CHAPTER 6

ARRAYS and ARRAYLISTS

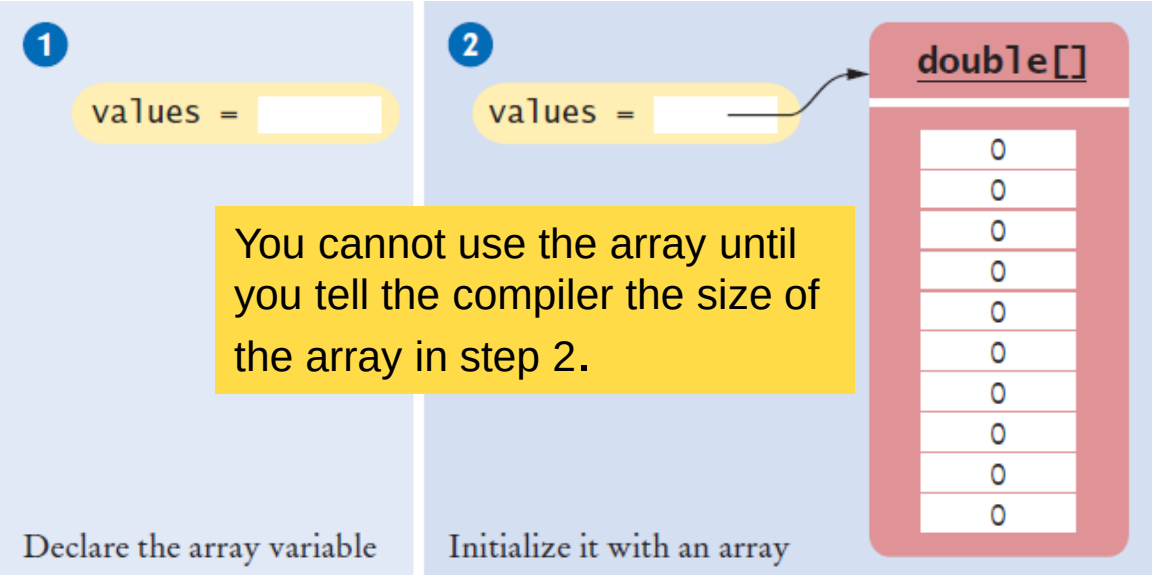
Slides by Donald W. Smith
TechNeTrain.com

Final Draft
10/30/2011

Declaring an Array

- Declaring an array is a **two-step** process

```
1)  double[] values;    // declare array variable
2)  values = new double[10];  // initialize array
```



Declaring an Array (Step 1)

- Make a named 'list' with the following parts:

Type	Square Braces	Array name	semicolon
`double`	`[ ]`	`values`	`;`

  - You are declaring that
    - There is an array named `values`
    - The elements inside are of type `double`
    - You have not (YET) declared how many elements are in inside

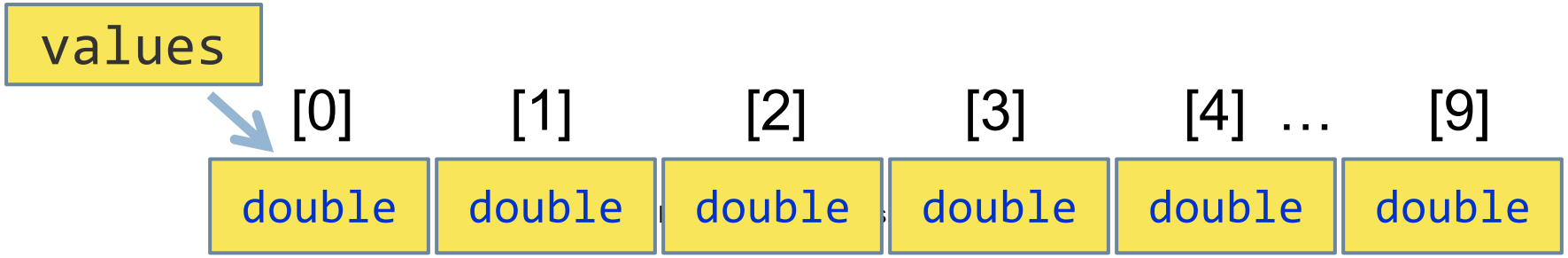
- Other Rules:
  - Arrays can be declared anywhere you can declare a variable
  - Do not use 'reserved' words or already used names

Declaring an Array (Step 2)

- Reserve memory for all of the elements:

Array name	Keyword	Type	Size	semicolon
`values`	`= new`	`double`	`[10]`	`;`

  - You are reserving memory for:
    - Array `values` needs storage for `[10]` elements the size of type `double`
  - You are also setting up the array variable
  - Now the compiler knows how many elements there are
    - You cannot change the size after you declare it!



