CHAPTER 10

Arrays

Even Solomon in all his glory was not arrayed like one of these.
—Matthew 6:29

IN the Java programming language *arrays* are objects (§4.3.1), are dynamically created, and may be assigned to variables of type `Object` (§4.3.2). All methods of class `Object` may be invoked on an array.

An array object contains a number of variables. The number of variables may be zero, in which case the array is said to be *empty*. The variables contained in an array have no names; instead they are referenced by array access expressions that use nonnegative integer index values. These variables are called the *components* of the array. If an array has *n* components, we say *n* is the *length* of the array; the components of the array are referenced using integer indices from 0 to $n - 1$, inclusive.

All the components of an array have the same type, called the *component type* of the array. If the component type of an array is *T*, then the type of the array itself is written *T*`[]`.

The value of an array component of type `float` is always an element of the float value set (§4.2.3); similarly, the value of an array component of type `double` is always an element of the double value set. It is not permitted for the value of an array component of type `float` to be an element of the float-extended-exponent value set that is not also an element of the float value set, nor for the value of an array component of type `double` to be an element of the double-extended-exponent value set that is not also an element of the double value set.

The component type of an array may itself be an array type. The components of such an array may contain references to subarrays. If, starting from any array type, one considers its component type, and then (if that is also an array type) the component type of that type, and so on, eventually one must reach a component type that is not an array type; this is called the *element type* of the original array, and the components at this level of the data structure are called the *elements* of the original array.

There are some situations in which an element of an array can be an array: if the element type is `Object` or `Cloneable` or `java.io.Serializable`, then some or all of the elements may be arrays, because any array object can be assigned to any variable of these types.

10.1 Array Types

An array type is written as the name of an element type followed by some number of empty pairs of square brackets `[]`. The number of bracket pairs indicates the depth of array nesting. An array's length is not part of its type.

The element type of an array may be any type, whether primitive or reference. In particular:

- Arrays with an interface type as the component type are allowed. The elements of such an array may have as their value a null reference or instances of any type that implements the interface.
- Arrays with an `abstract` class type as the component type are allowed. The elements of such an array may have as their value a null reference or instances of any subclass of the `abstract` class that is not itself `abstract`.

Array types are used in declarations and in cast expressions (§15.16).

10.2 Array Variables

A variable of array type holds a reference to an object. Declaring a variable of array type does not create an array object or allocate any space for array components. It creates only the variable itself, which can contain a reference to an array. However, the initializer part of a declarator (§8.3) may create an array, a reference to which then becomes the initial value of the variable.

Because an array's length is not part of its type, a single variable of array type may contain references to arrays of different lengths.

Here are examples of declarations of array variables that do not create arrays:

```
int[] ai;           // array of int
short[][] as;       // array of array of short
Object[]  ao,       // array of Object
          otherAo;  // array of Object
short  s,           // scalar short
      aas[][];          // array of array of short
```

Here are some examples of declarations of array variables that create array objects:

```
Exception ae[] = new Exception[3];
Object aao[][] = new Exception[2][3];
int[] factorial = { 1, 1, 2, 6, 24, 120, 720, 5040 };
char ac[] = { 'n', 'o', 't', ' ', 'a', ' ',
              'S', 't', 'r', 'i', 'n', 'g' };
String[] aas = { "array", "of", "String", };
```

The `[]` may appear as part of the type at the beginning of the declaration, or as part of the declarator for a particular variable, or both, as in this example:

```
byte[] rowvector, colvector, matrix[];
```

This declaration is equivalent to:

```
byte rowvector[], colvector[], matrix[][];
```

Once an array object is created, its length never changes. To make an array variable refer to an array of different length, a reference to a different array must be assigned to the variable.

If an array variable *`v`* has type *`A`*`[]`, where *`A`* is a reference type, then *`v`* can hold a reference to an instance of any array type *`B`*`[]`, provided *`B`* can be assigned to *`A`*. This may result in a run-time exception on a later assignment; see §10.10 for a discussion.

