Lab 11. 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, Yao Zhao

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.

```java
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())`.

```java
    public static void addListener(Object source, final Object param, final
Method m)
        throws ReflectiveOperationException {

        // source is the button
        // this means "button.addActionListener(listener)"
        Method adder = source.getClass().getMethod("addActionListener",
ActionListener.class);
        adder.invoke(source, (ActionListener) e -> {
            try {
                m.invoke(param);
            } catch (IllegalAccessException ex) {
                throw new RuntimeException(ex);
            } catch (InvocationTargetException ex) {
                throw new RuntimeException(ex);
            }
        });
    }
```

Here, `source` is the button, when `adder.invoke(source, (ActionListener) e ->m.invoke(param))` executes, it actually performs something like `yellowButton.addActionListener((e)->yellowBackground())`.

Rerun the annotated program

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

```java
  @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.