Software Engineering Concepts (COMP3003)

6b. Plugins and Scripting

CRICOS Provide Code: 00301J.
Updated: 08/10/2019.

Plugins and scripting allow third-party developers to get involved in extending your software. Both plugins and scripts are extra pieces of software that can be installed on top of your main application in some way, and which add features to it. This is a form of "future-proofing", where you yourself don't have to anticipate all the features users may actually want.

As we discussed in lecture 1, security is often an important tradeoff here. By creating a plugin mechanism, you're effectively telling your users to trust third-party developers. An attacker might take advantage of this by posing as a trustworthy plugin developer, while promoting a plugin that gives the attacker access to the user's data. This can be a manageable risk, though, if you keep your users educated and/or set up a process to verify the trustworthiness of plugin developers.

Indeed, despite the risks, the rewards in practice are too important to ignore. Plugins and scripts are a reality.

But what's the difference between a plugin and a script? For our purposes, we'll (somewhat arbitrarily) apply the following distinction:

- A **plugin** runs on the same platform as its host application (i.e., the main application that it is "plugging into"). It is *more likely* to be written in the same (or similar) language to its host application, often a compiled and strongly-typed language like Java, C++ or C#. It tends to have more direct access to the application itself and OS.
- A **script** runs on an interpreter launched by the host application, and is typically written in a dynamically-typed language like JavaScript or Python. The level of experience required to write a script is, on the whole, lower than for a plugin. It is also more likely to be **sandboxed**, to stop it accessing things that it shouldn't (though this is not always effective).

This distinction is open to challenge, and there are similar terms like "extension" and "add-on" that muddy the waters further. We really just want to illustrate that there are two slightly different technical approaches in play.

1 The API

In either case, we need to get used to having a "hybrid" application, made up of software from multiple software development projects. This is an important concept. One team of software developers will create the main application, and one or more *completely separate* teams (or individuals) will create the plugins/scripts. Nobody gets to claim 100% ownership of what actually happens in the running program. Instead, they have to cooperate.

In a sense, there is a border running through the middle of the running program. On one side, there is the main application code. On the other side, there is plugin code. Neither one "sees" what is really happening inside the other, despite the fact that they share the same environment. However, they must be able to interact, and so the one thing that they do agree on is an API. If the application and plugin are separate countries, then the API is the international treaty (or "duty free zone") between the two.

The API is defined and implemented by the main application, and used by the plugins. It consists of interfaces, classes, methods and/or functions that give the plugin access to key application features, events and data. However, there is also a clear separation between the API and the main application's internal workings. The main application doesn't (or shouldn't) let plugins access "implementation" classes, for maintainability reasons.

In the simplest case, the main application could expose two interfaces to the plugins – one representing the plugin itself, and one representing the API:

Listing 1: Plugins, in the simplest case.

```
public interface Plugin // Each plugin contains a class implementing this.
{
    void start(API api);
}

public interface API // The main application contains a class implementing this.
{
    ... // Methods for accessing the main application
}
```

The idea is as follows:

- Each plugin (written by third party developers) implements the `Plugin` interface and thus defines a `start()` method (or equivalent), which acts a bit like the plugin's `main()`. Through the parameter, it has a reference to an `API` object (or equivalent) that it can use to do whatever it likes, depending on what the API implementation allows.
- The main application contains a class that implements the `API` interface. This is not visible to the plugin, but provides all the functionality behind the API. The main application creates an object of this class, and provides it to the `Plugin` objects.

In general, you would expect the API to consist of *several* classes and interfaces, each representing a particular concept that the main application wants to let the plugins access or manipulate.

In particular, the API in practice almost certainly has various **callback** mechanisms. Plugins don't just want to run some code on start up. They want to run code in response to an event, or to produce certain results in certain situations.

For instance, consider a weather application. The point of having plugins in such an application might be to achieve the following:

- Acquiring weather data from different sources (different national weather agencies) that the core app developers can't manage for themselves;
- Performing certain actions in response to certain kinds of weather forecasts (storms, heatwaves, etc.).

The main application knows *when* these things should (or at least could) happen, and it must give its plugins the chance to intervene at the right time. To do this, its API should allow plugins to register callbacks. For instance:

Listing 2: Callback mechanisms in a plugin API.

```
public interface WeatherPlugin      // Implemented by each plugin.
{
    void start(Control api);
}

public interface Control            // Implemented by the main application.
{
    void registerDataCollector(String country, DataCollector callback);
    void registerForecastHandler(ForecastHandler callback);
    ... // Other methods for retrieving weather info.
}

public interface DataCollector      // Also implemented by plugins.
{
    WeatherData collect(); // Where 'WeatherData' is some object representing
}                          // whatever data needs to be retrieved.

public interface ForecastHandler    // Also implemented by plugins.
{
    void forecastChanged(Forecast newForecast); // Where 'Forecast' is some object
                                                // representing the projected
}                                               // weather over the next week.

public class WeatherData {...}
public class Forecast {...}
```

Any plugins would implement `WeatherPlugin`, but also provide zero-or-more implementations of `DataCollector` and/or `ForecastHandler`. The sequence of events would play out like this:

1. The main application loads the plugin, creates a `Control` object (an instance of an internal class that implements `Control`), and passes it to `start()`.
2. The plugin initialises itself. It creates `DataCollector` and/or `ForecastHandler` objects, and passes them to the "`register...()`" methods.
3. The main application now knows about the plugin's callbacks. It waits for the appropriate time (when it needs to update its data, and when a forecast changes), and calls `collect()` and/or `forecastChanged()`.
4. Now the plugin's own code runs. It can do whatever it likes, but presumably in the case of `collect()` it will download the appropriate data from `weatherbureau.gov.somecountry` and convert it into a `WeatherData` object. In the case of `forecastChanged()`, it will examine the `Forecast` object and perhaps pop up a notification or send an SMS alert.

