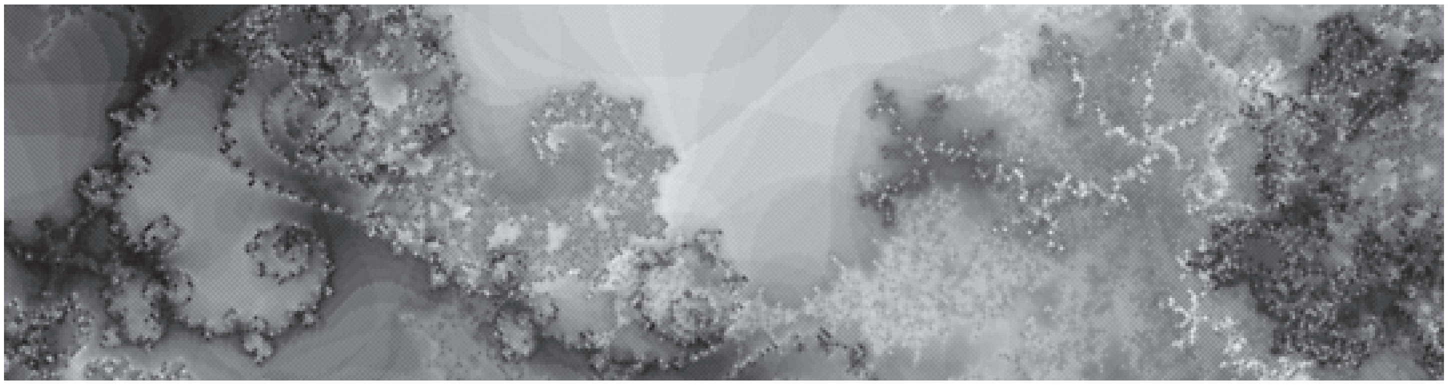

Part One

The Historical Collection Classes

CHAPTER 2

Arrays

Arrays are the only collection support defined within the Java programming language. They are objects that store a set of elements in an order accessible by *index,* or position. They are a subclass of `Object` and implement both the `Serializable` and `Cloneable` interfaces. However, there is no `.java` source file for you to see how the internals work. Basically, you create an array with a specific size and type of element, then fill it up.

NOTE *Since arrays subclass* `Object`*, you can synchronize on an array variable and call its* `wait()` *and* `notify()` *methods.*

Let's take a look at what we can do with array objects—beginning with basic usage and declaration and moving through to copying and cloning. We'll also look at array assignment, equality checking, and reflection.

Array Basics

Before going into the details of declaring, creating, initializing, and copying arrays, let's go over a simple array example. When creating a Java application, the `main()` method has a single argument that is a `String` array: `public static void main(String args[])`. The compiler doesn't care what argument name you use, only that it is an array of `String` objects.

Given that we now have the command-line arguments to our application as an array of `String` objects, we can look at each element and print it. Within Java, arrays know their size, and they are always indexed from position zero. Therefore, we can ask the array how big it is by looking at the sole instance variable for an array: `length`. The following code shows how to do this:

```
public class ArrayArgs {
  public static void main (String args[]) {
    for (int i=0, n=args.length; i<n; i++) {
      System.out.println("Arg " + i + ": " + args[i]);
    }
  }
}
```

NOTE *Array indices cannot be of type* `long`*. Because only non-negative integers can be used as indices, this effectively limits the number of elements in an array to 2,147,483,648, or 2^{31}, with a range of indices from 0 to 2^{31}–1.*

Because an array's size doesn't change as we walk through the loop, there is no need to look up the length for each test case, as in: `for (int i=0; i<args.length; i++)`. In fact, to go through the loop counting down instead of up as a check for zero test case is nominally faster in most instances: `for (int i=args.length-1; i>=0; i–)`. While the JDK 1.1 and 1.2 releases have relatively minor performance differences when counting down versus counting up, these timing differences are more significant with the 1.3 release. To demonstrate the speed difference on your platform, try out the program in Listing 2-1 to time how long it takes to loop "max int" times:

Listing 2-1. Timing loop performance.

```
public class TimeArray {
  public static void main (String args[]) {
    int something = 2;
    long startTime = System.currentTimeMillis();
    for (int i=0, n=Integer.MAX_VALUE; i<n; i++) {
      something = -something;
    }
    long midTime = System.currentTimeMillis();
    for (int i=Integer.MAX_VALUE-1; i>=0; i–) {
      something = -something;
    }
    long endTime = System.currentTimeMillis();
    System.out.println("Increasing Delta: " + (midTime - startTime));
    System.out.println("Decreasing Delta: " + (endTime - midTime));
  }
}
```

This test program is really timing the for-loop and not the array access because there is no array access.

