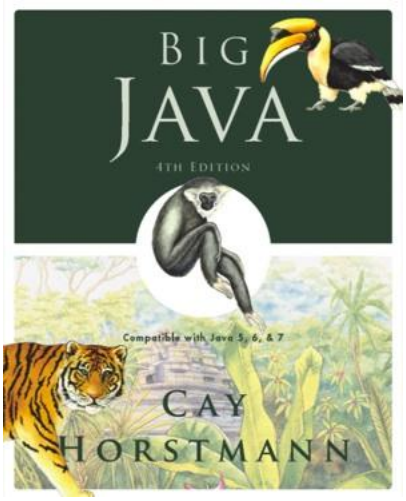


Chapter 7 – Arrays and Array Lists

Chapter Goals

- To become familiar with using arrays and array lists
- To learn about wrapper classes, auto-boxing and the generalized for loop
- To study common array algorithms
- To learn how to use two-dimensional arrays
- To understand when to choose array lists and arrays in your programs
- To implement partially filled arrays
- To understand the concept of regression testing

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Arrays

- Array: Sequence of values of the same type
- Construct array:

```
new double[10]
```

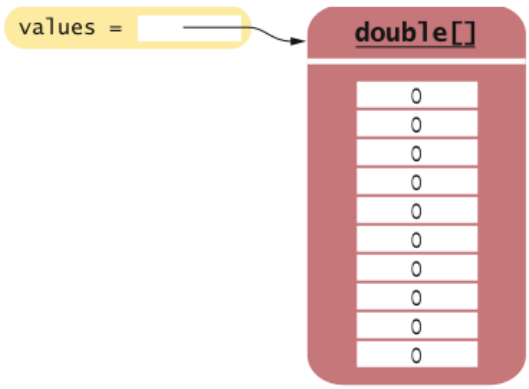
- Store in variable of type `double[]`:

```
double[] data = new double[10];
```

- When array is created, all values are initialized depending on array type:
  - *Numbers:* `0`
  - *Boolean :* `false`
  - *Object References:* `null`

Arrays

Figure 1
An Array Reference and an Array



Arrays

Use `[]` to access an element:

```
values[2] = 29.95;
```

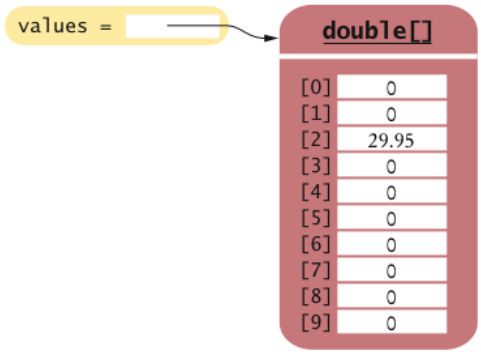


Figure 2
Modifying an Array Element

Arrays

- Using the value stored:

```
System.out.println("The value of this data item is "
   + values[2]);
```

- Get array length as `values.length` (Not a method!)
- Index values range from `0` to `length - 1`
- Accessing a nonexistent element results in a **bounds error**:

```
double[] values = new double[10];
values[10] = 29.95; // ERROR
```

- Limitation: Arrays have fixed length

Declaring Arrays

Table 1 Declaring Arrays

`int[] numbers = new int[10];`	An array of ten integers. All elements are initialized with zero.
`final int NUMBERS_LENGTH = 10;` `int[] numbers = new int[NUMBERS_LENGTH];`	It is a good idea to use a named constant instead of a "magic number".
`int valuesLength = in.nextInt();` `double[] values = new double[valuesLength];`	The length need not be a constant.
`int[] squares = { 0, 1, 4, 9, 16 };`	An array of five integers, with initial values.
`String[] names = new String[3];`	An array of three string references, all initially `null`.
`String[] friends = { "Emily", "Bob", "Cindy" };`	Another array of three strings.
`double[] values = new int[10]`	**Error:** You cannot initialize a `double[]` variable with an array of type `int[]`.

