Lab 10. Annotations and Reflections

> Code examples and part of the tutorial are based on Chapter 8, Core Java Volume II by Cay Horstmann.
>
> Author: Yida Tao

Preparation

By running `ButtonTest.java` with `ButtonFrame.java`, you should get a panel with three buttons; pressing each button changes the panel background color accordingly.

The primary code for specifying button behavior is: `yellowButton.addActionListener((e)->yellowBackground());`, where the lambda states which method should be executed when the source button is clicked. If you comment out the three event registration statements, nothing will happen when you click the button.

Let's see how we could dynamically (i.e., at runtime) add the event handlers using annotations and reflections.

Define a customized annotation interface

See `ActionListenerFor.java`. This annotation can annotate methods and can be accessed at runtime through reflections.

Analyzing the customized annotation

The `ActionListenerFor` annotation doesn't do anything by itself. It sits in the source file. The compiler places it in the class file, and the virtual machine loads it. We now need a mechanism to analyze it and install action listeners. That is the job of the `ActionListenerInstaller.java`.

The `ButtonFrame` constructor calls `ActionListenerInstaller.processAnnotations(this)`. The static `processAnnotations` method enumerates all methods of the object it received. For each method, it gets the `ActionListenerFor` annotation object and processes it.

```
Class<?> cl = obj.getClass();
for (Method m : cl.getDeclaredMethods())
{
   // @ActionListenerFor has RetentionPolicy.RUNTIME,
   // therefore we could access this annotation through reflection
   ActionListenerFor a = m.getAnnotation(ActionListenerFor.class);
   if (a != null)
   {
       ...
   }
}
```

The name of the source field is stored in the annotation object. We retrieve it by calling the source method, and then look up the matching field:

```
        Field f = cl.getDeclaredField(a.source());
        f.setAccessible(true);
```

Then we add the method `m` as an action listener for the corresponding field in `addListener(f.get(obj), obj, m)`. Specifically, this `addListener` method should do something like `f_instance.addActionListener((e)->m())`, where `(e)->m()` denotes that we need to "instantiate" a (functional) interface **at runtime**, which could be achieved using `java.lang.reflect.Proxy`.

The proxy class can create brand-new classes at runtime. Such a proxy class implements the interfaces that you specify. In particular, the proxy class has the following methods:

- All methods required by the specified interfaces; and
- All methods defined in the Object class (toString, equals, and so on).

However, you cannot define new code for these methods at runtime. Instead, you must supply an invocation handler. An invocation handler is an object of any class that implements the `InvocationHandler` interface, which has a single method `Object invoke(Object proxy, Method method, Object[] args)`. For more details, please refer to Core Java Volume I, Chapter 6.

```
   public static void addListener(Object source, final Object param, final Method
m)
         throws ReflectiveOperationException
   {
      var handler = new InvocationHandler()
         {
            public Object invoke(Object proxy, Method mm, Object[] args) throws
Throwable
            {
               return m.invoke(param);
            }
         };

      Object listener = Proxy.newProxyInstance(null,
         new Class[] { ActionListener.class }, handler);

      Method adder = source.getClass().getMethod("addActionListener",
ActionListener.class);
      adder.invoke(source, listener);
   }
```

Here we created a proxy object that implements the `ActionListener` interface with a given `handler`. Whenever a method is called on the proxy object at runtime, the `invoke` method of the invocation handler gets called, which figures out how to handle the call. In our case, when `adder.invoke(source, listener)` executes, it actually performs something like `yellowButton.addActionListener((e)->yellowBackground())`, except that the `(e)->yellowBackground()` part, which implements the `actionPerformed` method in the `ActionListener` (functional) interface, is now handled by `invoke` in the given `handler`.

Rerun the annotated program

Now, let's annotate each event methods with the customized annotation:

```
@ActionListenerFor(source = "yellowButton")
public void yellowBackground()
{
  panel.setBackground(Color.YELLOW);
}

@ActionListenerFor(source = "blueButton")
public void blueBackground()
{
  panel.setBackground(Color.BLUE);
}

@ActionListenerFor(source = "redButton")
public void redBackground()
{
  panel.setBackground(Color.RED);
}
```

We can replace the `addActionListener` method calls with `ActionListenerInstaller.processAnnotations(this);`

```
//      yellowButton.addActionListener((e)->yellowBackground());
//      blueButton.addActionListener((e)->blueBackground());
//      redButton.addActionListener((e)->redBackground());
ActionListenerInstaller.processAnnotations(this);
```

Now, let's re-execute the program. You'll see that the buttons still behave correctly.