In this way, the plugin can usefully add capabilities to the main application. In the general case, you would actually have *multiple* plugins, each providing `DataCollector` instances and/or `ForecastHandler` instances.

But there is one big unanswered question here: how does the main application actually get a `Plugin` object in the first place?

2 Reflection

Reflection lets you query language structures at runtime, to write code to manipulate other code dynamically. Most modern languages (notably *not* C/C++) incorporate reflection into their standard APIs.

2.1 Why?

In short, we want to be able to load plugins! The key challenge in loading plugins (and scripts) is that you, as the developer of the main application, don't actually know about them. The point is that the plugins are developed *after* your main application has been written, compiled, packaged, distributed and deployed.

So, there's no way that the following code can conceptually work:

Listing 3: This *is not* how the main application loads a plugin!

```
public class MainApplication                // Must be delivered by 31 January.
{
    public void doImportantThings()
    {
        AlicesPlugin p = new AlicesPlugin(); // Cannot be started until 1 March.
        p.doImportantPluginThings();
        ...
    }
}
```

This would be a paradox. The code for "`AlicesPlugin`" won't exist until months or years after the code for "`MainApplication`" is already running. So how can you possibly even know about `AlicesPlugin` when writing `MainApplication`? You can't, unless you have a time machine. There is no way for the `MainApplication` code to load plugins via their hard-coded names.

However, reflection can help you load a class given its name *as a string*. Thus, we don't need a hard-coded name. We just have to design `MainApplication` to do something like this:

1. Store the names of all installed plugins (or at least the names of each plugin's main class) in a database or text file. When the user installs a new plugin, `MainApplication` updates the database/file.
2. Read that database/file on start-up to recover the names of the currently-installed plugins.
3. Use reflection to load and instantiate the plugin classes, and call `start()` (or equivalent) on each object.

Reflection can generally help achieve flexibility not otherwise possible, especially when designing APIs. In the Android API, when we want to launch another activity (within the app), we write the construct:

Listing 4: Reflection has subtle uses elsewhere too.

```
startActivity(new Intent(this, MyOtherActivity.class));
```

"`MyOtherActivity.class`" is a runtime representation of the class `MyOtherActivity`. The API has never seen this class before, but we pass it a runtime *value* representing the class, so the API can still interact with it. It will create a new instance of `MyOtherActivity`, and call various methods on it.

2.2 Class Loading

The key to making a plugin-loading mechanism, in Java, is a segment of code like this:

Listing 5: Loading a class dynamically.

```
String pluginName = ...; // The main application must keep track of plugin
try                      // names in some way (e.g. a database or text file).
{
    Class<?> pluginClass = Class.forName(pluginName);
    Plugin pluginObj = (Plugin) pluginClass.getConstructor().newInstance();

    ... // Start the plugin
}
catch(...) {...} // Lots of possible errors; see below.
```

Notice the `Class` class. This is central to Java's reflection API. A `Class` object, such as `pluginClass` above, represents a class at runtime. That is, `pluginClass` could represent "`AlicesPlugin`" from Listing 3, but it could also represent any other plugin class.

The point is that we now have a general way to deal with classes dynamically – classes that will exist at runtime, but not at compile time. `Class.forName()` will load one of these classes given its name as a string.

Once we have a `Class` object, we can instantiate it by retrieving a `Constructor` object, and calling its `newInstance()` method. This is a bit like writing "`new AlicesPlugin()`", but again without the hard-coded name.

Some more finicky details:

- Why the "<?>"? `Class` has a generic type parameter that identifies *at compile time* which class it refers to. We obviously don't have that information at compile time, in this case, so we have to use the wildcard "?" instead.
- Similarly, we need the downcast "`(Plugin)`" because the compiler has no other way (yet) of knowing that we're actually getting a `Plugin` object.
- There are many things that might conceivably go wrong here, hence the `try-catch`. See Section 2.3 for the different kinds of exceptions that might occur.

There is slightly more to this in practice. `Class.forName()` only accepts a qualified class name like "`edu.curtin.AlicesPlugin`" (or just "`AlicesPlugin`" if it has no package). The JVM only looks for the actual class *files*, like "`AlicesPlugin.class`", within a particular set of directories called the **classpath**. If you give it a filename like "`C:\Program Files\App\MyPlugin.class`", all your hopes and dreams will be dashed.

If you do want to poke around at the lower levels of the API, the `ClassLoader` class lets you create additional ways to load classes. For instance, you could store your plugin classes in an arbitrary directory controlled by your application, or in an SQL database, or embedded within some other file format of your own creation. These all require a bit of extra work, of course.

