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.

5.7.5 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.

```
Employee[] 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
{
   Object[] 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 5.16 shows both methods in action. Note that the cast of the return value of badcopyOf will throw an exception.

Listing 5.16 arrays/CopyOfTest.java

```
package arrays;

import java.lang.reflect.*;
import java.util.*;

/**
 * This program demonstrates the use of reflection for manipulating arrays.
 * @version 1.2 2012-05-04
 * @author Cay Horstmann
 */
public class CopyOfTest
{
   public static void main(String[] args)
   {
      int[] a = { 1, 2, 3 };
      a = (int[]) goodCopyOf(a, 10);
      System.out.println(Arrays.toString(a));

      String[] b = { "Tom", "Dick", "Harry" };
      b = (String[]) goodCopyOf(b, 10);
      System.out.println(Arrays.toString(b));

      System.out.println("The following call will generate an exception.");
      b = (String[]) badCopyOf(b, 10);
   }

   /**
    * This method attempts to grow an array by allocating a new array and copying all elements.
    * @param a the array to grow
    * @param newLength the new length
    * @return a larger array that contains all elements of a. However, the returned array has
    * type Object[], not the same type as a
    */
   public static Object[] badCopyOf(Object[] a, int newLength) // not useful
   {
      Object[] newArray = new Object[newLength];
      System.arraycopy(a, 0, newArray, 0, Math.min(a.length, newLength));
      return newArray;
   }

   /**
    * This method grows an array by allocating a new array of the same type and
    * copying all elements.
    * @param a the array to grow. This can be an object array or a primitive
    * type array
    * @return a larger array that contains all elements of a.
    */
   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;
   }
}
```

java.lang.reflect.Array 1.1

- `static Object get(Object array, int index)`
- `static` *xxx* `get`*Xxx*`(Object array, int index)`

  (*xxx* is one of the primitive types `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, or `short`.) These methods return the value of the given array that is stored at the given index.

- `static void set(Object array, int index, Object newValue)`
- `static set`*Xxx*`(Object array, int index,` *xxx* `newValue)`

  (*xxx* is one of the primitive types `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, or `short`.) These methods store a new value into the given array at the given index.

- `static int getLength(Object array)`

  returns the length of the given array.

- `static Object newInstance(Class componentType, int length)`
- `static Object newInstance(Class componentType, int[] lengths)`

  returns a new array of the given component type with the given dimensions.

5.7.6 Invoking Arbitrary Methods

In C and C++, you can execute an arbitrary function through a function pointer. On the surface, Java does not have method pointers—that is, ways of giving the location of a method to another method, so that the second method can invoke it later. In fact, the designers of Java have said that method pointers are dangerous and error-prone, and that Java *interfaces* (discussed in the next chapter) are a superior solution. However, the reflection mechanism allows you to call arbitrary methods.