One Line Array Declaration

- Declare and Create on the same line:

Type	Braces	Array name		Keyword	Type	Size	semi
`double`	`[]`	`values`	`=`	`new`	`double`	`[10]`	`;`

Declaring and Initializing an Array

- You can optionally initialize the array when you declare:

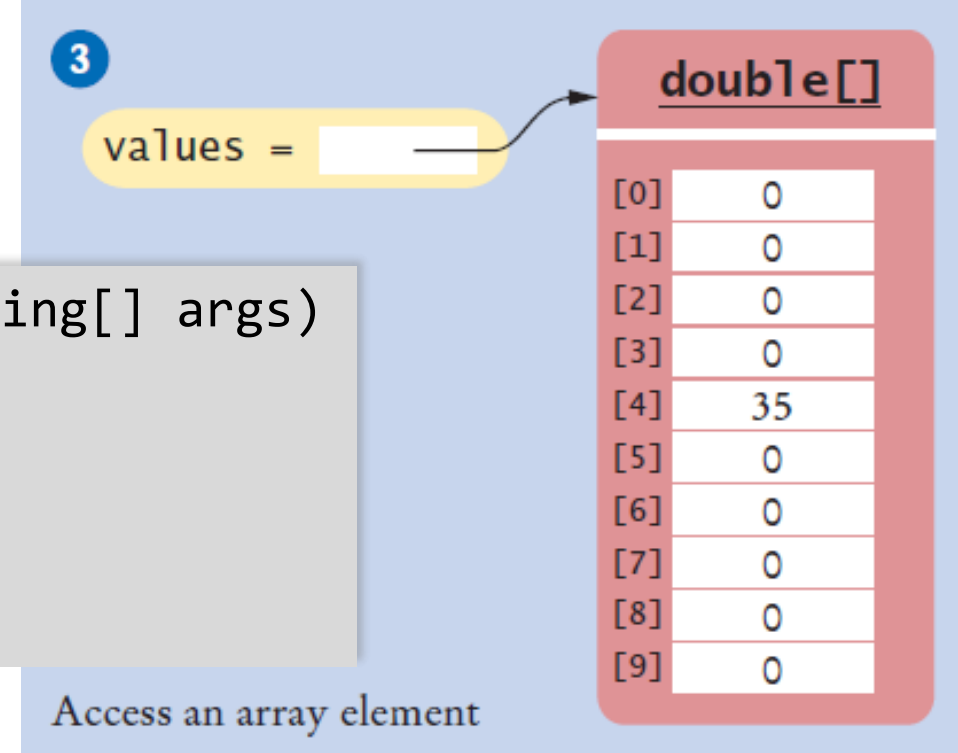
Type	Braces	Array name		contents list	semi
`int`	`[ ]`	`primes`	=	`{ 2, 3, 5, 7}`	`;`

- You are declaring that
  - There is an array named `primes`
  - The elements inside are of type `int`
  - Reserve space for four elements
    - The compiler counts them for you!
  - Set initial values to 2, 3, 5, and 7
  - Note the curly braces around the contents list

Accessing Array Elements

- Each element is numbered (called *index*)
  - Access an element by:
    - Name of the array
    - Index number

```
values[i]
```



Access an array element

```
public static void main(String[] args)
{
  double values[];
  values = new double[10];
  values[4] = 35;
}
```

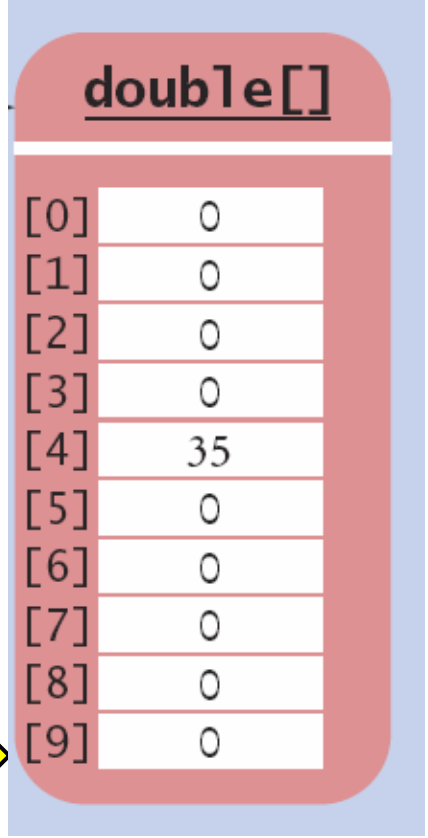
Array Index Numbers: 0 indexed

- Array index numbers start at 0
  - The rest are positive integers
- An 10 element array has indexes 0 through 9
  - There is NO element 10!

