CHAPTER 3

■ ■ ■

Reflection

QUESTIONS AND EXERCISES

1. What is reflection?

   Answer:

   Reflection is the ability of a program to query and modify its state "as data" during the execution of the program

2. Name two Java packages that contain the reflection related classes and interfaces.

   Answer:

   `java.lang.reflect` and `java.lang`.

3. What does an instance of the `Class` class represent?

   Answer:

   An instance of the class `Class` represents a class or an interface in a running Java application.

4. List three ways to get the reference of an instance of the `Class` class.

   Answer:

   The reference of the `Class` object of a class can be obtained using:

- The class literal such as `String.class`.
- The `getClass()` method of the `Object` class
- The `forName()` static method of the `Class` class

5. When do you use the `forName()` method of the `Class` class to get an instance of the `Class` class?

Answer:

The `forName()` static method of the `Class` class can be used to load a class/interface when name of class/interface to be loaded is known/available only at runtime or the name of the class/interface is available in a `String`.

6. Name three built-in class loaders. How do you get references of these class loaders?

Answer:

The three built-in class loaders are: bootstrap class loader, Platform class loader, and Application class loader.

The bootstrap class loader is represented by `null` and you cannot get its reference in your program, You can get the reference of the platform and system class loaders using the `getPlatformClassLoader()` and `getSystemClassLoader()` static methods of the `ClassLoader` class, respectively.

7. If you get a reference of the `Class` class, how do you know if this reference represents an interface?

Answer:

Use the `isInterface()` method on the reference. If it returns `true`, the reference represents an interface. Otherwise, the reference does not represent an interface.

8. What do instances of the `Field`, `Constructor`, and `Method` classes represent?

Answer:

Instances of the `Field`, `Constructor`, and `Method` classes represent fields, constructors and methods, respectively, in a class/interface.

9. What is the difference between using the `getFields()` and `getDeclaredFields()` methods of the `Class` class?

Answer:

The `getFields()` method returns a `Field[]` that contains all accessible public fields of the class or interface, and inherited fields from its superclass and superinterfaces.

The `getDeclaredFields()` method returns a `Field[]` that contains all fields (public, private, protected, and package-level) declared in the class or interface declaration. It does not include inherited fields.

10. You need to use `setAccessible(true)` or `trySetAccessible()` method of the `AccessibleObject` class to make a `Field`, `Constructor`, and `Method` object accessible even if they are inaccessible (e.g. they are declared private). What is the difference between these two methods?

Answer:

The `setAccessible()` method throws exception if accessible flag cannot be set and its return type is `void`. The `trySetAccessible()` method does not throw any exception if accessible flag cannot be set. Success or failure is indicated by a boolean value - `true` or `false`.

11. Assume that you have two modules named `R` and `S`. Module `R` contains a public `p.Test` class with a public method `m()`. The code in module `S` needs to use the class `p.Test` to declare variables and create its objects. Module `S` also needs to use reflection to access the public method `m()` of the `p.Test` class in module `R`. What is the minimum you need to do while declaring module `R`, so module `S` can perform these tasks?

Answer:

Exporting a package of a module grants access (at compile-time and at runtime) to the public types in the package and the accessible public members of those types to another module. Exporting a package also allows access to accessible public members by reflection. So, module `R` should export the package `p` as follows:

```
module R {
```

```
    exports p;

    // Other module statements go here
}
```

12. How do you open a package in a module? What is an open module?

Answer:

Opening a package in a module allows deep reflection on types of that package by code in other modules at runtime. An open module is a module that opens (allows deep reflection) *all of its packages* to all other modules.

13. What is the difference between exporting and opening a package of a module? Give an example when you will need to export and open the same package of a module.

Answer:

Exporting a package allows access to public types in the package at compile-time and runtime. Opening a package allows deep reflection at runtime. Consider a package `z` of module `X` having common bean classes. Some modules may be interested in using public methods while some other module wants to use deep reflection. In this case module `X` needs to export and open package `z`.

14. Consider the declarations of a module named `jdojo.reflection.exercise.model` and a `MagicNumber` class in that module as follows:

```
// module-info.java
module jdojo.reflection.exercises.model {
    /* Add your module statements here */
}

// MagicNumber.java
package com.jdojo.reflection.exercises.model;

public class MagicNumber {
    private int number;

    public int getNumber() {
        return number;
    }
```

```
    public void setNumber(int number) {
        this.number = number;
    }
}
```

Modify the module declaration so that code in other modules can perform deep reflection on the objects of the `MagicNumber` class. Create a class named `MagicNumberTest` in a module named `jdojo.reflection.exercises`. The code in the `MagicNumberTest` class should use reflection to create an object of the `MagicNumber` class, set its private `number` field directly, and read the current value of the number field using the `getNumber()` method.

Solution:

```
// module-info.java for module - jdojo.reflection.execercise.model
 module jdojo.reflection.exercises.model {
    exports com.jdojo.reflection.exercises.model;
    opens com.jdojo.reflection.exercises.model;
 }

 // module-info.java for module - jdojo.reflection.execercises
 module jdojo.reflection.exercises {
    requires jdojo.reflection.exercises.model;
 }

 // MagicNumberTest.java
 package com.jdojo.relection.exercises;

 import com.jdojo.reflection.exercises.model.MagicNumber;
 import java.lang.reflect.Field;

 public class MagicNumberTest {
    public static void main(String[] args) throws Exception {
        MagicNumber mg = new MagicNumber();
        mg.setNumber(100);

        String className = "com.jdojo.reflection.exercises.model.MagicNumber";
        Class<?> cls = Class.forName(className);

        // Get the value field reference
        Field valueField = cls.getDeclaredField("number");

        // try making the value field accessible before accessing it
        boolean accessEnabled = valueField.trySetAccessible();

        if (accessEnabled) {
            System.out.println("Current MagicNumber = " + valueField.get(mg));
            valueField.set(mg, 200);

            System.out.println("Updated MagicNumber = " + valueField.get(mg));
        } else {
            System.out.println("MagicNumber private field is not accessible.");
        }
```

```
    }
}
```

15. Can you access private members of JDK classes in Java 9? If your answer is yes, describe the rules and restrictions for such access.

Answer:

Yes. JDK 9 allows deep reflection on members of JDK internal types from the code in unnamed modules. So, deploy or run application on the class path to get access to private members. The runtime issues a warning for first illegal access.

16. Assume there are two modules `P` and `Q`. Module `P` is an `open` module. Module `Q` wants to perform deep reflection on types in module `P`. Is module `Q` required to read module `P` in its module's declaration?

Answer:

A module that performs deep reflection on open packages of another module does not need to read the module containing the open packages. However, adding a dependence on a module with open packages is allowed and strongly encouraged (if module name is known), so the module system can verify the dependence at compile-time and at runtime.

17. Assume there are two modules `M` and `N`. Module `M` does not open any of its packages to any modules, but it exports a `com.jdojo.m` package to all other modules. Can module `N` use reflection to access publicly accessible members of the `com.jdojo.m` package of module `M`?

Answer:

Yes. Since module M exports its package named `com.jdojo.m`, any module (including module `N`) can access the publicly accessible members of module `M` using reflection.