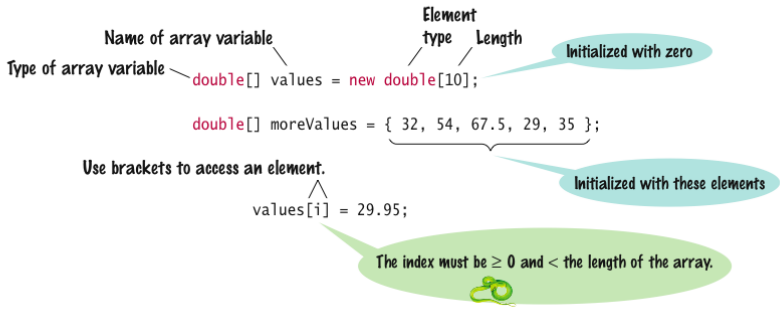
Syntax 7.1 Arrays

Syntax

To construct an array: `new typeName[length]`

To access an element: `arrayReference[index]`

Example



Self Check 7.1

What elements does the data array contain after the following statements?

```
double[] values = new double[10];
for (int i = 0; i < values.length; i++)
   values[i] = i * i;
```

Answer: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, but not 100

Self Check 7.2

What do the following program segments print? Or, if there is an error, describe the error and specify whether it is detected at compile-time or at run-time.

a)
```
double[] a = new double[10];
System.out.println(a[0]);
```

b)
```
double[] b = new double[10];
System.out.println(b[10]);
```

c)
```
double[] c;
System.out.println(c[0]);
```

Answer:

a) 0

b) a run-time error: array index out of bounds

c) a compile-time error: c is not initialized

Make Parallel Arrays into Arrays of Objects

```
// Don't do this
int[] accountNumbers;
double[] balances;
```

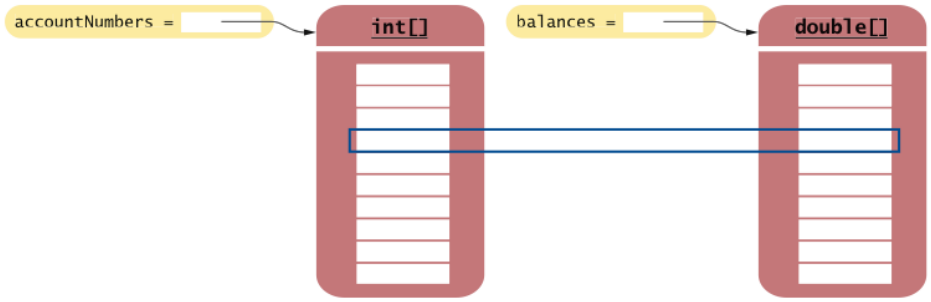


Figure 3 Avoid Parallel Arrays

Make Parallel Arrays into Arrays of Objects

Avoid parallel arrays by changing them into arrays of objects:

```
BankAccount[] accounts;
```

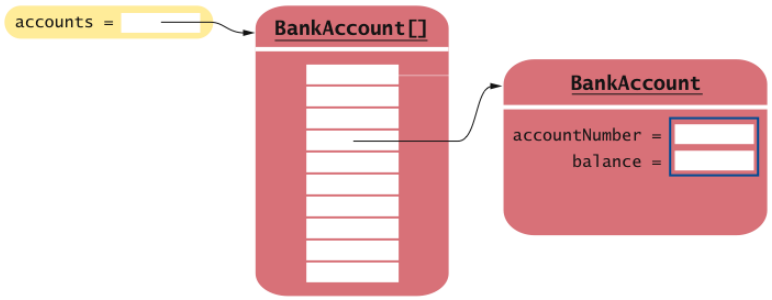


Figure 4 Reorganizing Parallel Arrays into an Array of Objects

Array Lists

- `ArrayList` class manages a sequence of objects
- Can grow and shrink as needed
- `ArrayList` class supplies methods for many common tasks, such as inserting and removing elements
- `ArrayList` is a **generic class**:

  `ArrayList<T>`

  collects objects of **type parameter** `T`:

```
ArrayList<String> names = new ArrayList<String>();
names.add("Emily");
names.add("Bob");
names.add("Cindy");
```

- `size` method yields number of elements

Adding Elements

To add an object to the end of the array list, use the `add` method:

```
names.add("Emily");
names.add("Bob");  (1)
names.add("Cindy");  (2)
```