2.3 Errors

There's a lot that can go wrong in reflection:

- We could try to load a class or a method that doesn't exist.
- We could supply the wrong number or type of parameters to a method or constructor.
- We could try to call a non-static method statically, or the other way around.
- We could call a non-static method on the wrong kind of object.

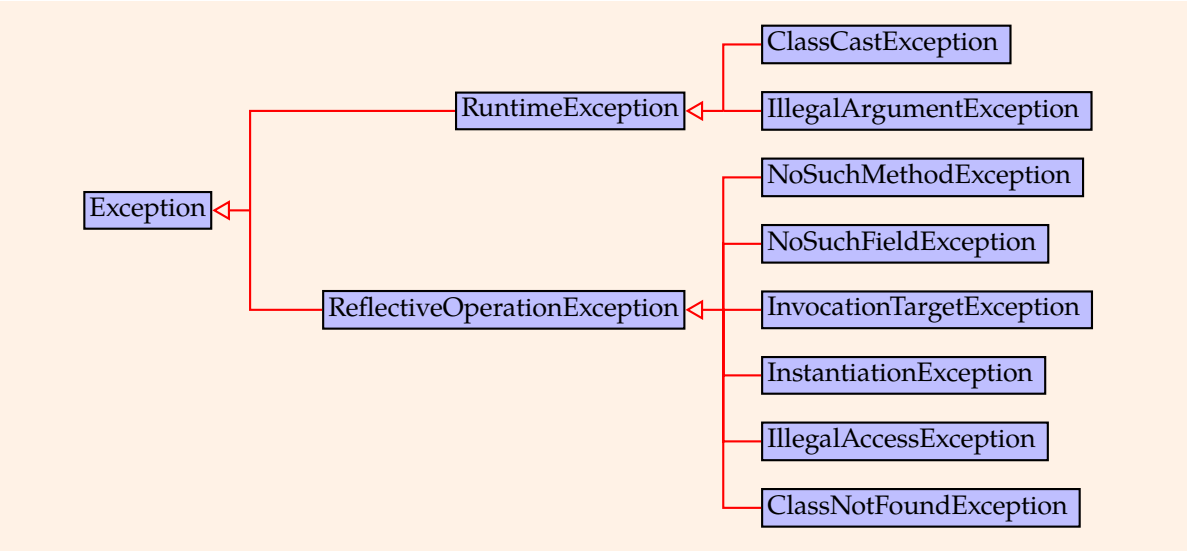

Figure 1: Java exceptions related to reflection.

- We could try to call a private, package-private or protected method from outside where it's allowed to be called.

Of course, we can *already* make all those mistakes in ordinary code, but then the compiler or IDE will pick them up right away. When using reflection, the compiler has no choice but to shrug its shoulders and assume you know what you're doing. It simply doesn't have enough information to sanity check these things for you.

Instead, the above mistakes (and others) will cause *exceptions* at runtime rather than compile errors. Figure 1 shows the exceptions that you're likely to encounter when using reflection. A lot of them inherit from `ReflectiveOperationException`, and you can usually deal with these in more-or-less the same way.

In most cases, these exceptions mean that the class/method you're dealing with doesn't conform to your expectations. In the case of running plugins, this probably means that the supplied plugin is broken in some way; i.e., it's not the main application's fault, but the plugin designer's (or maybe the user's). It does, of course, depend on the exact circumstances.

2.4 Reflection Broadly

By now, you already have enough information to develop a plugin mechanism. However, reflection runs a bit deeper than we've seen so far. We'll explore it a bit further, because it can give us additional options for designing a plugin API.

Some languages, notably Java and C#, have certain classes representing language constructs. For instance, the Java API defines the classes `Class`, `Constructor`, `Method`, `Field`, and others. The .NET API similarly defines `Type`, `ConstructorInfo`, `MethodInfo`, `FieldInfo` and others.

Objects of these classes represent instances of those language constructs. Hence, a `Class` object represents a class (or, in fact, any Java datatype, including interfaces, arrays and primitive types), and `Method` represents a method. For instance, we could write a method that accepts a `Class` object (for *any* class or datatype), displays its name, and displays its methods:

Listing 6: Java reflection.

```
public void queryClass(Class<?> cls)
{
    System.out.println(cls.getName()); // Display class name.
    for(Method m : cls.getMethods())
    {
        System.out.println(m.getName()); // Display each method.
    }
}
...
queryClass(SomeClass.class); // 'SomeClass.class' is the Class object
                             // representing SomeClass.
```

There is a great deal of interconnection amongst the reflection classes. For instance, `Field` contains a method `getType()` that returns `Class` (since `Class` is the standard runtime representation for all Java datatypes). `Method` has an array of `Class` values representing its parameter types, and one `Class` value for its return type. Figure 2 shows some these classes and relationships.

Other languages, notably Python, build querying capabilities directly into their own language constructs. In Python, modules, classes, methods and functions are themselves objects, with their own fields. You can pass them as arguments to other methods/functions, which may query their names and other details.

