

Array Fundamentals

- To create an array, use the syntax **DataType[] variable = new DataType[ArraySize]**. Alternatively, if you know the data beforehand, you can write **DataType[] variable = {item 1, item 2,...item n}**
 - All elements of the array will be automatically initialized with the *default value* for that datatype. This is *false* for boolean's, *0* for all numeric primitive types, and *null* for all reference types. So for example, the previous note created an array of `DataType` references, all of which are initialized to *null*.
- To access an item, use the syntax **variable[i]** where *i* is the index
- To set an item, use the syntax **variable[i] = data**
- To find the length of an array, use the syntax **variable.length**

Two-Dimensional Arrays

A two dimensional array is represented by an array of an array. Because an array is also an object, like any other object having the Object as the super class, it can be used to create an array where the element of the array is also an array. In this way an array with any number of dimensions can be created. Here are examples of two dimensional arrays with initializer blocks:

```
String [][] twoDimArray = { {"00", "01", "02", "03", "04"},
                             {"10", "11", "12", "13", "14"},
                             {"20", "21", "22", "23", "23"} };
...
int [][] twoDimIntArray = { {00, 01, 02, 03, 04},
                              {10, 11, 12, 13, 14},
                              {20, 21, 22, 23, 24} };
```

Note that the above "twoDimArray" is equivalent to the following more verbose code:

```
String [][] twoDimArray = new String[3][];
for (int i = 0; i < twoDimArray.length; i++) {
    twoDimArray[i] = new String[5];
    for (int j = 0; j < twoDimArray[i].length; j++)
        twoDimArray[i][j] = "" + i + j;
}
```

In the above example we defined an array which has three elements, each element contains an array having 5 elements. We could create the array having the 5 elements first and use that one in the initialize block.

[illegible]

Since they are arrays of array references, these multi-dimensional arrays can be "jagged" (i.e. subarrays can have different lengths), or the subarray reference can even be null. Consider

[illegible]

Multidimensional Array

Going further any number of dimensional array can be defined.

```
<elementType>[] [] ... [] <arrayName>
or
<elementType><arrayName>[] [] ... []
```

Java Programming		Index	
Overview	Getting Started	Fundamentals	Classes and Objects
Collections	Exceptions	Concurrent Programming	Reflection
Applets	JavaBeans	Libraries	

Data and Variables

Navigate **Language Fundamentals** topic:

A variable in Java can store two kinds of variables:

- Java primitive type values
- a reference to a Java object

Java's primitive types are

- integers (whole numbers, declared as *byte*, *short*, *int*, or *long*; only *int* need be of interest to a beginner)
- floating-point numbers (decimal numbers, declared as *float* or *double*; only *float* need be of interest at first)
- characters (declared as *char*, representing one character like 'A' or ',')
- boolean (holding only *true* or *false* as values)

In addition, the Java language has special features for its *String* class, and strings can be treated very much like primitives for many purposes.

As in most languages, a variable is declared to be a particular type of data; the syntax for a declaration is:

```
variabletype variablename;
```

To store a value in a variable, a program statement like

```
variablename = data;
```

And can reference the variable (and use the data stored in it) by its name.

For example, to create an *int* primitive type value, named **yr** that stores 2007;

```
int yr = 2007;
```

To access the data in *yr*, use the variable in place of the number.

```
System.out.println(yr);
```

Produces

```
2007
```

Strong Typing

Variables in Java are *strongly typed*, which means that the compiler checks the type of a variable matches the type of data stored in that variable. If you declare a variable to hold a String, for instance, you cannot assign an integer value to that variable. Some languages (such as C) define an interpretation of such a statement and use that interpretation without any warning; others (such as PL/I) define a conversion for almost all such statements and perform the conversion to complete the assignment. Strong typing is intended to prevent mistakes made by unwittingly assigning the wrong kind of value to a variable, and catching those mistakes when the program is compiled rather than waiting to find it when the program is running.

Case Conventions

Java is case-sensitive. A method called `mymethod` is completely separate from a method called `myMethod`. Be careful!

By convention, most identifiers that includes more than one word uses camel case. Classes begin with a capital letter; methods and variables do not. (Constructors have to start with a capital, because they must have the same name as the class.) Package names use lowercase, and do not use camel case. Thus:

```
package org.wikibooks.samplecode;

class CaseConventions {
    int variable;
    int multipleWordVariable;

    CaseConventions(String id) {
    }

    void method() {
    }

    void longMethodName() {
    }
}
```

Scope

Java uses **block scope**, which means that a variable is "un-defined" (and becomes useless) at the end of the block in which it is defined. A *block* is any section of code within curly braces. Common blocks include class definitions, methods and constructors, if/else blocks, and for, while, and do-while loops.

```
class BlockScope {
    int classScope; // valid in all of class BlockScope

    BlockScope(int param) { // param is valid only in this constructor
        int localVariable = 0; // valid only in this constructor
    }

    void someMethod() {
        int local = 42; // valid only in this method
    }
}
```