NOTE *In most cases, the numbers calculated on my 400 MHz Windows NT system were in the low 11,000s for JDK 1.1 and 1.2. However, under JDK 1.3 with the* `-classic` *option (no JIT), the timing numbers increased to around 250,000. Even using the HotSpot VM with 1.3, the numbers were between 19,000 and 30,000.*

If you ever try to access before the beginning or after the end of an array, an `ArrayIndexOutOfBoundsException` will be thrown. As a subclass of `IndexOutOfBoundsException`, the `ArrayIndexOutOfBoundsException` is a runtime exception, as shown in Figure 2-1. Thankfully, this means that you do not have to place array accesses within `try-catch` blocks. In addition, since looking beyond the bounds of an array is a runtime exception, your program will compile just fine. The program will only throw the exception when the access is attempted.

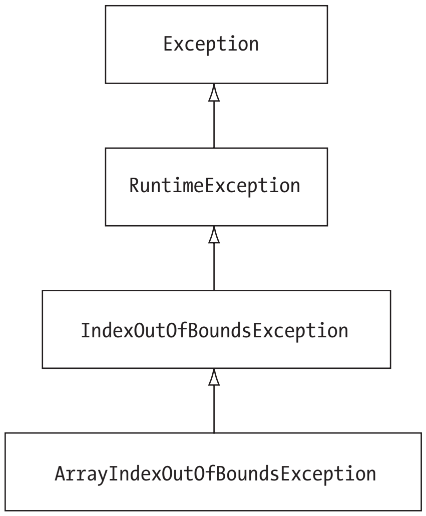

Figure 2-1. The class hierarchy of `ArrayIndexOutOfBoundsException`.

NOTE *You cannot turn off array bounds checking. It is part of the security architecture of the Java runtime to ensure that invalid memory space is never accessed.*

The following code demonstrates an improper way to read through the command-line array elements:

```
public class ArrayArgs2 {
  public static void main (String args[]) {
    try {
      int i=0;
      do {
        System.out.println("Arg " + i + ": " + args[i++]);
      } while (true);
    } catch (ArrayIndexOutOfBoundsException ignored) {
    }
  }
}
```

While functionally equivalent to the earlier `ArrayArgs` example, it is bad programming practice to use exception handling for control flow. Exception handling should be reserved for exceptional conditions.

Declaring and Creating Arrays

Remember that arrays are objects that store a set of elements in an index-accessible order. Those elements can either be a primitive datatype, such as an `int` or `float`, or any type of `Object`. To declare an array of a particular type, just add brackets (`[ ]`) to the declaration:

```
int[] variable;
```

For array declaration, the brackets can be in one of three places: `int[] variable`, `int []variable`, and `int variable[]`. The first says that the `variable` is of type `int[]`. The latter two say that the `variable` is an array and that the array is of type `int`.

> **NOTE** *This might sound like we're arguing semantics. However, there is a difference when you declare multiple variables depending upon which form you use. The form* `int[] var1, var2;` *will declare two variables that are* `int` *arrays, whereas* `int []var1, var2;` *or* `int var1[], var2;` *will declare one* `int` *array and another just of type* `int`.

Once you've declared the array, you can create the array and save a reference to it. The `new` operator is used to create arrays. When you create an array, you must specify its length. Once this length is set, you cannot change it. As the following demonstrates, the length can be specified as either a constant or an expression:

```
int variable[] = new int[10];
```

or

```
int[] createArray(int size) {
  return new int[size];
}
```

NOTE *If you try to create an array where the length is negative, the run-time* `NegativeArraySizeException` *will be thrown. Zero-length arrays, however, are valid.*

You can combine array declaration and creation into one step:

```
int variable[] = new int[10];
```

WARNING *In the event that the creation of an array results in an* `OutOfMemoryError` *being thrown, all dimension expressions will already have been evaluated. This is important if an assignment is performed where the dimension is specified. For example, if the expression* `int variable[] = new int[var1 = var2*var2]` *were to cause an* `OutOfMemoryError` *to be thrown, the variable* `var1` *will be set prior to the error being thrown.*

Once an array has been created, you can fill it up. This is normally done with a for-loop or with separate assignment statements. For instance, the following will create and fill a three-element array of names:

```
String names = new String[3];
names[0] = "Leonardo";
names[1] = "da";
names[2] = "Vinci";
```

Arrays of Primitives

When you create an array of primitive elements, the array holds the actual values for those elements. For instance, Figure 2-2 shows what an array of six integers (1452, 1472, 1483, 1495, 1503, 1519) referred to from the variable `life` would look like with regards to stack and heap memory.