Listing 7: Python reflection.

```
def queryClass(cls):
    print(cls.__name__)         # Display class name
    for member in cls.__dict__:
        print(member.__name__)  # Display each method (and static field) name

...
queryClass(SomeClass) # Pass 'SomeClass' directly
                      # (since classes are themselves objects).
```

2.5 Traversing Reflection Objects

Figure 2 shows a great deal of interconnectivity between different reflection objects. First, `Class` objects themselves have various direct connections with other `Class` objects, which we can retrieve:

Listing 8: Retrieving class-to-class relationships.

```
Class<?> cls = ...;
Class<?> superclass = cls.getSuperclass(); // There's no 'getSubclasses()'
Class<?>[] interfaces = cls.getInterfaces();
Class<?>[] nestedClasses = cls.getClasses();
Class<?>[] outerClass = cls.getEnclosingClass();
```

Note that interfaces are also represented by `Class` objects, and so in fact is *every* non-generic Java type – arrays (e.g., `String[]`), primitives (e.g., `int`) and even `void`.

`Class` objects also aggregate a series of `Constructor`, `Method` and `Field` objects (plus `Annotation` objects, which we'll get back to in Section 3). We can retrieve arrays of these:

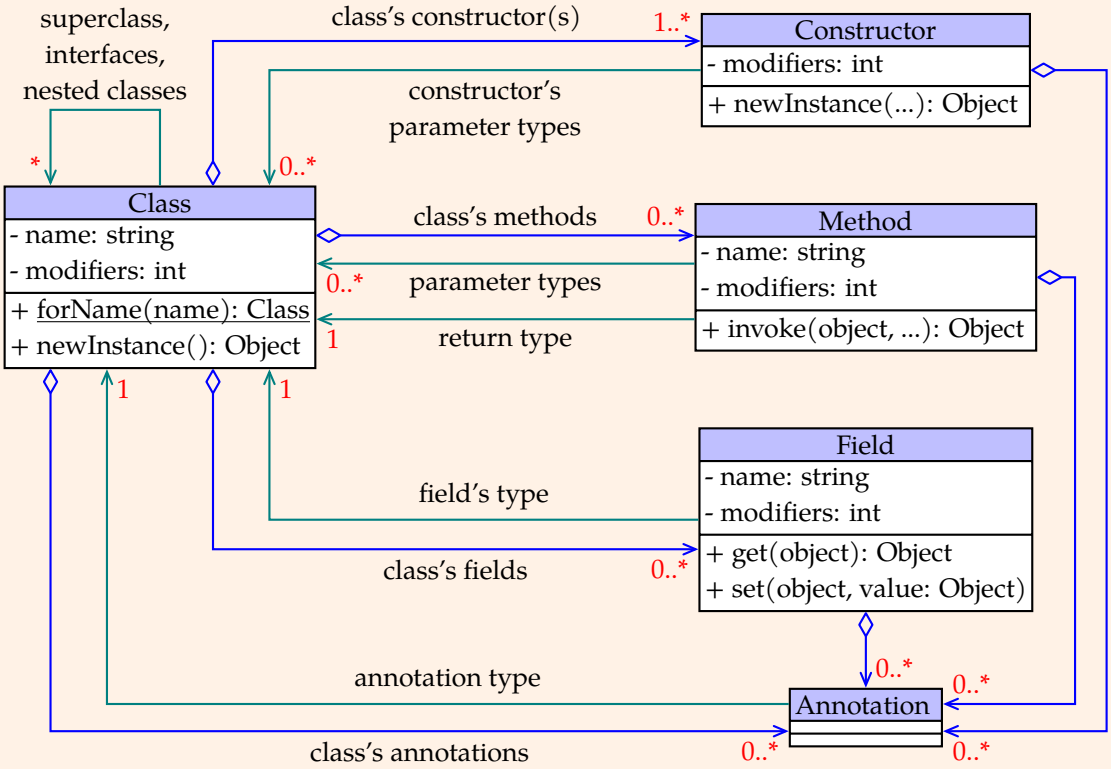

Figure 2: Key elements of the Java reflection API. *Note: this is not complete, and is more for illustration than reference.*

Listing 9: Retrieving all constructors, methods and fields.

```
Class<?> cls = ...;
Constructor<?>[] constructors = cls.getConstructors();
Method[] methods = cls.getMethods();
Field[] fields = cls.getFields();
```

We can also find specific constructors, methods, and fields, by name and parameters. As you probably know, Java allows overloaded constructors and methods. You can define as many constructors as you like, provided they have different sets of parameters. You can define as many methods with the same name as you like, with different parameters too.

So, to get one, we have to specify what the parameters are, and specifically the parameter *types*[a]. For instance:

[a] Parameter names are not preserved by the compilation process. They do not actually exist at runtime.

Listing 10: Retrieving specific `Constructor`, `Method` and `Field` objects.

```
public class ExampleClass // Say we have this class. (In practice, you won't
{                         // know exactly what class you'll be dealing with!)
    private boolean flag;

    public ExampleClass() {...}
    public ExampleClass(int[] intArray) {...}

    public static void method1(int i, String s) { ... }
    public String method2()                     { ... }
    public String method2(List<String> list)    { ... }
}
```

```
// Retrieve specific reflection objects.
Class<?> cls = Class.forName("ExampleClass");

Field f1 = cls.getField("flag");

Constructor c1 = cls.getConstructor();
Constructor c2 = cls.getConstructor(int[].class);

Method m1 = cls.getMethod("method1", int.class, String.class);
Method m2a = cls.getMethod("method2");
Method m2b = cls.getMethod("method2", List.class);
```

Notice that we can specify the constructor/method parameters with the syntax "`<type>.class`". This syntax is called a **class literal**, and it evaluates to a `Class` object representing that type. It's a bit like `Class.forName()`, but for when we already know the type at compile time. (We cannot write "`variableName.class`" – the compiler doesn't understand this.)

