Software Development (cs2500)

Lecture 13: Arrays

M.R.C. van Dongen

November 1, 2010

Contents

1 Introduction

This lecture partly corresponds to the last part of Chapter 4. The main objectives are as follows.

- Study primitive and object arrays.
- Learn techniques for iterating over the members of an array.

2 Arrays with Objects

We've already studied the creation of arrays that contain primitive and object types. Today we continue by studying object arrays in a bit more detail.

Basically, arrays are like trays with cups. Each cup contains a value of the type which was used to create the array. The number of cups is given by the array's `length` *attribute*. Notice, that this attribute violates encapsulation. Let `array` be an array variable. The notation `array[` i `]` gives you the cup at position i in `array`.

Cups behave just like variables. The following two examples demonstrate two important cases. The first case is where a value is assigned to a cup, the second case is where a value is needed from a cup.

- ‘`array[ `*i*` ] = `⟨`expr`⟩’ assigns the value of the expression ⟨`expr`⟩ to the *i*th cup. Basically, this fills the *i*th cup with the value of ⟨`expr`⟩.
- ‘`System.out.println( array[ `*i*` ] )`’ prints what’s in the *i*th cup. The point of this example is that here we’re not using the cup at the left hand side of an assignment, but in an expression where a value is needed. If you use a variable where a value is expected then you get the current value of the variable. With array subscripts this is similar: using `array[ i ]` results in the current value of the *i*th cup in `array`.

It is recalled that the binary operator ‘==’ is for equality testing, so ‘⟨`lhs`⟩ == ⟨`rhs`⟩’ gives `true` if and only the values of ⟨`lhs`⟩ and ⟨`rhs`⟩ are equal. Note that we’re talking about *values*, so ‘⟨`lhs`⟩ == ⟨`rhs`⟩’ is also `true` if ⟨`lhs`⟩ and ⟨`rhs`⟩ are different variables that have the same values. The binary operator ‘`!=`’ is for testing disequality: ‘⟨`lhs`⟩ `==` ⟨`rhs`⟩’ is equivalent to ‘`!(`⟨`lhs`⟩ `!=` ⟨`rhs`⟩`)`’.

It is also recalled that when numeric arrays are created (using `new`) their cups are filled with the value `0`. The following demonstrates the basic array usage.

```Java
int[] ints = new int[ 3 ]; // Magic constant.

// ints[ 0 ] == 0;
ints[ 1 ] = 1;
ints[ 2 ] = 2;

if (ints[ 0 ] != ints[ 1 ]) {
    System.out.println( ints[ 0 ] + " != " + ints[ 1 ] );
}

ints[ 0 ] = ints[ 2 ];
if (ints[ 0 ] == ints[ 2 ]) {
    System.out.println( ints[ 0 ] + " == " + ints[ 2 ] );
}

ints[ 1 ] = ints[ 2 ];
if (ints[ 0 ] == ints[ 1 ]) {
    System.out.println( ints[ 0 ] + " == " + ints[ 1 ] );
}
```

This example prints three lines.[1]

Object arrays with objects work just like primitive value arrays. This time, however, the cups contain object reference values which are given by (1) references to existing objects and (2) the special object reference value `null`, which can be assigned to any object reference variable and which does not correspond to *any* object. Initially, the cups are filled with `null`.

[1] If you don’t understand this, try and see if it makes sense if you leave out the array declaration and use a variable `int0` instead of `ints[ 0 ]`, a variable `int1` instead of `ints[ 1 ]`, and a variable `int2` instead of `ints[ 2 ]`. Next assign `0` to `int0`, `1` to `int1` and `2` to `int2`. Since (different) cups in arrays just behave like (different) variables, the example should now make sense.

The following is an example with object arrays.

```Java
Dog[] dogs = new Dog[ 3 ]; // Magic constant.

// dogs[ 0 ] = null;
dogs[ 1 ] = new Dog( "Zeus" );
dogs[ 2 ] = new Dog( "Bo" );
if (dogs[ 0 ] != dogs[ 1 ]) {
    System.out.println( dogs[ 0 ] + " != " + dogs[ 1 ] );
}

dogs[ 0 ] = dogs[ 2 ];
if (dogs[ 0 ] == dogs[ 2 ]) {
    System.out.println( dogs[ 0 ] + " == " + dogs[ 2 ] );
}

dogs[ 1 ] = dogs[ 2 ];
if (dogs[ 0 ] == dogs[ 1 ]) {
    System.out.println( dogs[ 0 ] + " == " + dogs[ 1 ] );
}
```

This example also prints three lines. If you don't understand this, again try and see if the example makes sense if you use variables instead of array subscript notation. *Hint: Except for the kinds of values that are used in this example, this example is identical to the previous example. For example, all values in* `dogs` *are different after the assignment to* `dogs[ 2 ]` *and the last 13 lines of the example are identical to those of the previous example up to renaming.*

3 Arrays and Iterators

This section studies basic techniques for iterating over arrays. There are two techniques for iterating over the members of an array.