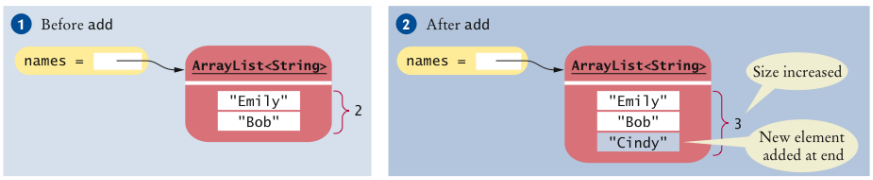


Figure 5 Adding an Element with add

Retrieving Array List Elements

- To obtain the value an element at an index, use the `get` method
- Index starts at 0
-
```
String name = names.get(2);
// gets the third element of the array list
```
- Bounds error if index is out of range
- Most common bounds error:

```
int i = names.size();
name = names.get(i); // Error
// legal index values are 0 ... i-1
```


Setting Elements

- To set an element to a new value, use the `set` method:

```
names.set(2, "Carolyn");
```

Removing Elements

- To remove an element at an index, use the `remove` method:

```
names.remove(1);
```


Adding and Removing Elements

```
names.add("Emily");
names.add("Bob");
names.add("Cindy");
names.set(2, "Carolyn");  1
names.add(1, "Ann");  2
names.remove(1);  3
```

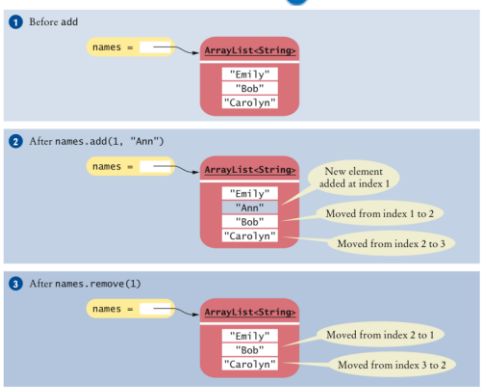


Figure 6 Adding and Removing Elements in the Middle of an Array List

Working with Array Lists

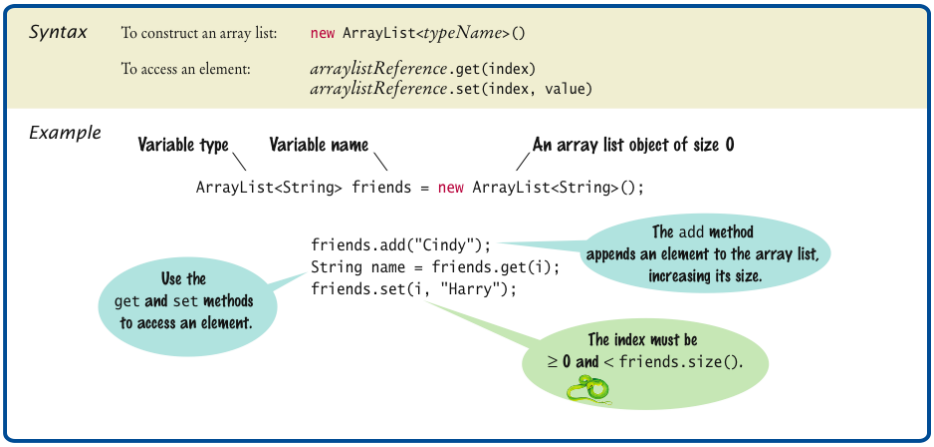
`ArrayList<String> names = new ArrayList<String>();`	Constructs an empty array list that can hold strings.
`names.add("Ann");` `names.add("Cindy");`	Adds elements to the end.
`System.out.println(names);`	Prints `[Ann, Cindy]`.
`names.add(1, "Bob");`	Inserts an element at index 1. `names` is now `[Ann, Bob, Cindy]`.
`names.remove(0);`	Removes the element at index 0. `names` is now `[Bob, Cindy]`.
`names.set(0, "Bill");`	Replaces an element with a different value. `names` is now `[Bill, Cindy]`.

Working with Array Lists (cont.)

`String name = names.get(i);`	Gets an element.
`String last = names.get(names.size() - 1);`	Gets the last element.

`ArrayList<Integer> squares = new ArrayList<Integer>();`
`for (int i = 0; i < 10; i++)`
`{`
`  squares.add(i * i);`
`}`
	Constructs an array list holding the first ten squares.

Syntax 7.2 Array Lists



Self Check 7.3

How do you construct an array of 10 strings? An array list of strings?