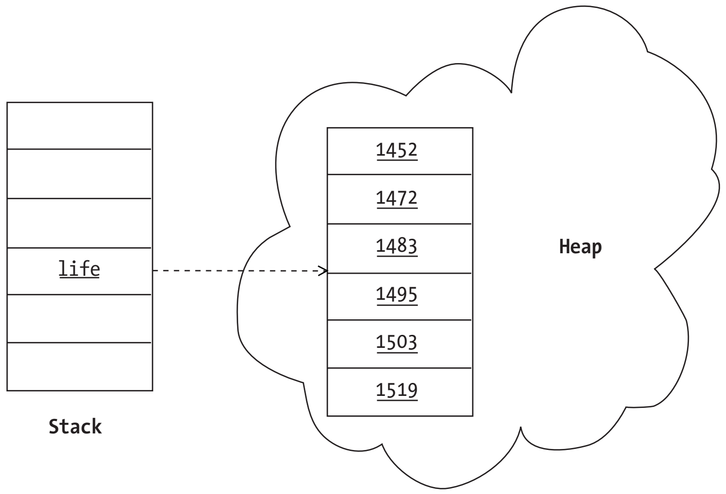

Figure 2-2. The stack and heap memory for an array of primitives.

Arrays of Objects

Unlike an array of primitives, when you create an array of objects, they are not stored in the actual array. The array only stores references to the actual objects, and initially each reference is null unless explicitly initialized. (More on initialization shortly.) Figure 2-3 shows what an array of `Object` elements would look like where the elements are as follows:

* Leonardo da Vinci's country of birth, Italy
* An image of his painting *The Baptism of Christ*
* His theories (drawings) about helicopters and parachutes
* An image of the *Mona Lisa*
* His country of death, France

The key thing to notice in Figure 2-3 is that the objects are not in the array: only references to the objects are in the array.

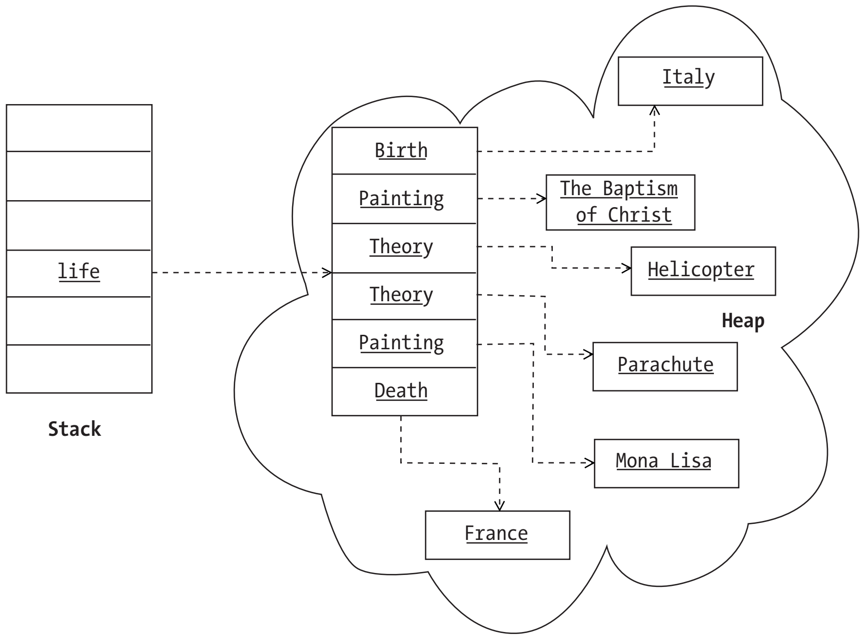

Figure 2-3. The stack and heap memory for an array of objects.

Multidimensional Arrays

Because arrays are handled through references, there is nothing that stops you from having an array element refer to another array. When one array refers to another, you get a *multidimensional array*. This requires an extra set of square brackets for each added dimension on the declaration line. For instance, if you wish to define a rectangular, two-dimensional array, you might use the following line:

```
int coordinates[][];
```

As with one-dimensional arrays, if an array is one of primitives, you can immediately store values in it once you create the array. Just declaring it isn't sufficient. For instance, the following two lines will result in a compilation-time error because the array variable is never initialized:

```
int coordinates[][];
coordinates[0][0] = 2;
```

If, however, you created the array between these two source lines (with something like coordinates = new int[3][4];), the last line would become valid.

In the case of an array of objects, creating the multidimensional array produces an array full of null object references. You still need to create the objects to store in the arrays, too.

Because each element in the outermost array of a multidimensional array is an object reference, there is nothing that requires your arrays to be rectangular (or cubic for three-dimensional arrays). Each inner array can have its own size. For instance, the following demonstrates how to create a two-dimensional array of floats where the inner arrays are sized like a set of bowling pins—the first row has one element, the second has two, the third has three, and the fourth has four:

```
float bowling[][] = new float[4][];
for (int i=0; i<4; i++) {
  bowling[i] = new float[i+1];
}
```

To help visualize the final array, see Figure 2-4.

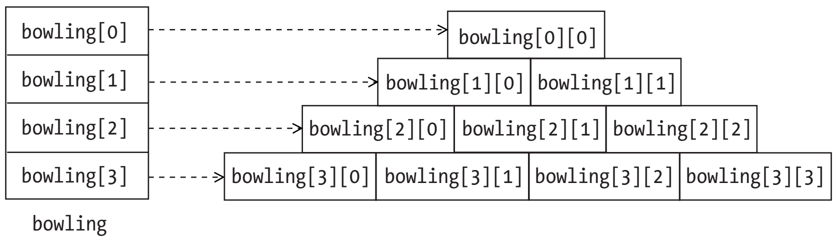

Figure 2-4. A triangular, bowling-pin-like array.