10.3 Array Creation

An array is created by an array creation expression (§15.10) or an array initializer (§10.6).

An array creation expression specifies the element type, the number of levels of nested arrays, and the length of the array for at least one of the levels of nesting. The array's length is available as a final instance variable `length`.

An array initializer creates an array and provides initial values for all its components.

10.4 Array Access

A component of an array is accessed by an array access expression (§15.13) that consists of an expression whose value is an array reference followed by an indexing expression enclosed by `[` and `]`, as in `A[i]`. All arrays are `0`-origin. An array with length *`n`* can be indexed by the integers `0` to *`n`*`-1`.

Arrays must be indexed by `int` values; `short`, `byte`, or `char` values may also be used as index values because they are subjected to unary numeric promotion

(§5.6.1) and become `int` values. An attempt to access an array component with a `long` index value results in a compile-time error.

All array accesses are checked at run time; an attempt to use an index that is less than zero or greater than or equal to the length of the array causes an `ArrayIndexOutOfBoundsException` to be thrown.

10.5 Arrays: A Simple Example

The example:

```
class Gauss {
    public static void main(String[] args) {
        int[] ia = new int[101];
        for (int i = 0; i < ia.length; i++)
            ia[i] = i;
        int sum = 0;
        for (int i = 0; i < ia.length; i++)
            sum += ia[i];
        System.out.println(sum);
    }
}
```

that produces the output:

```
5050
```

declares a variable `ia` that has type array of `int`, that is, `int[]`. The variable `ia` is initialized to reference a newly created array object, created by an array creation expression (§15.10). The array creation expression specifies that the array should have `101` components. The length of the array is available using the field `length`, as shown.

The example program fills the array with the integers from `0` to `100`, sums these integers, and prints the result.

10.6 Array Initializers

An *array initializer* may be specified in a declaration, or as part of an array creation expression (§15.10), creating an array and providing some initial values:

ArrayInitializer:
  `{` $\mathit{VariableInitializers}_{opt}$ $,_{opt}$ `}`

VariableInitializers:
  VariableInitializer
  VariableInitializers , *VariableInitializer*

The following is repeated from §8.3 to make the presentation here clearer:

VariableInitializer:
  Expression
  ArrayInitializer

An array initializer is written as a comma-separated list of expressions, enclosed by braces "{" and "}".

The length of the constructed array will equal the number of expressions.

The expressions in an array initializer are executed from left to right in the textual order they occur in the source code. The *n*th variable initializer specifies the value of the *n-1*st array component. Each expression must be assignment-compatible (§5.2) with the array's component type, or a compile-time error results. It is a compile-time error if the component type of the array being initialized is not reifiable (§4.7).

If the component type is itself an array type, then the expression specifying a component may itself be an array initializer; that is, array initializers may be nested.

A trailing comma may appear after the last expression in an array initializer and is ignored.

As an example:

```
class Test {
    public static void main(String[] args) {
        int ia[][] = { {1, 2}, null };
        for (int i = 0; i < 2; i++)
            for (int j = 0; j < 2; j++)
              System.out.println(ia[i][j]);
    }
}
```

prints:

```
1
2
```

before causing a `NullPointerException` in trying to index the second component of the array `ia`, which is a null reference.

10.7 Array Members

The members of an array type are all of the following:

- The `public final` field `length`, which contains the number of components of the array (`length` may be positive or zero)
- The `public` method `clone`, which overrides the method of the same name in class `Object` and throws no checked exceptions. The return type of the clone method of an array type *T[]* is *T[]*
- All the members inherited from class `Object`; the only method of `Object` that is not inherited is its `clone` method

An array thus has the same public fields and methods as the following class:

```
class A implements Cloneable, java.io.Serializable {
    public final int length = X;
    public Object clone() {
        try {
            return super.clone();
        } catch (CloneNotSupportedException e) {
            throw new InternalError(e.getMessage());
        }
    }
}
```

Every array implements the interfaces `Cloneable` and `java.io.Serializable`.