Answer:

```
new String[10];
new ArrayList<String>();
```

Self Check 7.4

What is the content of `names` after the following statements?

```
ArrayList<String> names = new ArrayList<String>();
names.add("A");
names.add(0, "B");
names.add("C");
names.remove(1);
```

Answer: `names` contains the strings `"B"` and `"C"` at positions 0 and 1

Wrapper Classes

- For each primitive type there is a **wrapper class** for storing values of that type:

```
Double d = new Double(29.95);
```

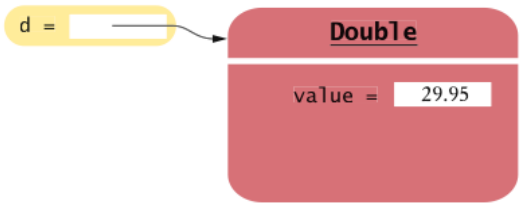


Figure 7 An Object of a Wrapper Class

- Wrapper objects can be used anywhere that objects are required instead of primitive type values:

```
ArrayList<Double> values= new ArrayList<Double>();
data.add(29.95);
double x = data.get(0);
```

Wrappers

There are wrapper classes for all eight primitive types:

Primitive Type	Wrapper Class
byte	Byte
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

Auto-boxing

- **Auto-boxing:** Automatic conversion between primitive types and the corresponding wrapper classes:

```
Double d = 29.95; // auto-boxing; same as
                  // Double d = new Double(29.95);
double x = d; // auto-unboxing; same as
              // double x = d.doubleValue();
```

- Auto-boxing even works inside arithmetic expressions:

```
d = d + 1;
```

  Means:

  - *auto-unbox* `d` *into a* `double`
  - *add* `1`
  - *auto-box the result into a new* `Double`
  - *store a reference to the newly created wrapper object in* `d`

Auto-boxing and Array Lists

- To collect numbers in an array list, use the wrapper type as the type parameter, and then rely on auto-boxing:

```
ArrayList<Double> values = new ArrayList<Double>();
values.add(29.95);
double x = values.get(0);
```

- Storing wrapped numbers is quite inefficient
  - *Acceptable if you only collect a few numbers*
  - *Use arrays for long sequences of numbers or characters*

Self Check 7.5

What is the difference between the types `double` and `Double`?

Answer: `double` is one of the eight primitive types. `Double` is a class type.

The Enhanced `for` Loop

- Traverses all elements of a collection:

```
double[] values = ...;
double sum = 0;
for (double element : values)
{
   sum = sum + element;
}
```

- Read the loop as "for each `element` in `values`"
- Traditional alternative:

```
double[] values = ...;
double sum = 0;
for (int i = 0; i < values.length; i++)
{
   double element = values[i];
   sum = sum + element;
}
```


The Enhanced `for` Loop

- Works for `ArrayLists` too:

```
ArrayList<BankAccount> accounts = ...;
double sum = 0;
for (BankAccount account : accounts)
{
   sum = sum + aaccount.getBalance();
}
```

- Equivalent to the following ordinary `for` loop:

```
double sum = 0;
for (int i = 0; i < accounts.size(); i++)
{
   BankAccount account = accounts.get(i);
   sum = sum + account.getBalance();
}
```

The Enhanced `for` Loop

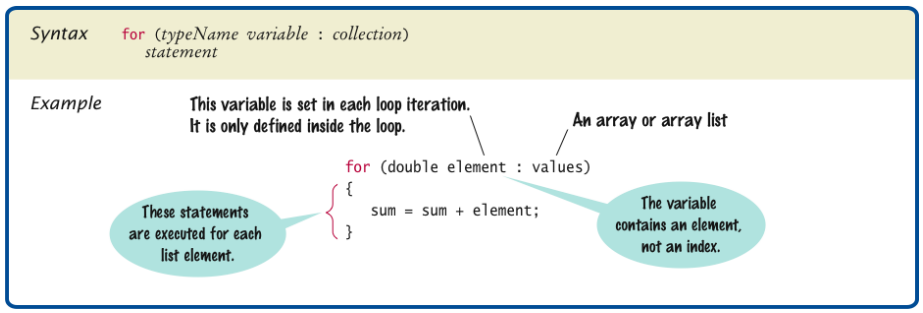
- The "for each loop" does not allow you to modify the contents of an array:

```
for (double element : values)
{
   element = 0;
   // ERROR—this assignment does not
   // modify array element
}
```

- Must use an ordinary `for` loop:

```
for (int i = 0; i < values.length; i++)
{
   values[i] = 0; // OK
}
```


Syntax 7.3 The "for each" Loop



Self Check 7.7

Write a "for each" loop that prints all elements in the array `values`.