When accessing an array with multiple dimensions, each dimension expression is fully evaluated before the next dimension expression to the right is ever examined. This is important to know if an exception happens during an array access.

NOTE *While you can syntactically place square brackets before or after an array variable when it is for a single dimension (* [index]name *or* name[index]*), you must place the square brackets after the array variable for multiple dimensions, as in* name[index1][index2]. *Syntactically,* [index1][index2]name *and* [index1]name[index2] *are illegal and will result in a compile-time error if found in your code. For declarations, it is perfectly legal to place the brackets before (*type [][]name*), after (*type name[][]*), or around (*type []name[]*) the variable name.*

Keep in mind that computer memory is linear—when you access a multidimensional array you are really accessing a one-dimensional array in memory. If you can access the memory in the order it is stored, the access will be most efficient. Normally, this wouldn't matter if everything fit in memory, as computer memory is quick to hop around. However, when using large data structures, linear access performs best and avoids unnecessary swapping. In addition, you can simulate multidimensional arrays by packing them into a one-dimensional array. This is done frequently with images. The two manners of packing the one-dimensional array are row-major order, where the array is filled one row at a time; and column-major order, where columns are placed into the array. Figure 2-5 shows the difference between the two.

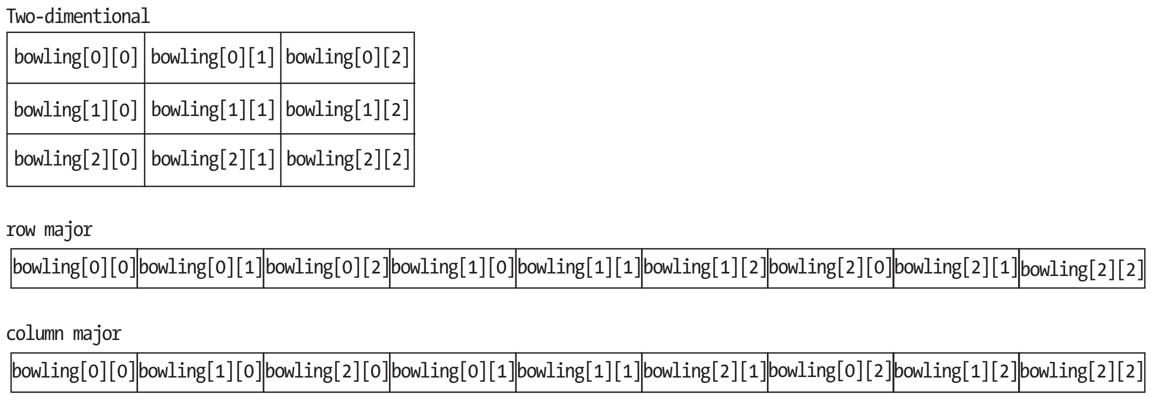
Two-dimentional
bowling[0][0] bowling[0][1] bowling[0][2]
bowling[1][0] bowling[1][1] bowling[1][2]
bowling[2][0] bowling[2][1] bowling[2][2]
row major
bowling[0][0] bowling[0][1] bowling[0][2] bowling[1][0] bowling[1][1] bowling[1][2] bowling[2][0] bowling[2][1] bowling[2][2]
column major
bowling[0][0] bowling[1][0] bowling[2][0] bowling[0][1] bowling[1][1] bowling[2][1] bowling[0][2] bowling[1][2] bowling[2][2]

Figure 2-5. Row-major versus column-major order.

NOTE *In many of the image processing routines, such as* `setPixels()` *of the* `ImageFilter` *class, you'll find two-dimensional image arrays flattened into row-major order, where Pixel (m, n) translates into a one-dimensional position, n * scansize + m. This reads top-down, left-to-right through the image data.*

Initializing Arrays

When an array is first created, the runtime-environment will make sure that the array contents are automatically initialized to some known (as opposed to undefined) value. As with uninitialized instance and class variables, array contents are initialized to either the numerical equivalent of zero, the character equivalent of `\u0000`, the boolean false, or null for object arrays, as shown in Table 2-1.

Table 2-1. Array Initial Values

DEFAULT VALUE	ARRAY
0	byte
	short
	int
	long
0.0	float
	double
\u0000	char
false	boolean
null	Object

When you declare an array you can specify the initial values of the elements. This is done by providing a comma-delimited list between braces [`{ }`] after an equal sign at the declaration point.

For instance, the following will create a three-element array of names:

```
String names[] = {"Leonardo", "da", "Vinci"};
```

Notice that when you provide an array initializer, you do not have to specify the length. The array length is set automatically based upon the number of elements in the comma-delimited list.

