Project Jigsaw: Module System Quick-Start Guide

This document provides a few simple examples to get developers started with modules.

The file paths in the examples use forward slashes, and the path separators are colons. Developers on Microsoft Windows should use file paths with back slashes and a semi-colon as the path separator.

- Greetings
- Greetings world
- Multi-module compilation
- Packaging
- Missing requires or missing exports
- Services
- The linker
- --patch-module

Greetings

This first example is a module named `com.greetings` that simply prints "Greetings!". The module consists of two source files: the module declaration (`module-info.java`) and the main class.

By convention, the source code for the module is in a directory that is the name of the module.

```
src/com.greetings/com/greetings/Main.java
src/com.greetings/module-info.java

$ cat src/com.greetings/module-info.java
module com.greetings { }

$ cat src/com.greetings/com/greetings/Main.java
package com.greetings;
public class Main {
    public static void main(String[] args) {
        System.out.println("Greetings!");
    }
}
```

The source code is compiled to the directory `mods/com.greetings` with the following commands:

```
$ mkdir -p mods/com.greetings

$ javac -d mods/com.greetings \
    src/com.greetings/module-info.java \
    src/com.greetings/com/greetings/Main.java
```

Now we run the example with the following command:

```
$ java --module-path mods -m com.greetings/com.greetings.Main
```

`--module-path` is the module path, its value is one or more directories that contain modules. The `-m` option specifies the main module, the value after the slash is the class name of the main class in the module.

Greetings world

This second example updates the module declaration to declare a dependency on module `org.astro`. Module `org.astro` exports the API package `org.astro`.

```
src/org.astro/module-info.java
src/org.astro/org/astro/World.java
src/com.greetings/com/greetings/Main.java
src/com.greetings/module-info.java

$ cat src/org.astro/module-info.java
module org.astro {
    exports org.astro;
}

$ cat src/org.astro/org/astro/World.java
```

```
package org.astro;
public class World {
    public static String name() {
        return "world";
    }
}

$ cat src/com.greetings/module-info.java
module com.greetings {
    requires org.astro;
}

$ cat src/com.greetings/com/greetings/Main.java
package com.greetings;
import org.astro.World;
public class Main {
    public static void main(String[] args) {
        System.out.format("Greetings %s!%n", World.name());
    }
}
```

The modules are compiled, one at a time. The `javac` command to compile module `com.greetings` specifies a module path so that the reference to module `org.astro` and the types in its exported packages can be resolved.

```
$ mkdir -p mods/org.astro mods/com.greetings

$ javac -d mods/org.astro \
    src/org.astro/module-info.java src/org.astro/org/astro/World.java

$ javac --module-path mods -d mods/com.greetings \
    src/com.greetings/module-info.java src/com.greetings/com/greetings/Main.java
```

The example is run in exactly the same way as the first example:

```
$ java --module-path mods -m com.greetings/com.greetings.Main
Greetings world!
```

Multi-module compilation

In the previous example then module `com.greetings` and module `org.astro` were compiled separately. It is also possible to compile multiple modules with one `javac` command:

```
$ mkdir mods

$ javac -d mods --module-source-path src $(find src -name "*.java")

$ find mods -type f
mods/com.greetings/com/greetings/Main.class
mods/com.greetings/module-info.class
mods/org.astro/module-info.class
mods/org.astro/org/astro/World.class
```

Packaging

In the examples so far then the contents of the compiled modules are exploded on the file system. For transportation and deployment purposes then it is usually more convenient to package a module as a *modular JAR*. A modular JAR is a regular JAR file that has a `module-info.class` in its top-level directory. The following example creates `org.astro@1.0.jar` and `com.greetings.jar` in directory `mlib`.

```
$ mkdir mlib

$ jar --create --file=mlib/org.astro@1.0.jar \
    --module-version=1.0 -C mods/org.astro .

$ jar --create --file=mlib/com.greetings.jar \
    --main-class=com.greetings.Main -C mods/com.greetings .

$ ls mlib
com.greetings.jar   org.astro@1.0.jar
```

In this example, then module `org.astro` is packaged to indicate that its version is `1.0`. Module `com.greetings` has been packaged to indicate that its main class is

`com.greetings.Main`. We can now execute module `com.greetings` without needing to specify its main class:

```
$ java -p mlib -m com.greetings
Greetings world!
```

The command line is also shortened by using `-p` as an alternative to `--module-path`.