Index-based iteration: The first technique is the well-known idiom that uses an index variable to enumerate the possible indices of the array.

Collection-based notation: This technique depends on a special recently introduced notation which avoids the use of index variables. The notation only works for arrays, *collections*, and `Iterable` objects. (For the moment it suffices to know that a collection is an object that supports storing and retrieval of other objects. Examples are `Lists`, `Sets`, `Queues`, and so on. `Iterable` objects may also be viewed as collections that store objects (but there are differences). A class is `Iterable` if it implements a method called `iterator( )` which returns an `Iterator` object which can be used to iterate over the members of the `Iterable` object. Here iterating over the members is done in a similar way as with the `hasNext-next` mechanism for `Scanner` objects which we studied in the previous lecture. We shall study collections and `Iterable` classes in future lectures.)

3.1 Index-based Iteration

The following uses the index-based notation to enumerate the members of the array from "left to right".

```Java
int index;
for ( index = 0; index < array.length; index ++ )
    ⟨Use array[ index ]⟩
```

You can also use the test '`index != array.length`' instead of '`index < array.length`'.

The following demonstrates why the basic index-based iteration idiom is prone to certain kinds of programming errors.

```Don't Try this at Home
int Index;
for ( Index = 0; Index < array.length; Index ++ )
    ⟨Use array[ Index ]⟩
⋮

int Index;
for ( Index = 0; Index < array.length; Index ++ )
    ⟨Use array[ Index ]⟩ // Oops.
```

The main reason for this error is that the second loop is in the scope of the variable `Index`. Stated differently, the scope of the variable `Index` is too larger: it should have been been "confined" to the `for` loop.

It is recalled that the scope of a local variable is restricted to the innermost block that encloses the variable's declaration. The following example demonstrates this point: the variable `index` cannot be used outside the block.

```Java
{
    int index;
    for ( index = 0; index < array.length; index ++ )
        ⟨Use array[ index ]⟩
}
```

Adding extra block just to restrict the scope of a variable isn't great because you get deeper block-nesting levels, which makes it difficult to see what's going on. The following idiom also doesn't expose the index variable outside the `for` construct. This is the preferred idiom if you don't need the index variable outside the `for` loop.

```Java
for ( int index = 0; index < array.length; index ++ )
    ⟨Use array[ index ]⟩
```

Despite it being an improvement to the basic index-based iteration idiom, the index-based notation with local variables still isn't (always) perfect. The following demonstrates why.

```
int Index;                                                    Don't Try this at Home
for ( int Index = 0; Index < array.length; Index ++ )
    ⟨Use array[ Index ]⟩
```

Another common error occurring in combination with array iteration are *off-by-one errors*. These errors are caused by errors in the termination condition as a result of which the iteration omits an index or treats an index too many. The following is an example.

```
for ( int index = 0; index <= array.length; index ++ )        Don't Try this at Home
    ⟨Use array[ index ]⟩
```

The last kind of errors which may occur in combination with the index-based iterator notation is caused by overflow. The following is a prototypical example. The idiom works for most arrays but it fails if `array.length == Integer.MAX_SIZE`: the condition `index <= array.length` cannot fail.

```
int[] array = new int[ Integer.MAX_LENGTH ];                  Don't Try this at Home
for ( int index = 1; index <= array.length; index ++ )
    ⟨Use array[ index - 1 ]⟩
```

3.2 Enhanced `for` Loop

If the iteration is over an array, over a so-called *collection* or an `Iterable` object then there is an alternative to the index-based idiom. The alternative notation is called the *enhanced `for` loop notation*. The following demonstrates how you use it.

```
int[] things = {1,2,3};                                                    Java
for ( int thing : things )
    ⟨Use thing⟩
```

Arguably this idiom it is easier to read and understand. More importantly, it completely avoids the use of the index variable, thereby avoiding possible off-by-one errors and unintentional changes to the value of the index variable inside the loop. In addition it avoids off-by-one errors and overflow errors. Clearly, this is the preferred idiom for object arrays and left-to-right iteration.

4 For Wednesday

Study the notes, study Pages 80–87 of the book, and carry out the exercises on Pages 92 and 93 of the Book.