put one location on the *ModulePath*:

```
src/modules/com.foo.app/com/foo/HelloWorld.java
                       /module-info.java
           /com.bar.app/com/bar/Baz.java
                       /module-info.java
```

ModulePath value shown in orange box. Module names written in orange.

- When compiling `com.foo.app/com/foo/XXX`, javac gets module-info from `com.foo.app/module-info`

ModulePath is the answer

- **src/modules**/com.foo.app/com/foo/HelloWorld.java
  /module-info.java
  /com.bar.app/com/bar/Baz.java
  /module-info.java

- Can compile one or multiple modules together
- Can move classes between modules trivially
- Easy to find complete module content
- Multi-module packages "for free"
- Structuring the source tree like this is good practice

Structuring the source tree

- Can easily evolve from single-module structure of ClassPath:

```
src/classes/com/foo/HelloWorld.java
           /com/bar/Baz.java
           /module-info.java
```

- To multi-module structure of ModulePath:

```
src/modules/com.foo.app/com/foo/HelloWorld.java
                       /module-info.java
           /com.bar.app/com/bar/Baz.java
                       /module-info.java
```

- Each child of ModulePath is like a traditional ClassPath entry
- Structure of output directory (-d) depends on:
  - If ClassPath set: output to legacy single-module structure
  - If ModulePath set: output to multi-module structure

Multiple locations on the ModulePath

- src/modules/com.foo.app/com/foo/HelloWorld.java
  /module-info.java
  /com.bar.app/com/bar/Baz.java
  /module-info.java
  :
  build/gensrc/com.foo.app/com/foo/parser/Parser.java
  /com/foo/lexer/Lexer.java
  /com.bar.app/...
  :
  lib/thirdparty/org.w3c.xml/org/w3c/dom/Node.class
  /org/w3c/sax/Parser.class
  /module-info.class
  /org.omg.corba/...

Multiple versions on the ModulePath

- ModulePath so far allows *some* version of a given module:

```
src/modules/com.foo.app/com/foo/HelloWorld.java
                       /module-info.java
                           module com.foo.app @ 4.0 {..}
```

- ModulePath can also support multiple versions of a module:

```
src/modules/com.foo.app-4.0/com/foo/HelloWorld.java
                           /module-info.java
                               module com.foo.app @ 4.0 {..}
                   /com.foo.app-5.0/com/foo/HelloWorld.java
                           /module-info.java
                               module com.foo.app @ 5.0 {..}
```

javac and multiple versions

- When compiling a module M, javac must determine the modules it requires and set an internal "magic ClassPath" listing those modules
- M's required modules may come from ModulePath and/or the library of the Jigsaw module system
  - These locations may, in aggregate, have multiple versions of a required module
  - javac delegates to the Jigsaw module system to select the "best" available version of each and every module required by M
- The "magic ClassPath" for M lists the selected modules (and their location either on the ModulePath or in a library)

Example of multiple versions

- Suppose module M being compiled requires com.foo.app @ 1.0+
- Multiple versions of com.foo.app are available:

```
src/modules/com.foo.app-4.0/com/foo/HelloWorld.java
                            /module-info.java
           /com.foo.app-5.0/com/foo/HelloWorld.java
                            /module-info.java
```

Jigsaw	com.foo.app@5.0	com/foo/HelloWorld.class
		module-info.class
	com.foo.app@6.0	com/foo/HelloWorld.class
		module-info.class

- javac offers versions 4.0 and 5.0 from ModulePath to the module system, which also considers its own 5.0 and 6.0 versions
- The module system selects 6.0; javac adds it to M's "magic ClassPath"

Module membership in source

- ModulePath is agnostic about 'module' declarations in source determining module membership

Does module membership in source:

- Specify something an IDE couldn't infer? **No.**
  - Trivial to infer from filesystem structure
- Provide essential safety at compile-time? **No.**
  - Just prevents accidental movement between directories
- Provide essential safety at runtime? **No.**
  - Module system can always override
- Help when compiling multiple modules? **No.**
  - The problem is finding *a* module-info, not *the* module-info

Issues with membership in source

- Should module declarations be in every normal source file (repetitive) or in package-info (mostly unknown) or both?
- Two meanings for 'module' keyword (membership+accessibility)
- Host system conventions like ModulePath still matter
- Makes easy case hard + hard case easy

No module membership in source means:

- Only one module declaration (in module-info) per module
- Obvious filesystem structure drives membership
- Makes easy case easy + hard case possible

Conclusion: no module declarations in source files or Module attributes in classfiles (except for module-info.java/class)

Summary of javac flags

- -modulepath
  - The module-aware replacement for -classpath
  - Used for compiled classes of modules
  - Checked for source files unless -modulesourcepath is also given
- -modulesourcepath
  - The module-aware replacement for -sourcepath
  - Always best to put all necessary files on command line
- -classpath and -sourcepath still supported
- -d
  - Output directory for classfiles
  - Output directory structure follows input directory structure (output is as JDK6 unless -modulepath is specified)
- -s
  - Output directory for source files generated by annotation processors
  - Will probably adopt -d convention: output structure follows input structure