> **NOTE** *The Java language syntax permits a trailing comma after the last element in an array initializer block, as in {"Leonardo", "da", "Vinci",}. This does not change the length of the array to four, but keeps it at three. This flexibility is primarily for the benefit of code generators.*

For multidimensional arrays, you would just use an extra set of parenthesis for each added dimension. For instance, the following creates a 6 × 2 array of years and events. Because the array is declared as an array of `Object` elements, it is necessary to use the `Integer` wrapper class to store each `int` primitive value inside. All elements within an array must be of the array's declared type, or a subclass of that type, in this case, `Object`, even though all of the elements are subclasses.

```
Object events[][] = {
    {new Integer(1452), new Birth("Italy")},
    {new Integer(1472), new Painting("baptismOfChrist.jpg")},
    {new Integer(1483), new Theory("Helicopter")},
    {new Integer(1495), new Theory("Parachute")},
    {new Integer(1503), new Painting("monaLisa.jpg")},
    {new Integer(1519), new Death("France")}
};
```

NOTE *In the event the type of the array is an interface, all elements in the array must implement the interface.*

Starting with the second dot-point Java release (Java 1.1), the concept of *anonymous arrays* was introduced. While it was easy to initialize an array when it was declared, you couldn't reinitialize the array later with a comma-delimited list unless you declared another variable to store the new array in. This is where anonymous arrays step in. With an anonymous array, you can reinitialize an array to a new set of values, or pass unnamed arrays into methods when you don't want to define a local variable to store said array.

Anonymous arrays are declared similarly to regular arrays. However, instead of specifying a length within the square brackets, you place a comma-delimited list of values within braces after the brackets, as shown here:

```
new type[] {comma-delimited-list}
```

To demonstrate, the following line shows how to call a method and pass to it an anonymous array of `String` objects:

```
method(new String[] {"Leonardo", "da", "Vinci"});
```

You'll find anonymous arrays used frequently by code generators.

Passing Array Arguments and Return Values

When an array is passed as an argument to a method, a reference to the array is passed. This permits you to modify the contents of the array and have the calling routine see the changes to the array when the method returns. In addition, because a reference is passed around, you can also return arrays created within methods and not worry about the garbage collector releasing the array's memory when the method is done.

Copying and Cloning Arrays

You can do many things when working with arrays. If you've outgrown the initial size of the array, you need to create a new larger one and copy the original elements into the same location of the larger array. If, however, you don't need to make the array larger, but instead you want to modify the array's elements while keeping the original array intact, you must create a copy or clone of the array.

The `arraycopy()` method of the `System` class allows you to copy elements from one array to another. When making this copy, the destination array can be larger; but if the destination is smaller, an `ArrayIndexOutOfBoundsException` will be thrown at runtime. The `arraycopy()` method takes five arguments (two for each array and starting position, and one for the number of elements to copy): `public static void arraycopy (Object sourceArray, int sourceOffset, Object destinationArray, int destinationOffset, int numberOfElementsToCopy)`. Besides type compatibility, the only requirement here is that the destination array's memory is already allocated.

WARNING *When copying elements between different arrays, if the source or destination arguments are not arrays or their types are not compatible, an* `ArrayStoreException` *will be thrown. Incompatible arrays would be where one is an array of primitives and the other is an array of objects; or the primitive types are different; or the object types are not assignable.*

To demonstrate, Listing 2-2 takes an integer array and creates a new array that is twice as large. The `doubleArray()` method in the following example does this for us:

Listing 2-2. Doubling the size of an array.

```
public class DoubleArray {
  public static void main (String args[]) {
    int array1[] = {1, 2, 3, 4, 5};
    int array2[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    System.out.println("Original size: " + array1.length);
    System.out.println("New size: " + doubleArray(array1).length);
    System.out.println("Original size: " + array2.length);
    System.out.println("New size: " + doubleArray(array2).length);
  }
  static int[] doubleArray(int original[]) {
    int length = original.length;
    int newArray[] = new int[length*2];
    System.arraycopy(original, 0, newArray, 0, length);
    return newArray;
  }
}
```

After getting the length of the original array, a new array of the right size is created before the old elements are copied into their original positions in the new array. After you learn about array reflection in a later section, you can generalize the method to double the size of an array of any type.

When executed, the program generates the following output:

```
Original size: 5
New size: 10
Original size: 9
New size: 18
```

NOTE *When copying arrays with* `arraycopy()`*, the source and destination arrays can be the same if you want to copy a subset of the array to another area within that array. This works even if there is some overlap.*

Since arrays implement the `Cloneable` interface, besides copying regions of arrays, you can also clone them. *Cloning* involves creating a new array of the same size and type and copying all the old elements into the new array. This is unlike *copying*, which requires you to create and size the destination array yourself. In the case of primitive elements, the new array has copies of the old elements, so changes to the elements of one are not reflected in the copy. However, in the case of object references, only the reference is copied. Thus, both copies of the array would point to the same object. Changes to that object would be reflected in both arrays. This is called a *shallow copy* or *shallow clone*.