Answer:

```
for (double element : values)
    System.out.println(element);
```


Self Check 7.8

What does this "for each" loop do?

```
int counter = 0;
for (BankAccount a : accounts)
{
   if (a.getBalance() == 0) { counter++; }
}
```

Answer: It counts how many accounts have a zero balance.

Partially Filled Arrays

- Array length = maximum number of elements in array
- Usually, array is partially filled
- Need companion variable to keep track of current size
  - *Uniform naming convention:*

```
final int VALUES_LENGTH = 100;
double[] values = new double[VALUES_LENGTH];
int valuesSize = 0;
```

- Update `valuesSize` as array is filled:

```
values[valuesSize] = x;
valuesSize++;
```


Partially Filled Arrays

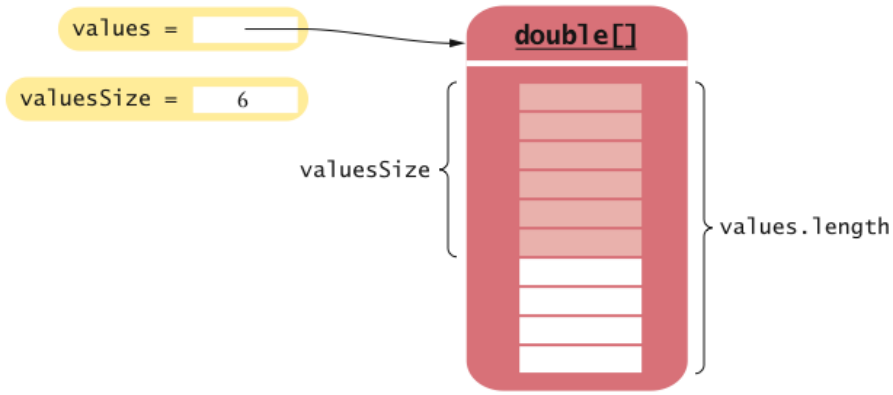


Figure 8 A Partially Filled Array

Partially Filled Arrays

- Example: Read numbers into a partially filled array:

```
int valuesSize = 0;
Scanner in = new Scanner(System.in);
while (in.hasNextDouble())
{
   if (valuesSize < values.length)
   {
      values[valuesSize] = in.nextDouble();
      valuesSize++;
   }
}
```

- To process the gathered array elements, use the companion variable, not the array length:

```
for (int i = 0; i < valuesSize; i++)
{
   System.out.println(values[i]);
}
```


Self Check 7.9

Write a loop to print the elements of the partially filled array `values` in reverse order, starting with the last element.

Answer:

```
for (int i = valuesSize - 1; i >= 0; i--)
   System.out.println(values[i]);
```

Self Check 7.10

How do you remove the last element of the partially filled array `values`?

Answer:

```
valuesSize--;
```


Common Array Algorithm: Finding the Maximum or Minimum

- Initialize a candidate with the starting element
- Compare candidate with remaining elements
- Update it if you find a larger or smaller value

Common Array Algorithm: Finding the Maximum or Minimum

- Example: Find the account with the largest balance in the bank:

```
BankAccount largestYet = accounts.get(0);
for (int i = 1; i < accounts.size(); i++)
{
   BankAccount a = accounts.get(i);
   if (a.getBalance() > largestYet.getBalance())
      largestYet = a;
}
return largestYet;
```

- Works only if there is at least one element in the array list — if list is empty, return `null`:

```
if (accounts.size() == 0) return null;
BankAccount largestYet = accounts.get(0);
...
```


Common Array Algorithm: Searching for a Value

- Check all elements until you have found a match
- Example: Determine whether there is a bank account with a particular account number in the bank:

```
public class Bank
{
   public BankAccount find(int accountNumber)
   {
      for (BankAccount account : accounts)
      {
         if (account.getAccountNumber() == accountNumber)
            // Found a match
            return account;
      }
      return null; // No match in the entire array list
   }
   ...
}
```

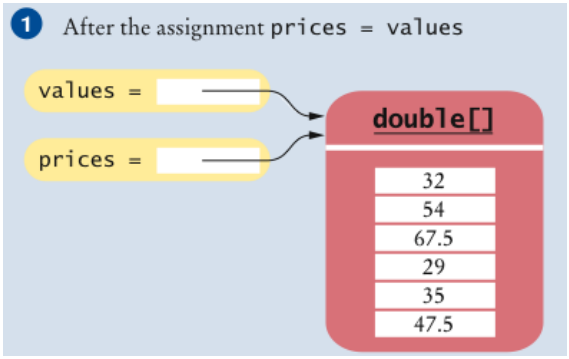
Common Array Algorithm: Searching for a Value

