

```

    public static int getInt(String prompt) {
        System.out.print(prompt + " ");
        int integer = input.nextInt();
        input.nextLine(); // Get rid of this line
                          // so that getString won't read it
        return integer;
    }

    public static String getString(String prompt) {
        System.out.print(prompt + " ");
        return input.nextLine();
    }
}

```

Copy the code and compile it. The program will ask you to enter some names then reprints the names in order. It demonstrates three major aspects of arrays: how to define an array, how to set data, and how to access it. The code `String[] names = new String[numNames];` tells Java to create an array of size *numNames* that will store Strings. To set data, use `names[x] = data` where *x* is the index to access. Note that all Java arrays start at 0 and go to (array size - 1). Thus, if you dimension an array to size 10, the highest index is 9.

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.

```

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

```

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

```

String [][] weirdTwoDimArray = { {"10", "11", "12"},
                                   null,
                                   {"20", "21", "22", "23", "24"} };

```

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	

Collection Classes

Navigate **Collection** topic:

Collections are a group of objects bound together by a common characteristic. Java has various built-in classes to support the collection of objects, either of the same type or general. The most basic construct to work with is the *array* of which you will come to know of in a section later in this chapter.

Introduction to Collections

The most basic collection interface is called `Collection`. This interface gives the user a generic usage of a collection.

```
import java.util.Collection;    // Interface
import java.util.ArrayList;     // Implementation
import java.util.HashSet;      // Implementation
...
Collection coll1 = new ArrayList();
Collection coll2 = new HashSet();
...
< Use coll1 & coll2 >
```

In the above there are two collections. The usage of the collections are the same, the implementations are different. If the existing collection implementations do not meet your needs, you can write your version of the implementation. Your version of the implementation just needs to implement the same `java.util.Collection` interface, then you can switch to using your implementation and the code that is using the collection does not need to be changed.

```
import java.util.Collection;
import com.yourcomp.util.YourCollectionImpl;
...
Collection coll1 = new YourCollectionImpl();
Collection coll2 = new YourCollectionImpl();
...
< Use coll1 & coll2 >
```

The Java JDK collection implementations are quite powerful and good, so it is unlikely that you will need to write your own.

All collections contain object references. Because of that, if the object is changed after it was put in the collection, the object that is 'in' the collection also 'changes'. The object is not really in the collection, only the object reference is. It is not guaranteed that the objects 'inside' the collections won't change. This is an issue only if you put an actively used object in the collection. In that case when you are adding an object that could change any time you need to make a copy or clone of the object. A new object will be created and its reference will be put in the collection. In that case there will be no object references outside of the collection, so the objects 'inside' the collection can only be changed if we take out an object reference from the collection.

Aside from the `java.util.Collection` interface, the Java JDK has the `java.util.Map` interface as well. This defines key value mappings. Implementations of the `Map` interface do not contain collections of objects. Instead they contain collections of key->value mappings.

```
import java.util.Map;
import java.util.Hashtable;
...
Map map = new Hashtable();
...
map.put( key, value );
```

All collections need to have the same basic operations. Those are:

- Adding element(s) to the collection
- Removing element(s) from the collection
- Obtaining the number of elements in the collection