To demonstrate, the following method takes one integer array and returns a copy of said array.

```
static int[] cloneArray(int original[]) {
  return (int[])original.clone();
}
```

Array cloning overrides the protected `Object` method that would normally throw a `CloneNotSupportedException` with a public one that actually works.

Array Immutability

It is useful to return an array clone from a method if you don't want the caller of your method to modify the underlying array structure. While you can declare arrays to be final, as in the following example:

```
final static int array[] = {1, 2, 3, 4, 5};
```

declaring an object reference `final` (specifically, an array reference here) does not restrict you from modifying the object. It only limits you from changing what the final variable refers to. While the following line results in a compilation error:

```
array = new int[] {6, 7, 8, 9};
```

changing an individual element is perfectly legal:

```
array[3] = 6;
```

TIP *Another way to "return" an immutable array from a method is to return an* `Enumeration` *or* `Iterator` *into the array, rather than returning the actual array. Either interface provides access to the individual elements without exposing the whole array to changes or requiring you to make a copy of the entire array. You'll learn more about these interfaces in later chapters.*

Array Assignments

Array assignments work like variable assignments. If variable `x` is a reference to an array of `y`, then `x` can be a reference to `z` if a variable of type `z` can be assigned to `y`. For instance, imagine that `y` is the AWT `Component` class and `z` is the AWT `Button` class. Because a `Button` variable can be assigned to a `Component` variable, a `Button` array can be assigned to a `Component` array:

```
Button buttons[] = {
  new Button ("One"),
  new Button("Two"),
  new Button("Three")};
Component components[] = buttons;
```

When an assignment like this is made, both variables' buttons and components refer to the same heap space in memory, as shown by Figure 2-6. Changing an array element for one array changes the element for both.

Zukow_02 4/2/01 10:10 AM Page 23

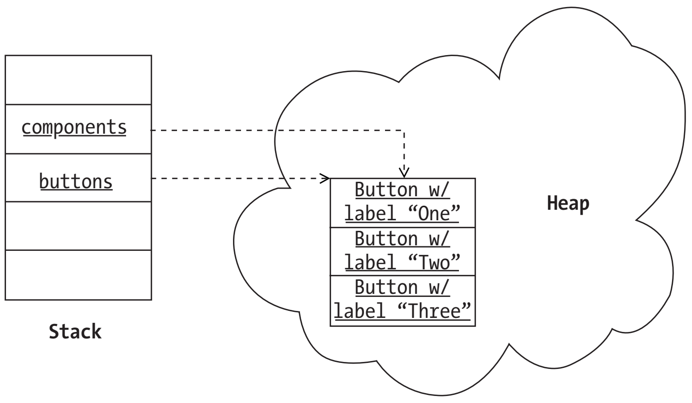

Figure 2-6. Shared memory after an array assignment.

If, after assigning an array variable to a superclass array variable (as in the prior example of assigning the button array to a component array variable) you then try to place a different subclass instance into the array, an `ArrayStoreException` is thrown. To continue the prior example, an `ArrayStoreException` would be thrown if you tried to place a `Canvas` into the `components` array. Even though the `components` array is declared as an array of `Component` objects, because the components array specifically refers to an array of `Button` objects, `Canvas` objects cannot be stored in the array. This is a run-time exception as the actual assignment is legal from the perspective of a type-safe compiler.

Checking for Array Equality

Checking for equality between two arrays can be done in one of two manners depending upon the type of equality you are looking for. Are the array variables pointing to the same place in memory and thus pointing to the same array? Or are the elements of two arrays comparatively equivalent?

Checking for two references to the same memory space is done with the double equal sign operator ==. For example, the prior `components` and `buttons` variables would be equal in this case since one is a reference to the other:

```
components == buttons // true
```

However, if you compare an array to a cloned version of that array then these would not be equal as far as == goes. Since these arrays have the same elements but exist in different memory space, they are different. In order to have a clone of an array be "equal" to the original, you must use the `equals()` method of the `java.util.Arrays` class.

```
String[] clone = (String[]) strarray.clone();
boolean b1 = Arrays.equals(strarray, clone); // Yes, they're equal
```

This will check for equality with each element. In the case where the arguments are arrays of objects, the `equals()` method of each object will be used to check for equality. `Arrays.equals()` works for arrays that are not clones, too. For more information on the `Arrays` class, see Chapter 13.

Array Reflection

If for some reason you are ever unsure whether an argument or object is an array, you can retrieve the object's `Class` object and ask it. The `isArray()` method of the `Class` class will tell you. Once you know you have an array, you can ask the `getComponentType()` method of `Class` what type of array you actually have. The `getComponentType()` method returns `null` if the `isArray()` method returns `false`. Otherwise, the `Class` type of the element is returned. You can recursively call `isArray()` if the array is multidimensional. It will still have only one component type. In addition, you can use the `getLength()` method of the `Array` class found in the `java.lang.reflect` package to discover the length of the array.