Listing 10 is a bit idealistic. If we knew what the constructors, methods and fields were in advance, we wouldn't need reflection! In practice, we need to *check* what exists:

Listing 11: Checking whether a method exists.

```
Method m = cls.getMethod("method1", int.class, String.class);
if(m == null)
{
    throw new MyException("'method1(int,String)' does not exist");
}
... // The method does exist. Do something with it.
```

Once you have a `Constructor`, `Method` or `Field` object, you can extract further information from them. For instance:

- `getName()` is fairly self explanatory (but doesn't apply to `Constructor`).
- `isAccessible()` will return `true` if you're actually allowed to call/access the object (according to the access modifiers `public`/`protected`/`private`).
- `getModifiers()` returns an integer whose individual bits represent the presence (1) or absence (0) of all the potential modifiers – `public`, `static`, `synchronized`, `final`, `native`, etc.

  You can extract these yourself with bit manipulation (`&`, `>>`, etc.), or you can use the `Modifiers` class; e.g.:

```
Method m = ...;
boolean staticMethod = Modifier.isStatic(m.getModifiers());
```

- `Method` in particular has `getReturnType()`, `getParameterTypes()` and `getExceptionTypes()`. These return the various parts of the method signature as `Class<?>` objects (`Class<?>[]`, in the case of parameters and exceptions).
- `Field` has `getType()`, which returns a `Class<?>` representing the field's datatype.

2.6 Calling Via Reflection

Reflection is not just about sightseeing. Once we have a `Method` or `Constructor` object, we can *call* it. We've already done this with `Constructor.newInstance()`. Meanwhile, the `Method` class has an `invoke()` method, which we use to call the underlying method.

`Constructor.newInstance()` just takes the same parameters as the constructor itself:

Listing 12: Constructor.-newInstance().

```
try
{
    Constructor<?> constructor = cls.getConstructor(int.class, String.class);
    if(constructor == null) { throw ... }

    Object newObject = constructor.newInstance(5, "Ralph");

    // This is conceptually like:
    // Object newObject = new ClassName(5, "Ralph");

    ...
}
catch(...) {...}
```

`Method.invoke()` is similar, but takes an extra first parameter. If you're calling a `static` method, this is simply `null`. So:

Listing 13: Method.invoke() for `static` methods.

```
try
{
    Method method = cls.getMethod("methodName", int.class, String.class);
    if(method == null ||
       !Modifier.isStatic(method.getModifiers())) { throw ... }

    Object returnValue = method.invoke(null, 5, "Ralph");

    // This is conceptually like:
    // Object returnValue = ClassName.methodName(5, "Ralph");

    ...
}
catch(...) {...}
```

If we don't expect the method to return anything, we can omit "`Object returnValue =`".

In the case of non-`static` (or "instance") methods, that extra parameter must be the object that

the method is being called on:

Listing 14: Method.invoke() for non-static (instance) methods.

```
try
{
    Constructor constructor = cls.getConstructor(...);
    Method method = cls.getMethod("methodName", int.class, String.class);
    if(constructor == null || method == null ||
       Modifier.isStatic(method.getModifiers())) { throw ... }

    Object newObject = constructor.newInstance(...);
    Object returnValue = method.invoke(newObject, 5, "Ralph");

    // This is conceptually like:
    // Object newObject = new ClassName(...);
    // Object returnValue = newObject.methodName(5, "Ralph");

    ...
}
catch(...) {...}
```

2.7 Smarter Plugin Mechanisms

The details of `Class`, `Constructor`, `Method` and `Field` are relevant to plugin creation. For instance, we now have an alternative way to implement certain kinds of callbacks.

Instead of registering callbacks, the plugin might simply define them with specially constructed names. For instance:

Listing 15: Name-based callbacks.

```
public class CLVWeatherPlugin implements WeatherPlugin
{
    public void start(Control api) {...}

    public WeatherData getCambodiaWeatherData() {...}
    public WeatherData getLaosWeatherData() {...}
    public WeatherData getVietnamWeatherData() {...}
}
```

Say that the main application knows which country the user is in, and needs to acquire weather data from that country's weather bureau. It could use reflection to find the right callback:

Listing 16: Finding a name-based callback.

```
// Main application.
List<WeatherPlugin> listOfPlugins = ...;
String country = ...;
WeatherData data;

for(WeatherPlugin plugin : listOfPlugins)
{
    Method m = plugin.getClass().getMethod("get" + country + "WeatherData");
    if(m != null && !Modifiers.isStatic(m.getModifiers()))
    {
        data = (WeatherData) m.invoke(plugin);
        break;
    }
}
```

For a given country, this code will find the first plugin that provides a `get<country>WeatherData()` method, and will call it. This avoids having to create a callback interface, or callback registration mechanism.

(Note that all objects have a `getClass()` method, which returns a `Class` object representing the original object's class.)

