CHAPTER 15

■ ■ ■

The Module API

QUESTIONS AND EXERCISES

1. What is the fully qualified name of the class whose instances represent a module at runtime?

Answer:

```
java.lang.Module
```

2. Write the code to get the reference of the module of a class named Person?

Solution:

```
// Get the Class object of the Person class
Class<Person> cls = Person.class;

// Get the module reference
Module module = cls.getModule();
```

3. If you have a class named `Person`, how do you know whether this class is a member of a named module or unnamed module?

Answer:

First, get the reference of the module of the `Person` class and then use the `isNamed()` method on the module reference. The `isNamed()` method of the `Module` class returns `true` for a named module and `false` for an unnamed module. The following snippet of code does this:

```
Module m = Person.class.getModule();
if(m.isNamed()) {
    // The Person class is a member of a named module
} else {
    // The Person class is a member of an unnamed module
}
```

4. What does an instance of the `ModuleDescriptor` class represent? Is the instance of the `ModuleDescriptor` class immutable?

Answer:

An instance of the `ModuleDescriptor` class represents a module definition, which is created from a module declaration—typically from a `module-info.class` file or created on the fly using the `ModuleDescriptor.Builder` class. A `ModuleDescriptor` is immutable.

5. Can you directly obtain a `ModuleDescriptor` from a `module-info.class` file? If your answer is yes, explain how you do it.

Answer:

Yes, a `ModuleDescriptor` object can be obtained by reading the binary form of the module declaration from a `module-info.class` file using one of the static `read()` methods of the `ModuleDescriptor` class. The following snippet of code shows how to do it:

```
String moduleInfoPath = "<path-to-the-module-info.class-file>";
try {
    ModuleDescriptor desc = ModuleDescriptor.read(new FileInputStream(moduleInfoPath));
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

6. Can you get a `ModuleDescriptor` for an unnamed module?

Answer:

An unnamed module does not have a module descriptor. The `getDescriptor()` method of the `Module` class returns `null` for an unnamed module.

7. Name the classes whose instances represent exports, opens, provides, and requires statements in a module declaration.

Answer:

he `ModuleDescriptor` class contains the following static nested classes whose instances represent a statement with the same name in a module declaration:

- `ModuleDescriptor.Exports`
- `ModuleDescriptor.Opens`

- `ModuleDescriptor.Provides`
- `ModuleDescriptor.Requires`

8. What is the difference between the `ModuleDescriptor::packages()` and the `Module::getPackages()` methods? Both methods return a set of package names.

Answer:

The `ModuleDescriptor::packages()` method returns the set of packages (exported or not) defined in the module declaration for named modules only. The `Module::getPackages()` method can be used to get `ModuleDescriptor` for an unnamed module. Another difference is that the package names reported by a `ModuleDescriptor` are static; the package names reported by a `Module` are dynamic, which reports the packages loaded in the module at the time the `getPackages()` method is called.

9. How do you check if a module exports a package to all other modules or to a specific module?

Answer:

The `isExported(String packageName)` method of the `Module` class returns `true` if this module exports the given `packageName` to all modules. The `isExported(String packageName, Module module)` method of the `Module` class returns `true` if this module exports the given `packageName` to at least the given `module`.

10. How do you know if a module is automatic?

Answer:

The `isAutomatic()` method of the `ModuleDescriptor` class returns `true` for an automatic module and `false` otherwise.

11. Suppose there is a module named `M`, which contains a package named `P`, but does not export the package to any other module. Can the code in another module named N export the package `P` in module `M` to module `N` at runtime?

Answer:

No. Only the code in module `M` can use the `addExports()` method of the `Module` class using the reference of module `M` to export package `P` to module `N`.

12. Suppose there is a module named `M`, which contains a package named `Q` and opens the package to module `N`. Can the code in module `N` open the package `Q` in module `M` to another module `T` at runtime?

Answer:

Yes.

13. If a module named `M` contains resources in a package named `P`. How can the module `M` make the resources available to all other modules?

Answer:

Module `M` can open package `P` to make the resources available to all other modules.

14. If a module named M contains resources in a directory named META-INF, can other modules access those resources?

Answer:

Yes. As per rules applicable to resource contained in a named module, since resource directory META-INF is not a valid java package name, resources can be accessed by code in any modules.

15. What is the name of the scheme you must use to access resources in Java runtime image in JDK9?

Answer:

The `jrt` scheme.

16. Write the URL that you need to use to access to the `Object`.class file from the runtime image in JDK9.

Answer:

```
jrt:/java.base/java/lang/Object.class
```

17. Can you use annotations on module declarations?

Answer:

Yes

18. Is the following statement true or false?

    When a module is deprecated, the use of that module in `requires`, `exports`, and `opens` statements causes a warning to be issued.

Answer:

False. When a module is deprecated, the use of that module in `requires`, but not in `exports` or `opens` statements, causes a warning to be issued.

19. Can you use annotation on `requires`, `exports`, and `opens` statements in a module declaration?

Answer:

No. Individual module statements in a module declaration cannot be annotated. You can use annotations only on the module declaration.

20. What is a module layer? How are the layers, configurations, and class loaders related?

Answer:

A layer is a set of resolved modules with a function that maps each module to a class loader, which is responsible for loading all types in that module.

The set of resolved module is called a *configuration.*

Relationship between layers, configurations, and class loaders is as follows:

```
Configuration = A module graph
Module Layer = Configuration + (Module -> Class loader)
```

21. What is the use of an instance of the `ModuleFinder` interface and an instance of the `ModuleReference` class?

Answer:

A `ModuleFinder` is used to find modules (the `ModuleReference` class instance) during module resolution and service binding. A `ModuleReference` represents a reference to the contents of a module.

22. Name the class whose instances represent a configuration in a module layer. Can a configuration have multiple parent configurations?

Answer:

An instance of the `java.lang.module.Configuration` class represents a configuration. Yes. A configuration has at least one parent, except the empty configuration.

23. How many parent layers can exist for a given module layer? What is the parent layer of the boot layer?

Answer:

Layers are arranged hierarchically with one layer having one or more parent layers. Parent layer of boot layer is the empty layer.

24. Write a program that prints the names of all modules loaded into the boot layer.

Answer:

```
// BootLayerModules.java
package com.jdojo.module.api.exercises;

import java.util.Set;

public class BootLayerModules {
    public static void main(String[] args) {
        System.out.println("Modules loaded by boot layer:");
        Set<Module> modules = ModuleLayer.boot().modules();
        modules.forEach(m -> System.out.println(m.getName()));
    }
}
```