

```
String s = "This is a test of the " + // Do this instead.
"emergency broadcast system";
```

This concatenation of literals is done when your program is compiled, not when it is run, so you do not need to worry about any kind of performance penalty.

The second class that supports its own special object literal syntax is the class named `Class`. `Class` is a (self-referential) data type that represents all Java data types, including primitive types and array types, not just class types. To include a `Class` object literally in a Java program, follow the name of any data type with `.class`. For example:

```
Class typeInt = int.class;
Class typeIntArray = int[].class;
Class typePoint = Point.class;
```

This feature is supported by Java 1.1 and later.

The Java reserved word `null` is a special literal that can be used with any class. Instead of representing a literal object, it represents the absence of an object. For example:

```
String s = null;
Point p = null;
```

Finally, objects can also be included literally in a Java program through the use of a construct known as an anonymous inner class. Anonymous classes are discussed in Chapter 3.

Using an Object

Now that we've seen how to define classes and instantiate them by creating objects, we need to look at the Java syntax that allows us to use those objects. Recall that a class defines a collection of fields and methods. Each object has its own copies of those fields and has access to those methods. We use the dot character (`.`) to access the named fields and methods of an object. For example:

```
Point p = new Point(2, 3); // Create an object
double x = p.x; // Read a field of the object
p.y = p.x * p.x; // Set the value of a field
double d = p.distanceFromOrigin(); // Access a method of the object
```

This syntax is central to object-oriented programming in Java, so you'll see it a lot. Note, in particular, the expression `p.distanceFromOrigin()`. This tells the Java compiler to look up a method named `distanceFromOrigin()` defined by the class `Point` and use that method to perform a computation on the fields of the object `p`. We'll cover the details of this operation in Chapter 3.

Array Types

Array types are the second kind of reference types in Java. An array is an ordered collection, or numbered list, of values. The values can be primitive values, objects, or even other arrays, but all of the values in an array must be of the same type.

From:

Java In A Nutshell

64 Chapter 2 - Java Syntax from the Ground Up

The type of the array is the type of the values it holds, followed by the characters `[]`. For example:

```
byte b; // byte is a primitive type
byte[] arrayOfBytes; // byte[] is an array type: array of byte
Point[] points; // Point[] is an array of Point objects
```

For compatibility with C and C++, Java also supports another syntax for declaring variables of array type. In this syntax, one or more pairs of square brackets follow the name of the variable, rather than the name of the type:

```
byte arrayOfBytes[]; // Same as byte[] arrayOfBytes
byte arrayOfArrayOfBytes[][]; // Same as byte[][] arrayOfArrayOfBytes
byte[] arrayOfArrayofBytes[]; // Ugh! Same as byte[][] arrayOfArrayOfBytes
```

This is almost always a confusing syntax, however, and it is not recommended.

With classes and objects, we have separate terms for the type and the values of that type. With arrays, the single word array does double duty as the name of both the type and the value. Thus, we can speak of the array type `int[]` (a type) and an array of `int` (a particular array value). In practice, it is usually clear from context whether a type or a value is being discussed.

Creating Arrays

To create an array value in Java, you use the new keyword, just as you do to create an object. Arrays don't need to be initialized like objects do, however, so you don't pass a list of arguments between parentheses. What you must specify, though, is how big you want the array to be. If you are creating a `byte[]`, for example, you must specify how many `byte` values you want it to hold. Array values have a fixed size in Java. Once an array is created, it can never grow or shrink. Specify the desired size of your array as a non-negative integer between square brackets:

```
byte[] buffer = new byte[1024];
String[] lines = new String[50];
```

When you create an array with this syntax, each of the values held in the array is automatically initialized to its default value. This is `false` for boolean values, `\u0000` for char values, 0 for integer values, 0.0 for floating-point values, and `null` for objects or array values.

Using Arrays

Once you've created an array with the new operator and the square-bracket syntax, you also use square brackets to access the individual values contained in the array. Remember that an array is an ordered collection of values. The elements of an array are numbered sequentially, starting with 0. The number of an array element refers to the element. This number is often called the *index*, and the process of looking up a numbered value in an array is sometimes called *indexing* the array.