3 Annotations

An **annotation** (in Java) is one of several standard or customisable labels, starting with "`@`", that you can put in front of declarations. You might be familiar with some:

Listing 17: Annotations in Java.

```
// JUnit test method.
@Test
public void testThing() { ... }

// Mark as not being used anymore.
@Deprecated
public void update() { ... }
```

C# has a different name (**attribute**) and syntax for the same thing:

Listing 18: Attributes in C#.

```
// NUnit test method
[Test]
public void TestThing() { ... }

// Mark as not being used anymore.
[Obsolete]
public void Update() { ... }
```

3.1 How They Work (Briefly)

You can create your own annotation types as follows:

Listing 19: Defining a simple annotation.

```
public @interface MyAnnotation {}                    // MyAnnotation.java
```

Despite the keyword, this *isn't* an interface. It's more like a special and very limited kind of class[b]. Nonetheless, with an annotation type defined, we can put it in front of declarations:

Listing 20: Using a custom annotation.

```
public class MyClientCode
{
    @MyAnnotation
    public void method1() { ... }

    @MyAnnotation
    public void method2() { ... }
}
```

The compiler will accept this, and will create two objects of type `MyAnnotation`, each attached to a different method. But their purpose won't be very clear yet. You see, annotations don't actually *do* anything by themselves. They are just labels, or "metadata".

What we need is *another* piece of code to "process" annotations in some way, using reflection. For instance:

Listing 21: Example for dynamically retrieving annotated methods from a class.

```
import java.lang.reflect.*;

public class ProviderProcessor
{
    public void process(Class<?> cls)
    {
        for(Method m : cls.getMethods())
        {
            Annotation a = m.getAnnotation(MyAnnotation.class);
            if(a != null)
            {
                System.out.println("Method " + m + " is annotated");
            }
        }
    }
}
```

Things will kick off when `process()` is called with a value of `MyClientCode.class`. (How and when this happens is a higher-level design decision.) The method will iterate over the class's `Methods`, and check whether each method has a `MyAnnotation` annotation. If it does, we can choose to do anything we like with that information.

3.2 JUnit's Annotations

But *why* use annotations?

[b] Annotations were added some time after Java was originally created, but the language designers knew that they couldn't just introduce a new Java keyword like "`annotation`", because thousands of projects might already use that as the name of a field, method, etc.

Standard annotations like `@Override`, `@Deprecated` and `@SuppressWarnings` are there to give hints to the compiler – important to use, but you're not going to be *defining new ones* very often.

You're much more likely to use annotations as a kind of callback registration system – a more sophisticated version of what we did in Section 2.7. JUnit (the popular unit testing framework) is a good example of this. It defines a few key annotations, such as:

- `@Test`, to label a test method.
- `@Before` and `@After`, to indicate setup and tear-down methods.
- `@RunWith`, to indicate a test suite class.

This is really a kind of API. A developer uses JUnit's annotations to tell JUnit which methods to call, and for what reasons. To see the value of this, imagine if we didn't have annotations. How would a developer tell JUnit which test methods to run? Perhaps like this:

Listing 22: Fictional unit testing, by specifying the names of test methods.

```
public class TestSuite
{
    public TestSuite()
    {
        TestFramework.registerTests("testThing1", "testThing2");
        TestFramework.registerBefore("setup");
        TestFramework.registerAfter("tearDown");
    }

    public void setup() { ... }
    public void testThing1() { ... }
    public void testThing2() { ... }
    public void tearDown() { ... }
}
```

Here, the test framework would find the test/setup/teardown methods by name, which can be done using reflection. We could also use method references:

Listing 23: Fictional unit testing, by specifying method references (callbacks).

```
public class TestSuite
{
    public TestSuite()
    {
        TestFramework.registerTests(this::testThing1, this::testThing2);
        TestFramework.registerBefore(this::setup);
        TestFramework.registerAfter(this::tearDown);
    }

    public void setup() { ... }
    public void testThing1() { ... }
    public void testThing2() { ... }
    public void tearDown() { ... }
}
```

This lets the compiler stop us making spelling mistakes, *but* it also deprives the test framework of the name of each method. It can *call* `testThing1()`, for instance, but it can't properly *log* it or *show* the user that it's doing so.

In any case, both these "registration" approaches are clumsier than using annotations:

Listing 24: Actual JUnit test suite.

```
public class TestSuite
{
    @Before public void setup() { ... }
    @Test public void testThing1() { ... }
    @Test public void testThing2() { ... }
    @After public void tearDown() { ... }
}
```

3.3 Annotations With Data

Annotations can add more sophisticated information too. Say we wanted to annotate things with the contact details (name, email address, etc.) of the author.