To demonstrate, Listing 2-3 shows that the argument to the `main()` method is an array of `java.lang.String` objects where the length is the number of command-line arguments specified:

Listing 2-3. Using reflection to check array type and length.

```
public class ArrayReflection {
  public static void main (String args[]) {
    printType(args);
  }
  private static void printType (Object object) {
    Class type = object.getClass();
    if (type.isArray()) {
      Class elementType = type.getComponentType();
      System.out.println("Array of: " + elementType);
      System.out.println(" Length: " + Array.getLength(object));
    }
  }
}
```

NOTE *If* `printType()` *was to be called with the previously defined* `buttons` *and* `components` *variables, each would state that the array is of the* `java.awt.Button` *type.*

If you don't use the `isArray()` and `getComponentType()` methods and you try to print the `Class` type for an array, you'll get a string that includes a [followed by a letter and the class name (or no class name if a primitive). For instance, if you tried to print out the `type` variable in the `printType()` method above, you would get `class [Ljava.lang.String;` as the output.

In addition to asking an object if it is an array and what type of array it is, you can also create arrays at runtime with the `java.lang.reflect.Array` class. This might be helpful to create generic utility routines that perform array tasks such as size doubling. (We'll return to that shortly.)

To create a new array, use the `newInstance()` method of `Array`, which comes in two varieties. For single dimension arrays you would normally use the simpler version, which acts like the statement `new type[length]` and returns the array as an object: `public static Object newInstance(Class type, int length)`. For instance, the following creates an array with room for five integers:

```
int array[] = (int[])Array.newInstance(int.class, 5);
```

NOTE *To specify the* `Class` *object for a primitive, just add* `.class` *to the end of the primitive type name. You can also use the* `TYPE` *variable of the wrapper classes, like* `Integer.TYPE`.

The second variety of the `newInstance()` method requires the dimensions to be specified as an array of integers: `public static Object newInstance(Class type, int dimensions[])`. In the simplest case of creating a single dimension array, you would create an array with only one element. In other words, if you were to create the same array of five integers, instead of passing the integer value of 5, you would need to create an array of the single element 5 to pass along to the `newInstance()` method:

```
int dimensions[] = {5};
int array[] = (int[])Array.newInstance(int.class, dimensions);
```

As long as you only need to create rectangular arrays, you can fill up the `dimensions` array with each array length. For example, the following is the equivalent of creating a 3 × 4 array of integers:

```
int dimensions[] = {3, 4};
int array[][] = (int[][])Array.newInstance(int.class, dimensions);
```

If, however, you need to create a non-rectangular array, you would need to call the `newInstance()` method several times. The first call would define the length of the outer array and would have what looks like a funny-looking class argument (`float[].class` for an array of floats). Each subsequent call would define the length of each inner array. For instance, the following demonstrates how to create an array of floats where the inner arrays are sized like a set of bowling pins: the first row with one element, the second with two, the third with three, and the fourth with four. To help you visualize this, recall the triangular array shown earlier in Figure 2-4.

```
float bowling[][] = (float[][])Array.newInstance(float[].class, 4);
for (int i=0; i<4; i++) {
  bowling[i] = (float[])Array.newInstance(float.class, i+1);
}
```

Once you've created your arrays at runtime, you can also get and set the elements of the array. This isn't normally done unless your square bracket keys on your keyboard aren't working or you're working in a dynamic programming environment where the array names were unknown when the program was created. As shown in Table 2-2, the `Array` class has a series of getter and setter methods for getting and setting the array elements. Which method you use depends upon the type of array you're working with.

Table 2-2. Array Getter and Setter Methods

GETTER METHODS	**SETTER METHODS**
get(Object array, int index)	set(Object array, int index, Object value)
getBoolean(Object array, int index)	setBoolean(Object array, int index, boolean value)
getByte(Object array, int index)	setByte(Object array, int index, byte value)
getChar(Object array, int index)	setChar(Object array, int index, char value)
getDouble(Object array, int index)	setDouble(Object array, int index, double value)
getFloat(Object array, int index)	setFloat(Object array, int index, float value)
getInt(Object array, int index)	setInt(Object array, int index, int value)
getLong(Object array, int index)	setLong(Object array, int index, long value)
getShort(Object array, int index)	setShort(Object array, int index, short value)

NOTE *You can always use the* `get()` *and* `set()` *methods. If the array is one of primitives, the return value of the* `get()` *method or the value argument to the* `set()` *method would be wrapped into the wrapper class for the primitive type, as in an* `Integer` *with an* `int` *array.*

Listing 2-4 provides a complete example of how to create, fill up, and display information about an array. Square brackets are used only in the `main()` method declaration.

