CORE JAVA

Volume I—Fundamentals

ELEVENTH EDITION

CAY S. HORSTMANN

Core Java

Volume I—Fundamentals

Eleventh Edition

Alternatively, you can see how the program will behave in a future version of Java, by running:

```
java --illegal-access=deny objectAnalyzer/ObjectAnalyzerTest
```

Then the program will simply fail with an `IllegalAccessException`.

NOTE: It is possible that future libraries will use *variable handles* instead of reflection for reading and writing fields. A `VarHandle` is similar to a `Field`. You can use it to read or write a specific field of any instance of a specific class. However, to obtain a `VarHandle`, the library code needs a `Lookup` object:

```
public Object getFieldValue(Object obj, String fieldName, Lookup lookup)
      throws NoSuchFieldException, IllegalAccessException
{
   Class<?> cl = obj.getClass();
   Field field = cl.getDeclaredField(fieldName);
   VarHandle handle = MethodHandles.privateLookupIn(cl, lookup)
      .unreflectVarHandle(field);
   return handle.get(obj);
}
```

This works provided the `Lookup` object is generated in the module that has the permission to access the field. Some method in the module simply calls `MethodHandles.lookup()`, which yields an object encapsulating the access rights of the caller. In this way, one module can give permission for accessing private members to another module. The practical issue is how those permissions can be given with a minimum of hassle.

While we can still do so, let us look at a generic `toString` method that works for *any* class (see Listing 5.15). The generic `toString` method uses `getDeclaredFields` to obtain all data fields and the `setAccessible` convenience method to make all fields accessible. For each field, it obtains the name and the value. Each value is turned into a string by recursively invoking `toString`.

The generic `toString` method needs to address a couple of complexities. Cycles of references could cause an infinite recursion. Therefore, the `ObjectAnalyzer` keeps track of objects that were already visited. Also, to peek inside arrays, you need a different approach. You'll learn about the details in the next section.

You can use this `toString` method to peek inside any object. For example, the call

```
var squares = new ArrayList<Integer>();
for (int i = 1; i <= 5; i++) squares.add(i * i);
System.out.println(new ObjectAnalyzer().toString(squares));
```

yields the printout

```
java.util.ArrayList[elementData=class java.lang.Object[]{java.lang.Integer[value=1][][],
java.lang.Integer[value=4][][],java.lang.Integer[value=9][][],
   java.lang.Integer[value=16][][],
java.lang.Integer[value=25][][],null,null,null,null,null},size=5][modCount=5][][]
```

You can use this generic `toString` method to implement the `toString` methods of your own classes, like this:

```
public String toString()
{
   return new ObjectAnalyzer().toString(this);
}
```

This is a hassle-free and undoubtedly useful method for supplying a universal `toString` method. However, before you get too excited about never having to implement `toString` again, remember that the days of uncontrolled access to internals are numbered.

Listing 5.15 objectAnalyzer/ObjectAnalyzerTest.java

```
1  package objectAnalyzer;
2
3  import java.util.*;
4
5  /**
6   * This program uses reflection to spy on objects.
7   * @version 1.13 2018-03-16
8   * @author Cay Horstmann
9   */
10 public class ObjectAnalyzerTest
11 {
12    public static void main(String[] args)
13          throws ReflectiveOperationException
14    {
15       var squares = new ArrayList<Integer>();
16       for (int i = 1; i <= 5; i++)
17          squares.add(i * i);
18       System.out.println(new ObjectAnalyzer().toString(squares));
19    }
20 }
```

Listing 5.16 objectAnalyzer/ObjectAnalyzer.java

```
1  package objectAnalyzer;
2
3  import java.lang.reflect.AccessibleObject;
4  import java.lang.reflect.Array;
```

```
5  import java.lang.reflect.Field;
6  import java.lang.reflect.Modifier;
7  import java.util.ArrayList;
8
9  public class ObjectAnalyzer
10 {
11    private ArrayList<Object> visited = new ArrayList<>();
12
13    /**
14     * Converts an object to a string representation that lists all fields.
15     * @param obj an object
16     * @return a string with the object's class name and all field names and values
17     */
18    public String toString(Object obj)
19          throws ReflectiveOperationException
20    {
21       if (obj == null) return "null";
22       if (visited.contains(obj)) return "...";
23       visited.add(obj);
24       Class cl = obj.getClass();
25       if (cl == String.class) return (String) obj;
26       if (cl.isArray())
27       {
28          String r = cl.getComponentType() + "[]{";
29          for (int i = 0; i < Array.getLength(obj); i++)
30          {
31             if (i > 0) r += ",";
32             Object val = Array.get(obj, i);
33             if (cl.getComponentType().isPrimitive()) r += val;
34             else r += toString(val);
35          }
36          return r + "}";
37       }
38
39       String r = cl.getName();
40       // inspect the fields of this class and all superclasses
41       do
42       {
43          r += "[";
44          Field[] fields = cl.getDeclaredFields();
45          AccessibleObject.setAccessible(fields, true);
46          // get the names and values of all fields
47          for (Field f : fields)
48          {
49             if (!Modifier.isStatic(f.getModifiers()))
50             {
51                if (!r.endsWith("[")) r += ",";
52                r += f.getName() + "=";
```