The first element is at index 0:

```
public static void main(String[] args)
{
  double values[];
  values = new double[10];
}
```

The last element is at index 9:



Array Bounds Checking

- An array knows how many elements it can hold
  - `values.length` is the size of the array named `values`
  - It is an integer value (index of the last element + 1)
- Use this to range check and prevent bounds errors

```
public static void main(String[] args)
{
  int i = 10, value = 34;
  double values[];
  values = new double[10];
  if (0 <= i && i < values.length)   // length is 10
  {
    value[i] = value;
  }
}
```

Strings and arrays use different syntax to find their length:
Strings: `name.length()`
Arrays: `values.length`

Summary: Declaring Arrays

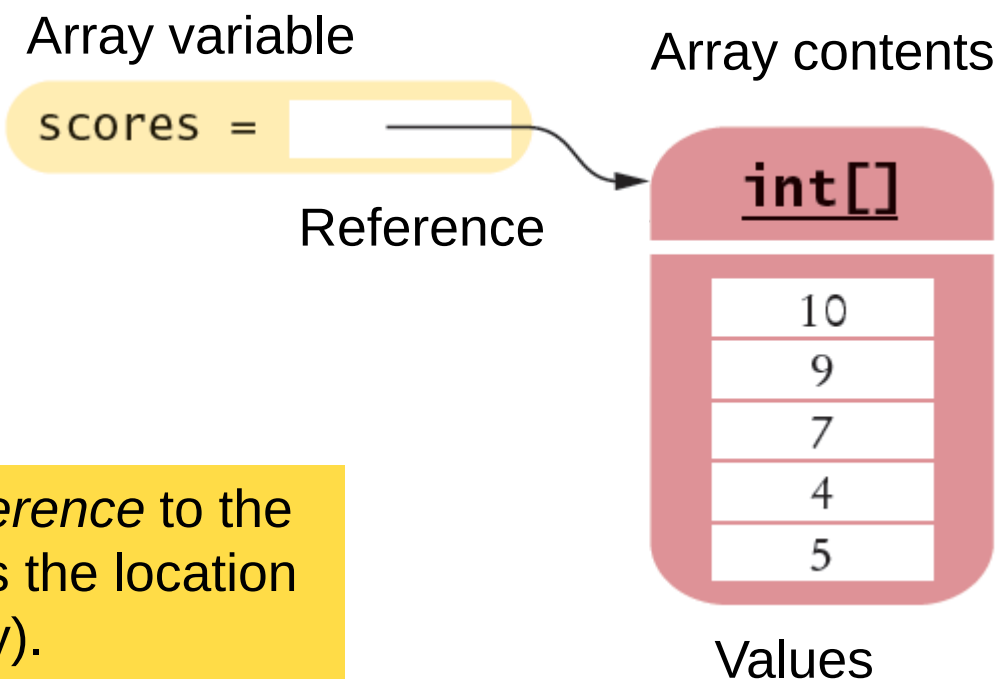
Table 1 Declaring Arrays

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

Array References

- Make sure you see the difference between the:
  - Array variable: The named 'handle' to the array
  - Array contents: Memory where the values are stored

```
int[] scores = { 10, 9, 7, 4, 5 };
```

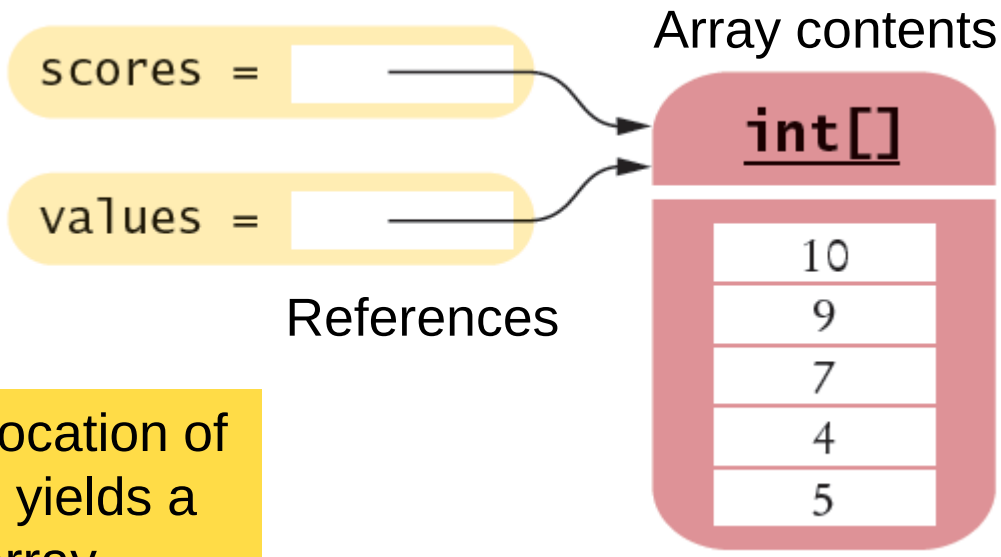


An array variable contains a *reference* to the array contents. The *reference* is the location of the array contents (in memory).