Listing 2-4. Using reflection to create, fill, and display an array.

```
import java.lang.reflect.Array;
import java.util.Random;
public class ArrayCreate {
  public static void main (String args[]) {
    Object array = Array.newInstance(int.class, 3);
    printType(array);
    fillArray(array);
    displayArray(array);
  }
  private static void printType (Object object) {
    Class type = object.getClass();
    if (type.isArray()) {
      Class elementType = type.getComponentType();
      System.out.println("Array of: " + elementType);
      System.out.println("Array size: " + Array.getLength(object));
    }
  }
  private static void fillArray(Object array) {
    int length = Array.getLength(array);
    Random generator = new Random(System.currentTimeMillis());
    for (int i=0; i<length; i++) {
      int random = generator.nextInt();
      Array.setInt(array, i, random);
    }
  }
  private static void displayArray(Object array) {
    int length = Array.getLength(array);
    for (int i=0; i<length; i++) {
      int value = Array.getInt(array, i);
      System.out.println("Position: " + i + ", value: " + value);
    }
  }
}
```

When run, the output will look like the following (although the random numbers will differ):

```
Array of: int
Array size: 3
Position: 0, value: -54541791
Position: 1, value: -972349058
Position: 2, value: 1224789416
```

Let's return to our earlier example of creating a method that doubles the size of an array. Now that you know how to get an array's type, you can create a method that will double the size of any type of array. This method ensures that we have an array before getting its length and type. It then doubles the size of a new instance before copying over the original set of elements.

```
static Object doubleArray(Object original) {
  Object returnValue = null;
  Class type = original.getClass();
  if (type.isArray()) {
    int length = Array.getLength(original);
    Class elementType = type.getComponentType();
    returnValue = Array.newInstance(elementType, length*2);
    System.arraycopy(original, 0, returnValue, 0, length);
  }
  return returnValue;
}
```

Character Arrays

One last thing to mention before we wrap up our look at Java arrays: unlike C and C++, character arrays in Java are not strings. While you can easily go back and forth between a `String` and a `char[]`with the `String` constructor (which takes an array of `char` objects) and the `toCharArray()` method of `String`, they are definitely different.

Byte arrays are another case though. While they too are not strings, trying to go back and forth between a `byte[]` and a `String` involves a bit of work, since strings in Java are Unicode-based and 16 bits wide. You need to tell the `String` constructor what the encoding scheme is. Table 2-3 shows the primary available encoding schemes with the 1.3 platform. For a list of the extended set, see the online list at `http://java.sun.com/j2se/1.3/docs/guide/intl/encoding.doc.html`. These vary by JDK version.

Table 2-3. Primary Byte-to-Character Encoding Schemes

NAME	DESCRIPTION
ASCII	American Standard Code for Information Interchange
Cp1252	Windows Latin-1
ISO8859_1	ISO 8859-1, Latin alphabet No. 1
UnicodeBig	Sixteen-bit Unicode Transformation Format, big-endian byte order, with byte-order mark
UnicodeBigUnmarked	Sixteen-bit Unicode Transformation Format, big-endian byte order
UnicodeLittle	Sixteen-bit Unicode Transformation Format, little-endian byte order, with byte-order mark
UnicodeLittleUnmarked	Sixteen-bit Unicode Transformation Format, little-endian byte order
UTF16	Sixteen-bit Unicode Transformation Format, byte order specified by a mandatory initial byte-order mark
UTF8	Eight-bit Unicode Transformation Format

If you do specify an encoding, you must place the call to the `String` constructor within a try-catch block because an `UnsupportedEncodingException` can be thrown if the specified encoding scheme is invalid.

If you are working only with ASCII characters, you really don't have to worry much here. Passing a `byte[]` to the `String` constructor without any encoding scheme argument uses the platform's default encoding, which is sufficient. Of course, to be safe, you can always just pass "ASCII" as the scheme.

NOTE *To check the default on your platform, look at the "file.encoding" system property.*

Summary

Arrays in Java seem easy to work with, but there are many things to be aware of to fully utilize their capabilities. While basic array declaration and usage can be considered simple, there are different things to be concerned about when working with arrays of primitives and objects, as well as multidimensional arrays. Array initialization doesn't have to be complicated, but throw in anonymous arrays and things get more involved.

Once you have an array, it takes a little thought to figure out how best to work with it. You need to take special care when passing arrays to methods as they are passed by reference. If you outgrow the original array size, you'll need to make a copy with additional space. Array cloning lets you pass around a copy without worrying about the idiosyncrasies of the `final` keyword. When assigning arrays to other variables, be careful not to run across an `ArrayStoreException` as it can be ugly to deal with at runtime. Equality checking of arrays can involve either checking for the same memory area or checking for equivalently valued elements. Through the magic of Java reflection, you can manipulate objects that happen to be arrays. The last thing you learned in this chapter was how to convert between byte arrays and strings, and how byte arrays are not strings by default as they are in other languages.

In the next chapter, we'll explore the many facets of working with vectors in Java. We'll learn about the inner workings of vectors and how best to deal with concerns like type safety.