Listing 25: Defining an annotation with data.

```
public @interface Contact
{
    String name();
    String email() default "";
    int id() default 0;
}
```

An annotation (`@interface`) declaration contains a series of accessor methods, with no arguments and no code. The syntax shown in Listing 25 results in something conceptually like this:

Listing 26: Conceptual annotation declaration. You can't actually write annotation types this way, but this is essentially what's happening.

```
public class Contact implements Annotation
{
    private String name;
    private String email = "";
    private int id = 0;
    ... // Constructors and standard annotation methods

    public String name()  { return name; }
    public String email() { return email; }
    public int id()       { return id; }
}
```

When it comes to using the annotation, you must specify all values that *don't* have defaults, and optionally any values that do:

Listing 27: Using an annotation with data.

```
public class MyClientCode
{
    @Contact(name = "Phil")
    public void method1() { ... }

    @Contact(
        name = "Hal",
        email = "hal@example.com",
        id = 99
    )
    public void method2() { ... }
}
```

We can also do this with several annotations, one for each kind of data:

Listing 28

```
public @interface Name
{
    String value(); // Annotations with one value should call it 'value'.
}

public @interface Email
{
    String value();
}

public @interface Id
{
    int value();
}
```

Listing 29

```
public class MyClientCode
{
    @Name("Phil") // Can omit 'value = ' if only one value.
    public void method1() { ... }

    @Name("Hal")
    @Email("hal@example.com")
    @Id(99)
    public void method2() { ... }
}
```

3.4 Plugin Annotations

Getting back to the main topic at hand, how can we use annotations as part of a plugin mechanism?

Let's revisit Listing 15, where we have the method names `getCambodiaWeatherData()`, etc. What we're doing here is embedding certain *metadata* into the method name. The metadata (the country name, in this case) lets the main application know when to run the method. This works, but what if the metadata starts becoming complicated?

Say that each country's weather bureau has restrictions on how often you can look for updates. Perhaps in Country A you're allowed to do it every 10 minutes, and in country B every 1 hour. As well as this, perhaps the application is required to display the name of the weather bureau that the data came from. There needs to be a way for the plugin to tell the main application about this.

We might have a `get_Cambodia_45_MinistryOfWaterResourcesAndMeteorology_WeatherData()` method. But you can see the problem – method names were never really designed to contain data. It would be much nicer to use annotations. For instance:

Listing 30: Name-based callbacks.

```
public class CLVWeatherPlugin implements WeatherPlugin
{
    public void start(Control api) {...}

    @DataSource(country = "Cambodia",
                frequency = 45,
                agency = "Ministry of Water Resources and Meteorology")
    public WeatherData getCambodiaWeatherData() {...}

    ...
}
```

Now the name of the method is purely decorative, and all the information is in the `@DataSource` annotation. The main application would define `@DataSource`. It would also search through the methods in `CLVWeatherPlugin`, looking for methods with this annotation. Hopefully you'll agree that this is a much more elegant approach.

However, there is a slight functionality tradeoff when compared with the `DataCollector` interface from Listing 2. Using more traditional OO design techniques (as in Listing 2), our plugins have the flexibility to register and unregister themselves dynamically, whereas the annotation approach only allows the developer to nominate a callback at compile time. This compromise may be worth it, for the sake of readability, but it is a compromise.

3.5 Limitations

Do not think of annotations as anything more than a simple bundle of data. In particular, they can *only* contain the following types of data [4, Section 9.6.1]:

- Primitive types (`int`, `boolean`, etc.);
- `String`;
- `Class` (yes, you can have an annotation that refers to a datatype);
- An `enum` type;[7]
- Another annotation type;
- An single-dimensional array of one of the previous types (not an array of arrays).

3.6 Side Note: Python's Decorators

Python has something it calls a *decorator*. You may recall the Decorator design pattern, and though Python's decorators don't *exactly* follow the pattern, the similarity is significant.

Python decorators use the "@" symbol followed by an identifier, just like Java's annotations, and can be used to achieve the same things. But they work quite differently.

Take this example:

Listing 31: Python decorator.

```
def my_decorator(func):
    ...

@my_decorator
def other_function(...):
    ...
```

Notice that the decoration `@my_decorator` refers to the *function* `my_decorator()`. This is what happens, conceptually, when you use a decorator:

Listing 32: What `@my_decorator` actually does in Python.

```
other_function = my_decorator(other_function)
```

The decorator function is called, taking the original function as a parameter, and returning *another* function that (generally) wraps around the original function in order to modify its behaviour. You need to understand something about functional programming (particularly closures) in order to make use of this.

However, the decorator *could* return the original function unmodified, and just record some information about it (and a reference to it) elsewhere. This is more comparable to what happened in Listing 21.

4 Scripting

Implementing a scripting mechanism doesn't (necessarily) require reflection. Instead, we are more likely to do the following:

1. Read the text of the script code (e.g., from a file or database).
2. Create/initialise an interpreter.
3. Bind one or more API objects/functions to the interpreted environment.
4. Tell the interpreter to run the script.

Some examples of interpreters you might use for scripting purposes:

- Jython[1] is a JVM-based Python interpreter/compiler. If you want a pure JVM application, but with the ability to run Python scripts, this may be what you're after.

  Unfortunately, at the time of writing, Jython still only supports version 2.7 of the Python language, and it is unclear if/when it will begin supporting any 3.x version. Nonetheless, it is still a reasonably simple way to demonstrate the idea of scripting.

- Jpy,[2] Py4J[3] and Jep[5] are alternative libraries to help run Python code from Java.