That arrays are cloneable is shown by the test program:

```
class Test {
    public static void main(String[] args) {
        int ia1[] = { 1, 2 };
        int ia2[] = (int[])ia1.clone();
        System.out.print((ia1 == ia2) + " ");
        ia1[1]++;
        System.out.println(ia2[1]);
    }
}
```

which prints:

```
false 2
```

showing that the components of the arrays referenced by `ia1` and `ia2` are different variables. (In some early implementations of the Java programming language this example failed to compile because the compiler incorrectly believed that the clone method for an array could throw a `CloneNotSupportedException`.)

A `clone` of a multidimensional array is shallow, which is to say that it creates only a single new array. Subarrays are shared.

This is shown by the example program:

```
class Test {
        public static void main(String[] args) throws Throwable {
            int ia[][] = { { 1 , 2}, null };
            int ja[][] = (int[][])ia.clone();
            System.out.print((ia == ja) + " ");
            System.out.println(ia[0] == ja[0] && ia[1] == ja[1]);
        }
    }
```

which prints:

```
    false true
```

showing that the `int[]` array that is `ia[0]` and the `int[]` array that is `ja[0]` are the same array.

10.8 `Class` Objects for Arrays

Every array has an associated `Class` object, shared with all other arrays with the same component type. The direct superclass of an array type is `Object`. Every array type implements the interfaces `Cloneable` and `java.io.Serializable`.

This is shown by the following example code:

```
    class Test {
        public static void main(String[] args) {
            int[] ia = new int[3];
            System.out.println(ia.getClass());
            System.out.println(ia.getClass().getSuperclass());
        }
    }
```

which prints:

```
    class [I
    class java.lang.Object
```

where the string “`[I`” is the run-time type signature for the class object “array with component type `int`”.

10.9 An Array of Characters is Not a `String`

In the Java programming language, unlike C, an array of `char` is not a `String`, and neither a `String` nor an array of `char` is terminated by `'\u0000'` (the NUL character).

A `String` object is immutable, that is, its contents never change, while an array of `char` has mutable elements. The method `toCharArray` in class `String` returns an array of characters containing the same character sequence as a

String. The class StringBuffer implements useful methods on mutable arrays of characters.

10.10 Array Store Exception

If an array variable *v* has type *A*[], where *A* is a reference type, then *v* can hold a reference to an instance of any array type *B*[], provided *B* can be assigned to *A*.

Thus, the example:

```
class Point { int x, y; }

class ColoredPoint extends Point { int color; }

class Test {
    public static void main(String[] args) {
        ColoredPoint[] cpa = new ColoredPoint[10];
        Point[] pa = cpa;
        System.out.println(pa[1] == null);
        try {
            pa[0] = new Point();
        } catch (ArrayStoreException e) {
            System.out.println(e);
        }
    }
}
```

produces the output:

```
true
java.lang.ArrayStoreException
```

Here the variable pa has type Point[] and the variable cpa has as its value a reference to an object of type ColoredPoint[]. A ColoredPoint can be assigned to a Point; therefore, the value of cpa can be assigned to pa.

A reference to this array pa, for example, testing whether pa[1] is null, will not result in a run-time type error. This is because the element of the array of type ColoredPoint[] is a ColoredPoint, and every ColoredPoint can stand in for a Point, since Point is the superclass of ColoredPoint.

On the other hand, an assignment to the array pa can result in a run-time error. At compile time, an assignment to an element of pa is checked to make sure that the value assigned is a Point. But since pa holds a reference to an array of ColoredPoint, the assignment is valid only if the type of the value assigned at run-time is, more specifically, a ColoredPoint.

The Java virtual machine checks for such a situation at run-time to ensure that the assignment is valid; if not, an ArrayStoreException is thrown. More formally: an assignment to an element of an array whose type is *A*[], where *A* is a reference type, is checked at run-time to ensure that the value assigned can be assigned to the actual element type of the array, where the actual element type may be any reference type that is assignable to `A`.

At length burst in the argent revelry,
With plume, tiara, and all rich array . . .
—John Keats, *The Eve of St. Agnes* (1819)