The `jar` tool has many new options (see `jar -help`), one of which is to print the module declaration for a module packaged as a modular JAR.

```
$ jar --describe-module --file=mlib/org.astro@1.0.jar
org.astro@1.0 jar:file:///d/mlib/org.astro@1.0.jar/!module-info.class
exports org.astro
requires java.base mandated
```

Missing requires or missing exports

Now let's see what happens with the previous example when we mistakenly omit the `requires` from the `com.greetings` module declaration:

```
$ cat src/com.greetings/module-info.java
module com.greetings {
    // requires org.astro;
}

$ javac --module-path mods -d mods/com.greetings \
    src/com.greetings/module-info.java src/com.greetings/com/greetings/Main.java
src/com.greetings/com/greetings/Main.java:2: error: package org.astro is not visible
    import org.astro.World;
               ^
  (package org.astro is declared in module org.astro, but module com.greetings does not read it)
1 error
```

We now fix this module declaration but introduce a different mistake, this time we omit the `exports` from the `org.astro` module declaration:

```
$ cat src/com.greetings/module-info.java
module com.greetings {
    requires org.astro;
}
$ cat src/org.astro/module-info.java
module org.astro {
    // exports org.astro;
}

$ javac --module-path mods -d mods/com.greetings \
    src/com.greetings/module-info.java src/com.greetings/com/greetings/Main.java
$ javac --module-path mods -d mods/com.greetings \
   src/com.greetings/module-info.java src/com.greetings/com/greetings/Main.java
src/com.greetings/com/greetings/Main.java:2: error: package org.astro is not visible
    import org.astro.World;
               ^
  (package org.astro is declared in module org.astro, which does not export it)
1 error
```

Services

Services allow for loose coupling between *service consumers* modules and *service providers* modules.

This example has a service consumer module and a service provider module:

- module `com.socket` exports an API for network sockets. The API is in package `com.socket` so this package is exported. The API is *pluggable* to allow for alternative implementations. The service type is class `com.socket.spi.NetworkSocketProvider` in the same module and thus package `com.socket.spi` is also exported.
- module `org.fastsocket` is a service provider module. It provides an implementation of `com.socket.spi.NetworkSocketProvider`. It does not export any packages.

The following is the source code for module `com.socket` .

```
$ cat src/com.socket/module-info.java
module com.socket {
    exports com.socket;
    exports com.socket.spi;
    uses com.socket.spi.NetworkSocketProvider;
}

$ cat src/com.socket/com/socket/NetworkSocket.java
package com.socket;

import java.io.Closeable;
import java.util.Iterator;
import java.util.ServiceLoader;

import com.socket.spi.NetworkSocketProvider;

public abstract class NetworkSocket implements Closeable {
    protected NetworkSocket() { }

    public static NetworkSocket open() {
        ServiceLoader<NetworkSocketProvider> sl
            = ServiceLoader.load(NetworkSocketProvider.class);
        Iterator<NetworkSocketProvider> iter = sl.iterator();
        if (!iter.hasNext())
            throw new RuntimeException("No service providers found!");
        NetworkSocketProvider provider = iter.next();
        return provider.openNetworkSocket();
    }
}


$ cat src/com.socket/com/socket/spi/NetworkSocketProvider.java
package com.socket.spi;

import com.socket.NetworkSocket;

public abstract class NetworkSocketProvider {
    protected NetworkSocketProvider() { }

    public abstract NetworkSocket openNetworkSocket();
}
```