- CPython and PyPy are both Python implementations that can be accessed from native code using the "C Foreign Function Interface for Python".[9]

- `javax.script.ScriptEngine`[8] is a scripting framework built into the standard Java API. It theoretically supports multiple languages, but the standard JVM only provides a JavaScript interpreter. At the time of writing, this JavaScript interpreter has unfortunately been deprecated (due to the logistics of maintaining it) and is scheduled for removal in a future Java release.[6]

- Apache Commons BSF™(Bean Scripting Framework)[10] is an alternative scripting framework for Java that supports various languages, including Groovy and Ruby.

Listing 33: Jython: Java code executing a Python script.

```
import org.python.core.*;
import org.python.util.*;

public class ScriptHandler
{
    public void runScript(Control api, String pythonScript)
    {
        // Initialise the interpreter
        PythonInterpreter interpreter = new PythonInterpreter();

        // Bind the API to the script environment
        interpreter.set("api", api);

        // Run the script
        interpreter.exec(pythonScript);
    }
}
```

- V8[11] is a C++-based Javascript interpreter. It is used by Google Chrome, but can be used in any C++ application.

Whichever one we choose, our build system should be able to resolve a dependency. For instance, for Jython, we need to add "`implementation 'org.python:jython-standalone:2.7.1'`" to the `dependencies{}` section of your `build.gradle`. (The "standalone" version is the one designed to be embedded in another application.)

Listing 33 shows how to invoke a Python script using Jython. A key concept here is the **binding** of objects from the main application to the script environment. We can choose Java objects (`api`, in Listing 33) to appear as global variables from within the Python code. This is how the script is able to access the API.

There are some typing differences between Java and Python, just as there are between Java and C, though not as drastic, and the interpreter (Jython) is able to "smooth" them over. It can allow the Python code to access a Java object as if it was a Python object. It can even allow the Python code to extend a Java class/interface, create an object of it, and have Java recognise that object as an instance of the original class/interface.

In fact, this is precisely how we can arrange for scripts to register callbacks. A Python script executed by the `ScriptHandler` class in Listing 33 could look like that shown in Listing 34. The script can access the API via `api` variable, and use it to register its callbacks.

Apart from the necessary inclusion of an interpreter, managing scripts is perhaps easier than managing plugins. You can store scripts however you like. From the main application's point of view, they are simply plain text that it feeds into the interpreter. They could therefore be stored in a directory that the main application controls, or a database.

5 Further Reading

- Oracle, The Java™ Tutorials: Trail: The Reflection API, https://docs.oracle.com/javase/tutorial/reflect/index.html.

Listing 34: Jython: A Python script, registering callbacks using the API.

```
# Start-up logic
print("Now starting WeatherAlert script...")

# Define a Python class that implements one of our API callback interfaces
class MyForecastHandler(edu.curtin.ForecastHandler):
    def forecastChanged(newForecast):
        # Take some action
        if newForecast.getDay(0).isStormy():
            print("STORM ALERT")

# Define some more callbacks
class TunisiaDataCollector(edu.curtin.DataCollector):
    def collect():
        ... # Get data from somewhere
        return WeatherData(...)

class AlgeriaDataCollector(edu.curtin.DataCollector):
    def collect():
        ...

# Make new objects of these classes
handler = MyForecastHandler()
tCollector = TunisiaDataCollector()
aCollector = AlgeriaDataCollector()

# Register them with the API
api.registerForecastHandler(handler)
api.registerDataCollector("Tunisia", tCollector)
api.registerDataCollector("Algeria", aCollector)
```

References

[1] Jython. https://www.jython.org/. Accessed: 2019-10-07. [**See section 4.**]

[2] Brockmann Consult Development. bcdev/jpy: A bi-directional python-java bridge used to embed java in cpython or the other way round. https://github.com/bcdev/jpy. Accessed: 2019-10-07. [**See section 4.**]

[3] Barthélémy Dagenais. Welcome to py4j. https://www.py4j.org/, 2015. Accessed: 2019-10-07. [**See section 4.**]

[4] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Daniel Smith. The Java Language Specification: Java SE 11 Edition. https://docs.oracle.com/javase/specs/jls/se11/html/index.html, 2018. Accessed: 2019-10-07. [**See section 3.5.**]

[5] Jep Team. ninia/jep: Embed Python in Java. https://github.com/ninia/jep. Accessed: 2019-10-07. [**See section 4.**]

[6] Jim Laskey. JEP 355: Deprecate the Nashorn JavaScript engine. https://openjdk.java.net/jeps/335, 2018. Accessed: 2019-10-07. [**See section 4.**]

[7] Oracle. The java™ tutorials: Enum types. https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html. Accessed: 2019-10-07. [**See section 3.5.**]

[8] Oracle. Java scripting programmer's guide. https://docs.oracle.com/en/java/javase/11/scripting/java-scripting-programmers-guide.pdf, 2018. Accessed: 2019-10-07. [**See section 4.**]

[9] Armin Rigo and Maciej Fijalkowski. Using CFFI for embedding. https://cffi.readthedocs.io/en/latest/embedding.html, 2018. Accessed: 2019-10-07. [**See section 4.**]

[10] The Apache Software Foundation. Apache commons BSF™(bean scripting framework). https://commons.apache.org/proper/commons-bsf/index.html. Accessed: 2019-10-07. [**See section 4.**]

[11] V8 Project Authors. Getting started with embedding V8. https://v8.dev/docs/embed. Accessed: 2019-10-07. [**See section 4.**]