- The process of checking all elements until you have found a match is called a **linear search**

Common Array Algorithm: Copying an Array

- Copying an array variable yields a second reference to the same array:

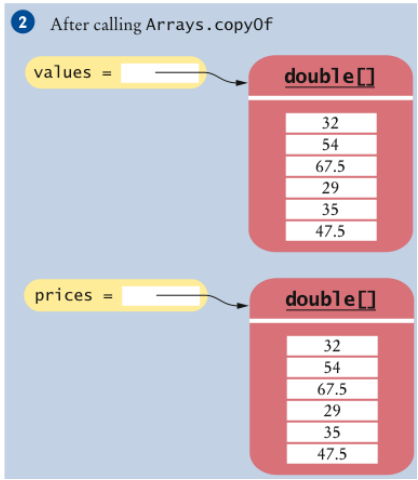
```
double[] values = new double[6];
. . .  // Fill array
double[] prices = values;
```



Common Array Algorithm: Copying an Array

- To make a true copy of an array, call the `Arrays.copyOf` method:

```
double[] prices = Arrays.copyOf(values, values.length);
```



Common Array Algorithm: Copying an Array

- To grow an array that has run out of space, use the `Arrays.copyOf` method:

```
values = Arrays.copyOf(values, 2 * values.length);
```

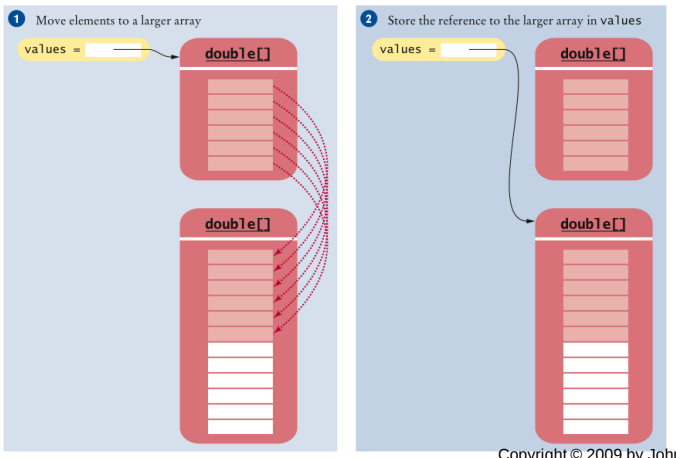


Figure 14 Growing an Array

Common Array Algorithm: Growing an Array

- Example: Read an arbitrarily long sequence numbers into an array, without running out of space:

```
int valuesSize = 0;
while (in.hasNextDouble())
{
   if (valuesSize == values.length)
      values = Arrays.copyOf(values, 2 * values.length);
   values[valuesSize] = in.nextDouble();
   valuesSize++;
}
```


Regression Testing

- **Test suite:** a set of tests for repeated testing
- **Cycling:** bug that is fixed but reappears in later versions
- **Regression testing:** repeating previous tests to ensure that known failures of prior versions do not appear in new versions

Two-Dimensional Arrays

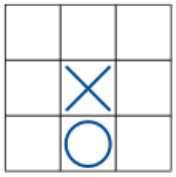


Figure 15 A Tic-Tac-Toe Board

- When constructing a two-dimensional array, specify how many rows and columns are needed:

```
final int ROWS = 3;
final int COLUMNS = 3;
String[][] board = new String[ROWS][COLUMNS];
```

- Access elements with an index pair:

```
board[1][1] = "x";
board[2][1] = "o";
```


Traversing Two-Dimensional Arrays

- It is common to use two nested loops when filling or searching:

```
for (int i = 0; i < ROWS; i++)
   for (int j = 0; j < COLUMNS; j++)
      board[i][j] = " ";
```

Traversing Two-Dimensional Arrays

- You can also recover the array dimensions from the array variable:
  - *board.length is the number of rows*
  - *board[0].length is the number of columns*
- Rewrite the loop for filling the tic-tac-toe board:

```
for (int i = 0; i < board.length; i++)
   for (int j = 0; j < board[0].length; j++)
      board[i][j] = " ";
```