Documentation

The Java™ Tutorials

Trail: The Reflection API
Lesson: Members
Section: Constructors

Creating New Class Instances

There are two reflective methods for creating instances of classes: `java.lang.reflect.Constructor.newInstance()` and `Class.newInstance()`. The former is preferred and is thus used in these examples because:

- `Class.newInstance()` can only invoke the zero-argument constructor, while `Constructor.newInstance()` may invoke any constructor, regardless of the number of parameters.
- `Class.newInstance()` throws any exception thrown by the constructor, regardless of whether it is checked or unchecked. `Constructor.newInstance()` always wraps the thrown exception with an `InvocationTargetException`.
- `Class.newInstance()` requires that the constructor be visible; `Constructor.newInstance()` may invoke `private` constructors under certain circumstances.

Sometimes it may be desirable to retrieve internal state from an object which is only set after construction. Consider a scenario where it is necessary to obtain the internal character set used by `java.io.Console`. (The `Console` character set is stored in an private field and is not necessarily the same as the Java virtual machine default character set returned by `java.nio.charset.Charset.defaultCharset()`). The `ConsoleCharset` example shows how this might be achieved:

```
import java.io.Console;
import java.nio.charset.Charset;
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import static java.lang.System.out;

public class ConsoleCharset {
    public static void main(String... args) {
        Constructor[] ctors = Console.class.getDeclaredConstructors();
        Constructor ctor = null;
        for (int i = 0; i < ctors.length; i++) {
            ctor = ctors[i];
            if (ctor.getGenericParameterTypes().length == 0)
                break;
        }

        try {
            ctor.setAccessible(true);
            Console c = (Console)ctor.newInstance();
            Field f = c.getClass().getDeclaredField("cs");
            f.setAccessible(true);
            out.format("Console charset         :  %s%n", f.get(c));
            out.format("Charset.defaultCharset():  %s%n",
                       Charset.defaultCharset());

        // production code should handle these exceptions more gracefully
        } catch (InstantiationException x) {
            x.printStackTrace();
        } catch (InvocationTargetException x) {
            x.printStackTrace();
        } catch (IllegalAccessException x) {
            x.printStackTrace();
        } catch (NoSuchFieldException x) {
            x.printStackTrace();
        }
    }
}
```

Note:

`Class.newInstance()` will only succeed if the constructor is has zero arguments and is already accessible. Otherwise, it is necessary to use `Constructor.newInstance()` as in the above example.

Example output for a UNIX system:

```
$ java ConsoleCharset
Console charset          :  ISO-8859-1
Charset.defaultCharset() :  ISO-8859-1
```

Example output for a Windows system:

```
C:\> java ConsoleCharset
Console charset          :  IBM437
Charset.defaultCharset() :  windows-1252
```

Another common application of `Constructor.newInstance()` is to invoke constructors which take arguments. The `RestoreAliases` example finds a specific single-argument constructor and invokes it:

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import static java.lang.System.out;

class EmailAliases {
    private Set<String> aliases;
    private EmailAliases(HashMap<String, String> h) {
        aliases = h.keySet();
    }

    public void printKeys() {
        out.format("Mail keys:%n");
        for (String k : aliases)
            out.format("  %s%n", k);
    }
}

public class RestoreAliases {

    private static Map<String, String> defaultAliases = new HashMap<String, String>();
    static {
        defaultAliases.put("Duke", "duke@i-love-java");
        defaultAliases.put("Fang", "fang@evil-jealous-twin");
    }

    public static void main(String... args) {
        try {
            Constructor ctor = EmailAliases.class.getDeclaredConstructor(HashMap.class);
            ctor.setAccessible(true);
            EmailAliases email = (EmailAliases)ctor.newInstance(defaultAliases);
            email.printKeys();

        // production code should handle these exceptions more gracefully
        } catch (InstantiationException x) {
            x.printStackTrace();
        } catch (IllegalAccessException x) {
            x.printStackTrace();
        } catch (InvocationTargetException x) {
            x.printStackTrace();
        } catch (NoSuchMethodException x) {
            x.printStackTrace();
        }
    }
}
```

This example uses `Class.getDeclaredConstructor()` to find the constructor with a single argument of type `java.util.HashMap`. Note that it is sufficient to pass `HashMap.class` since the parameter to any `get*Constructor()` method requires a class only for type purposes. Due to type erasure, the following expression evaluates to `true`:

```
HashMap.class == defaultAliases.getClass()
```

The example then creates a new instance of the class using this constructor with `Constructor.newInstance()`.

```
$ java RestoreAliases
Mail keys:
  Duke
  Fang
```

Previous page: Retrieving and Parsing Constructor Modifiers
Next page: Troubleshooting