The following is the source code for module `org.fastsocket` .

```
$ cat src/org.fastsocket/module-info.java
module org.fastsocket {
    requires com.socket;
    provides com.socket.spi.NetworkSocketProvider
        with org.fastsocket.FastNetworkSocketProvider;
}

$ cat src/org.fastsocket/org/fastsocket/FastNetworkSocketProvider.java
package org.fastsocket;

import com.socket.NetworkSocket;
import com.socket.spi.NetworkSocketProvider;

public class FastNetworkSocketProvider extends NetworkSocketProvider {
    public FastNetworkSocketProvider() { }

    @Override
    public NetworkSocket openNetworkSocket() {
        return new FastNetworkSocket();
    }
}

$ cat src/org.fastsocket/org/fastsocket/FastNetworkSocket.java
package org.fastsocket;

import com.socket.NetworkSocket;

class FastNetworkSocket extends NetworkSocket {
    FastNetworkSocket() { }
```

```
    public void close() { }
}
```

For simplicity, we compile both modules together. In practice then the service consumer module and service provider modules will nearly always be compiled separately.

```
$ mkdir mods
$ javac -d mods --module-source-path src $(find src -name "*.java")
```

Finally we modify our module `com.greetings` to use the API.

```
$ cat src/com.greetings/module-info.java
module com.greetings {
    requires com.socket;
}

$ cat src/com.greetings/com/greetings/Main.java
package com.greetings;

import com.socket.NetworkSocket;

public class Main {
    public static void main(String[] args) {
        NetworkSocket s = NetworkSocket.open();
        System.out.println(s.getClass());
    }
}


$ javac -d mods/com.greetings/ -p mods $(find src/com.greetings/ -name "*.java")
```

Finally we run it:

```
$ java -p mods -m com.greetings/com.greetings.Main
class org.fastsocket.FastNetworkSocket
```

The output confirms that the service provider has been located and that it was used as the factory for the `NetworkSocket`.

The linker

`jlink` is the linker tool and can be used to link a set of modules, along with their transitive dependences, to create a custom modular run-time image (see JEP 220).

The tool currently requires that modules on the module path be packaged in modular JAR or JMOD format. The JDK build packages the standard and JDK-specific modules in JMOD format.

The following example creates a run-time image that contains the module `com.greetings` and its transitive dependences:

```
jlink --module-path $JAVA_HOME/jmods:mlib --add-modules com.greetings --output greetingsapp
```

The value to `--module-path` is a PATH of directories containing the packaged modules. Replace the path separator ':' with ';' on Microsoft Windows.

`$JAVA_HOME/jmods` is the directory containing `java.base.jmod` and the other standard and JDK modules.

The directory `mlib` on the module path contains the artifact for module `com.greetings`.

The `jlink` tool supports many advanced options to customize the generated image, see `jlink --help` for more options.

`--patch-module`

Developers that checkout `java.util.concurrent` classes from Doug Lea's CVS will be used to compiling the source files and deploying those classes with `-Xbootclasspath/p`.

`-Xbootclasspath/p` has been removed, its module replacement is the option `--patch-module` to override classes in a module. It can also be used to augment the contents of module. The `--patch-module` option is also supported by `javac` to compile code "as if" part of the module.

Here's an example that compiles a new version of `java.util.concurrent.ConcurrentHashMap` and uses it at run-time:

```
javac --patch-module java.base=src -d mypatches/java.base \
    src/java.base/java/util/concurrent/ConcurrentHashMap.java

java --patch-module java.base=mypatches/java.base ...
```

More information

- The State of the Module System
- JEP 261: Module System
- Project Jigsaw

Feedback

Please send usage questions and experience reports to the jigsaw-dev list. Specific suggestions about the design of the module system should be sent to the JSR 376 Expert Group's comments list.