CHAPTER 16

■ ■ ■

Breaking Module Encapsulation

QUESTIONS AND EXERCISES

1. What is breaking module encapsulation?

Answer:

Allowing access to otherwise inaccessible packages and types of a module is called breaking encapsulation of the module. Sometimes, you need to break encapsulation to enable white-box testing, use unsupported JDK-internal APIs, or use third-party libraries.

2. Describe the effects of using the `--add-exports`, `--add-opens`, and `--add-reads` command-line options.

Answer:

If a package is not exported by a module, it can be exported by using the `--add-exports` command-line option. The syntax to use the `--add-exports` command-line option is as follows:

```
--add-exports <source-module>/<package>=<target-module-list>
```

If a package of a module is not open, it can be opened by using the `--add-opens` command-line option. The syntax to use the `--add-opens` command-line option is as follows:

```
--add-opens <source-module>/<package>=<target-module-list>
```

The `--add-reads` command-line option can be used to add a readability edge from a module to another module. The syntax to use the `--add-reads` command-line option is as follows:

```
--add-reads <source-module>=<target-module-list>
```

3. What is the difference between using the `--add-exports` and `--add-opens` command-line options and their counterparts the `Add-Exports` and `Add-Opens` attributes in the manifest file?

Answer:

The effects of using these attributes in the manifest file is the same as using the similarly named command-line options, except that these manifest attributes export or open the specified packages to all unnamed modules. Another difference is that the `Add-Exports` and `Add-Opens` manifest attributes are used by the runtime only when the application is run using an executable JAR – using the `-jar` option with the java command; in other cases, these attributes are ignored.

4. Suppose you have a module named `M`, which needs to use types in unnamed modules. Write the command-line option to achieve this.

Answer:

The following command-line option will make the module `M` read all unnamed modules:

```
--add-reads M=ALL-UNNAMED
```

If the target module list is a special value, `ALL-UNNAMED`, for the `--add-reads` option, the source module reads all unnamed modules. This is the only way a named module can read unnamed modules. There is no equivalent module statement that you can use in a named module declaration to read an unnamed module.

5. Describe the `--illegal-access` command-line option with its default behaviors in JDK9 and its proposed behaviors in the future JDK.

Answer:

Many applications written prior to JDK 9 may not be modularized soon. To take advantage of the latest JDK, they may be migrated to JDK 9 without modularizing them. Such applications will run from the class path. These applications were allowed to access non-public members of the JDK-internal API using deep reflection. To ease migration of these applications, JDK 9 allows deep reflection on the JDK 9 modules by default, which breaks the JDK module's encapsulation. This is allowed to help the Java community adapt JDK 9 sooner. Another reason behind allowing this was the backward compatibility. Applications performing deep reflections on the JDK-internals will continue to work in JDK 9 because the Java community did not get an advance notice that deep reflection on JDK-internals will stop working in JDK 9.

JDK 9 provides an `--illegal-access` option for the `java` command. As its name suggests, it is to work with illegal access by code in any *unnamed module* (code on the class path) to members of types in any named modules of the JDK using deep reflection. Its syntax is as follows:

```
java --illegal-access=<permit|deny|warn|debug> <other-arguments>
```

The option takes one of the four parameters: `permit`, `deny`, `warn`, and `debug`. The default is `permit`, which means that the absence of the `--illegal-access` option is the same as `--illegal-access=permit`. That is, by default, all packages in all explicit modules in the JDK are open to the code in all unnamed modules. The code performing illegal reflective access should be fixed sooner or later. To help identify such offending code, the JDK 9 issues a warning to standard error on the first such illegal access. You cannot suppress this warning.

The `deny` parameter disables all illegal access to the members of the JDK internal types using deep reflection, except when it is allowed using other options such as `--add-opens`. In a future release, `deny` will become the default mode. That is, by default, illegal reflective access will be disabled in a future release and you will need to use `--illegal-access=permit` explicitly to enable illegal reflective access.

The `warn` parameter issues a warning for each illegal-reflective access. This is helpful in identifying all code in your existing applications that uses illegal reflective access.

The `debug` parameter issues a warning and prints a stack trace for each illegal-reflective access. This is helpful in identifying all code in your existing application that uses illegal reflective access.

The `--illegal-access` option does not permit illegal access by code in a named module to the members of types in other named modules. To perform illegal reflective access in such cases, you can combine this option with `--add-exports`, `--add-opens`, and `--add-reads` options.

The `--illegal-access` option will be removed in a future release. First, its default mode will be changed from `permit` to `deny` and then the option itself will be removed. It is to be emphasized that this option is only for allowing illegal access to the JDK-internals by the code on the class path. This option does not allow illegal access to the members of the non-JDK modules; you must use the `--add-opens` option or open the module/package if illegal access is needed to the members of non-JDK modules.