Worksheet 6: API Design, Plugins and Scripting

Updated: 11th October, 2019

Note: Don't mix up the idea of writing a plugin system with the idea of using plugins. You've already been using Gradle's plugins, of course, when writing `build.gradle`. These are quite separate from developing a plugin system for your own application.

(Theoretically, you can write your own Gradle plugins too, in order to help writers of build scripts perform additional tasks. The lecture notes and practical worksheets for this unit have been prepared using a custom-built Gradle "TEX" plugin, for instance.)

1. Design an API

Say you would like to write a pluggable/scriptable calendar application.

(a) Propose an API that would allow plugin/script code to do the following:

- Add an event to the calendar. A calendar event has (at a minimum) a name, a date, a start time, and a duration.
- Be notified (and thus take some action) when a particular named calendar event is starting.

(b) Show how the API could actually be used by writing an example plugin/script.

For this question, you only need to write the API declarations and the client code, *not* the plugin/script loader or any of the API implementation. Obviously you won't be able to actually run your code, so you will need to have confidence in your ability to understand whether it would run!

2. Using Reflection

Note: To use the Java reflection API, you need to import "`java.lang.reflect.*`".

(The `Class` class is itself imported automatically, being inside "`java.lang`", but the other reflection-related classes are not.)

Using the Java reflection API, write a simple program (one .java file will suffice) to do the following:

(a) Prompt the user to input the name of a Java class. (Alternatively, you can use a command-line parameter, if you wish.)

> **Note:** Remember that this is a *class name* and not a *filename*.

(b) Check for the existence of a constructor that takes a single `String` parameter. If confirmed, prompt the user for such a string, and call the constructor to retrieve a new object of this class.

(c) Output a list of all non-`static` methods within the class that take a single `int` as a parameter.

(d) Ask the user to select a method from the list, ask them to enter a value for the parameter, and then call the method with that value.

(e) Output the return value.

(f) Write a test class to test your program!

For convenience, you can simply place it alongside the code for your actual program. (In principle, it's highly likely that the class you're accessing dynamically is something outside your main project. However, to do this, you need to know how to setup the Java "classpath".)

3. Using Annotations

Consider the following (partial) plugin code:

```
public class NerdDatesPlugin
{
    @CalendarDate(day = 4, month = 5, name = "May the Fourth")
    public void mayTheFourth() {...}

    @CalendarDate(day = 25, month = 5, name = "International Towel Day")
    public void towelDay() {...}

    @CalendarDate(day = 19, month = 9,
                  name = "International Talk Like a Pirate Day")
    public void pirates() {...}
}
```

(a) Create `CalendarDate.java`, containing the declaration for this annotation type.

> **Note:** To work with annotations, you need to import "`java.lang.annotation.*`".

> **Note:** Annotation declarations often need their own "meta-annotations" to tell the compiler and JVM how to handle them:
>
> - `@Retention(RetentionPolicy.RUNTIME)` specifies that the annotation is to be made available ("retained") at runtime, so that reflection code can access it. This is *not* the default, and so specifying it will be essential in our case.
> - `@Target(...)` specifies where an annotation may appear. By default, they can appear almost anywhere, but we could restrict them to methods and fields with this: `@Target({ElementType.METHOD, ElementType.FIELD})`.

(b) Write a Java program to find all methods in a given Java class (specified by the user) that have a `@CalendarDate` annotation.

(c) For each method, output the method name, and the `day`, `month` and `name` annotation details.

4. Scripting

Obtain a copy of `canvas.zip` from Blackboard. This contains:

- `Canvas.java` – a class containing several methods (a simple API) for drawing coloured lines on the screen.
- `CanvasClickHandler.java` – an interface (part of the same API) intended to be implemented by a script/plugin in order to receive mouse click events.
- `script.py` – an example Python script intended to use that API.

Write a Java program that sets up a Jython Python interpreter to run a script *like* `script.py`, such that it is able to actually use the API. Notice that the example script refers to a global object called `canvas`.

You must ask the user for the filename, rather than hard-coding it, so that other similar scripts can be run too.

Note: Gradle can manage Jython, as a dependency, so that you don't need to actually have Jython installed. In your `build.gradle`, specify the following dependency:

```
implementation 'org.python:jython-standalone:2.7.1'
```

However, in using Gradle (and in particular *running* your program with "`gradlew run`"), there is another trick needed to ensure console input works as expected:

```
run {
    standardInput = System.in        // No quotation marks
}
```

This is only needed if you're using `Scanner` (or similar) to get user input.

Note: Here's a quick way to read a text file into a string:

```
import java.nio.file.*;
import java.io.IOException;
...
String filename = ...;
String content;
try
{
    content = new String(Files.readAllBytes(Paths.get(filename)), "UTF-8");
    ...
}
catch(IOException e) {...}
```

End of Worksheet