(Continues)

Listing 5.16 *(Continued)*

```
53                Class t = f.getType();
54                Object val = f.get(obj);
55                if (t.isPrimitive()) r += val;
56                else r += toString(val);
57             }
58          }
59          r += "]";
60          cl = cl.getSuperclass();
61       }
62       while (cl != null);
63
64       return r;
65    }
66 }
```

java.lang.reflect.AccessibleObject 1.2

- `void setAccessible(boolean flag)`

  sets or clears the accessibility flag for this accessible object, or throws an `IllegalAccessException` if the access is denied.

- `void setAccessible(boolean flag)`
- `boolean trySetAccessible()` 9

  sets the accessibility flag for this accessible object, or returns `false` if the access is denied.

- `boolean isAccessible()`

  gets the value of the accessibility flag for this accessible object.

- `static void setAccessible(AccessibleObject[] array, boolean flag)`

  is a convenience method to set the accessibility flag for an array of objects.

java.lang.Class 1.1

- `Field getField(String name)`
- `Field[] getFields()`

  gets the public field with the given name, or an array of all fields.

- `Field getDeclaredField(String name)`
- `Field[] getDeclaredFields()`

  gets the field that is declared in this class with the given name, or an array of all fields.

java.lang.reflect.Field 1.1

- `Object get(Object obj)`

  gets the value of the field described by this `Field` object in the object `obj`.

- `void set(Object obj, Object newValue)`

  sets the field described by this `Field` object in the object `obj` to a new value.

5.7.6 Using Reflection to Write Generic Array Code

The `Array` class in the `java.lang.reflect` package allows you to create arrays dynamically. This is used, for example, in the implementation of the `copyOf` method in the `Arrays` class. Recall how this method can be used to grow an array that has become full.

```
var a = new Employee[100];
. . .
// array is full
a = Arrays.copyOf(a, 2 * a.length);
```

How can one write such a generic method? It helps that an `Employee[]` array can be converted to an `Object[]` array. That sounds promising. Here is a first attempt:

```
public static Object[] badCopyOf(Object[] a, int newLength) // not useful
{
   var newArray = new Object[newLength];
   System.arraycopy(a, 0, newArray, 0, Math.min(a.length, newLength));
   return newArray;
}
```

However, there is a problem with actually *using* the resulting array. The type of array that this code returns is an array of *objects* (`Object[]`) because we created the array using the line of code

```
new Object[newLength]
```

An array of objects *cannot* be cast to an array of employees (`Employee[]`). The virtual machine would generate a `ClassCastException` at runtime. The point is that, as we mentioned earlier, a Java array remembers the type of its entries—that is, the element type used in the `new` expression that created it. It is legal to cast an `Employee[]` temporarily to an `Object[]` array and then cast it back, but an array that started its life as an `Object[]` array can never be cast into an `Employee[]` array. To write this kind of generic array code, we need to be able to make a new array of the *same* type as the original array. For this, we need the methods of the `Array` class in the `java.lang.reflect` package. The key is the static `newInstance` method of the `Array` class that constructs a new array. You

must supply the type for the entries and the desired length as parameters to this method.

```
Object newArray = Array.newInstance(componentType, newLength);
```

To actually carry this out, we need to get the length and the component type of the new array.

We obtain the length by calling `Array.getLength(a)`. The static `getLength` method of the `Array` class returns the length of an array. To get the component type of the new array:

1. First, get the class object of `a`.
2. Confirm that it is indeed an array.
3. Use the `getComponentType` method of the `Class` class (which is defined only for class objects that represent arrays) to find the right type for the array.

Why is `getLength` a method of `Array` but `getComponentType` a method of `Class`? We don't know—the distribution of the reflection methods seems a bit ad hoc at times.

Here's the code:

```
public static Object goodCopyOf(Object a, int newLength)
{
   Class cl = a.getClass();
   if (!cl.isArray()) return null;
   Class componentType = cl.getComponentType();
   int length = Array.getLength(a);
   Object newArray = Array.newInstance(componentType, newLength);
   System.arraycopy(a, 0, newArray, 0, Math.min(length, newLength));
   return newArray;
}
```

Note that this `copyOf` method can be used to grow arrays of any type, not just arrays of objects.

```
int[] a = { 1, 2, 3, 4, 5 };
a = (int[]) goodCopyOf(a, 10);
```

To make this possible, the parameter of `goodCopyOf` is declared to be of type `Object`, *not an array of objects* (`Object[]`). The integer array type `int[]` can be converted to an `Object`, but not to an array of objects!

Listing 5.17 shows both methods in action. Note that the cast of the return value of `badcopyOf` will throw an exception.