Reflection

Overview

In this lab, you will refactor an existing application so that it uses reflection to detect and use class types at run-time.

Source folders

Student project: `StudentReflection`
Solution project: `SolutionReflection`

Roadmap

There are 3 exercises in this lab, of which the last exercise is "if time permits". Here is a brief summary of the tasks you will perform in each exercise; more detailed instructions follow later:

1. Creating an object for a specified class type
2. Creating an object for a specified class name
3. Testing if a class implements a target interface

Familiarization with the application

Open the student project. The application is complete as it stands, although it doesn't make use of reflection yet. Expand the `student.reflection` package and take a look at the following interfaces and classes:

- `Logger`
  The `Logger` interface provides a pluggable logging capability for the application, i.e. the application can potentially use any logging class that implements this interface. By the time you have finished this lab, the application will be capable of using absolutely any logger class at all, courtesy of reflection (the application will even be able to use completely new classes that are unknown at development time).

- `MemoryLogger`
  This is a simple implementation of the `Logger` interface, which stores log messages in an in-memory collection. The log messages are discarded when a `MemoryLogger` object is destroyed.

- `FileLogger`
  This is another implementation of the `Logger` interface, which writes log messages to a file on the hard disk (the name of the log file is hard-coded for simplicity). The log messages persist after a `FileLogger` object is destroyed.

- `Account`
  This is a simple "business class". The `Account` constructor receives a logger object, and the various `Account` methods use this logger object to write log messages.

- `Main`
  This is the entry-point class for the application. At the moment, the application creates a logger object using static data-typing (see the `createLoggerStatically()` method). The purpose of this lab is to make the application more flexible, so that it can create a logger object of any type via reflection. This would allow the application to use logger classes that are completely unknown at compile-time.

Run the application. Verify that the application writes log messages either to memory or to a file, depending on which type of logger you return from the `createLoggerStatically()` method.

Exercise 1: Creating an object for a specified class type

In the `Main` class, add a new method named `createLoggerForClassType()`. The purpose of this method is to create a new instance of a specified type of logger, by using reflection. Follow these guidelines:

- The method should take a `Class<?>` parameter, which indicates the type of class to instantiate.
- The method return type should be `Logger`, i.e. the method will return an instance of a class that implements `Logger` (such as `MemoryLogger` or `FileLogger`).
- The method must be `static` (so it can be called easily from `main()`).

Inside `createLoggerForClassType()`, use the `Class<?>` object's `newInstance()` method to create an instance of the appropriate class type. You'll need to wrap this code in a `try/catch` block, to catch various reflection-related exceptions.

Now go back to `main()`, and locate the call to `createLoggerStatically()`. Replace this with a call to your new `createLoggerForClassType()` method. Pass in a class-type parameter, such as `MemoryLogger.class`.

Run the application, and verify that it still works now that you're using reflection to create the logger object from a class type.

Exercise 2: Creating an object for a specified class name

In the `Main` class, add another new method named `createLoggerForClassName()`. This is an even more general-purpose version of the `createLoggerForClassType()` method, where the parameter is just a class name (rather than a class type).

Follow these guidelines:

- The method should take a `String` parameter, which indicates the class name to instantiate. This is a common pattern in reflection-based applications, where you just specify the "class name to instantiate" as a `String` parameter.
- The method return should be `Logger`, as per `createLoggerForClassType()` .
- The method must be `static`, as per `createLoggerForClassType()`.

Inside `createLoggerForClassName()`, call `Class.forName()` to obtain a `Class<?>` object for the specified class type. Once you have this `Class<?>` object, you can simply pass it into your `createLoggerForClassType()` method you wrote in Exercise 1, to create a new instance of that class type.

Now go back to `main()` and invoke your `createLoggerForClassName()` method. Pass in a parameter such as `"student.reflection.MemoryLogger"`.

Run the application, and verify that it still works now that you're using reflection to create the logger object from a class name.

Exercise 3 (If time permits): Testing if a class implements a target interface

When you use reflection to create objects for a specified class type or class name, you'll probably want to add some sanity checks into your code. For example, you might want to ensure that the specified class exists(!) or that the class implements a particular interface…

Enhance `createLoggerForClassType()` and `createLoggerForClassName()` so that they test whether the specified class implements the `Logger` interface.

Hints:

- `Class<?>` has a method named `getInterfaces()`, which returns an array that describes all the interfaces implemented by the class. The array is expressed as a `Class<?>[]`, i.e. each array element is a `Class<?>` that describes an interface type.

- Loop through the array to see if any of the `Class<?>` objects represents the `Logger` interface. If no such element is found, the class doesn't implement `Logger`!