To refer to a particular element of an array, simply place the index of the desired element in square brackets after the name of the array. For example:

```
String[] responses = new String[2]; // Create an array of two strings
responses[0] = "Yes"; // Set the first element of the array
responses[1] = "No"; // Set the second element of the array

// Now read these array elements
System.out.println(question + " (" + responses[0] + "/" +
    responses[1] + "): ");
```

In some programming languages, such as C and C++, it is a common bug to write code that tries to read or write array elements that are past the end of the array. Java does not allow this. Every time you access an array element, the Java interpreter automatically checks that the index you have specified is valid. If you specify a negative index or an index that is greater than the last index of the array, the interpreter throws an exception of type `ArrayIndexOutOfBoundsException`. This prevents you from reading or writing nonexistent array elements.

Array index values are integers; you cannot index an array with a floating-point value, a `boolean`, an object, or another array. `char` values can be converted to `int` values, so you *can* use characters as array indexes. Although `long` is an integer data type, `long` values cannot be used as array indexes. This may seem surprising at first, but consider that an `int` index supports arrays with over two billion elements. An `int[]` with this many elements would require eight gigabytes of memory. When you think of it this way, it is not surprising that `long` values are not allowed as array indexes.

Besides setting and reading the value of array elements, there is one other thing you can do with an array value. Recall that whenever we create an array, we must specify the number of elements the array holds. This value is referred to as the length of the array; it is an intrinsic property of the array. If you need to know the length of the array, append `.length` to the array name:

```
if (errorCode < errorMessages.length)
    System.out.println(errorMessages[errorCode]);
```

Every array has a `length` field that specifies the number of elements it contains. Note that this field is read-only: you can use it to read the length of the array, but you cannot assign any value to it or use it to set or change the length of an array.

In the previous example, the array index within square brackets is a variable, not an integer literal. In fact, arrays are most often used with loops, particularly for loops, where they are indexed using a variable that is incremented or decremented each time through the loop:

```
int[] values; // Assume array is created and initialized elsewhere
int total = 0; // Score sum of elements here
for(int i = 0; i < values.length; i++) // Loop through array elements
    total += values[i]; // Add them up
```

In Java, the first element of an array is always element number 0. If you are accustomed to a programming language that numbers array elements beginning with 1, this will take some getting used to. For an array `a`, the first element is `a[0]`, the second element is `a[1]`, and the last element is:

```
a[a.length - 1] // The last element of any array named a
```

Array Literals

The `null` literal used to represent the absence of an object can also be used to represent the absence of an array. For example:

```
char[] password = null;
```

In addition to the `null` literal, Java also defines special syntax that allows you to specify array values literally in your programs. There are actually two different syntaxes for array literals. The first, and more commonly used, syntax can be used only when declaring a variable of array type. It combines the creation of the array object with the initialization of the array elements:

```
int[] powersOfTwo = {1, 2, 4, 8, 16, 32, 64, 128};
```

This creates an array that contains the eight `int` elements listed within the curly braces. Note that we don't use the new keyword or specify the type of the array in this array literal syntax. The type is implicit in the variable declaration of which the initializer is a part. Also, the array length is not specified explicitly with this syntax; it is determined implicitly by counting the number of elements listed between the curly braces. There is a semicolon following the close curly brace in this array literal. This is one of the fine points of Java syntax. When curly braces delimit classes, methods, and compound statements, they are not followed by semicolons. However, for this array literal syntax, the semicolon is required to terminate the variable declaration statement.

The problem with this array literal syntax is that it works only when you are declaring a variable of array type. Sometimes you need to do something with an array value (such as pass it to a method) but are going to use the array only once, so you don't want to bother assigning it to a variable. In Java 1.1 and later, there is an array literal syntax that supports this kind of anonymous arrays (so called because they are not assigned to variables, so they don't have names). This kind of array literal looks as follows:

```
// Call a method, passing an anonymous array literal that contains two strings
String response = askQuestion("Do you want to quit?",
    new String[] {"Yes", "No"});

// Call another method with an anonymous array (of anonymous objects)
double d = computeAreaOfTriangle(new Point[] { new Point(1,2),
    new Point(3,4),
    new Point(3,2) });
```

With this syntax, you use the new keyword and specify the type of the array, but the length of the array is not explicitly specified.

It is important to understand that the Java Virtual Machine architecture does not support any kind of efficient array initialization. In other words, array literals are created and initialized when the program is run, not when the program is compiled. Consider the following array literal:

```
int[] perfectNumbers = {6, 28};
```