Array Aliases

- You can make one array reference refer to the same contents of another array reference:

```
int[] scores = { 10, 9, 7, 4, 5 };
int[] values = scores;  // Copying the array reference
```



An array variable specifies the location of an array. Copying the reference yields a second reference to the same array.

Partially-Filled Arrays

- An array cannot change size at run time
  - Programmer may need to guess at maximum number of elements required
  - It is a good idea to use a constant for the size chosen
  - Use a variable to track how many elements are filled

```
final int LENGTH = 100;
double[] values = new double[LENGTH];
int currentSize = 0;
Scanner in = new Scanner(System.in);
while (in.hasNextDouble())
{
  if (currentSize < values.length)
  {
    values[currentSize] = in.nextDouble();
    currentSize++;
  }
}
```

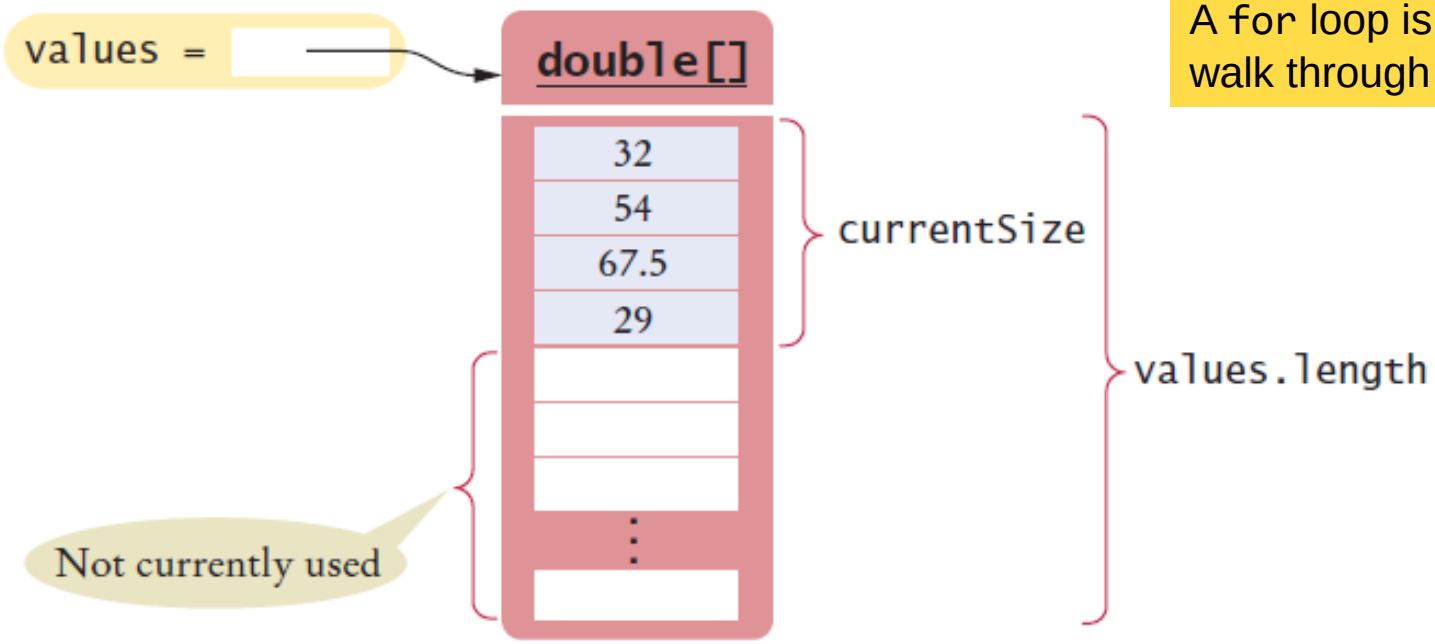
Maintain the number of elements filled using a variable (`currentSize` in this example)

Looping through a Partially Filled Array

- Use `currentSize`, not `values.length` for the last element

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

A `for` loop is a natural choice to walk through an array

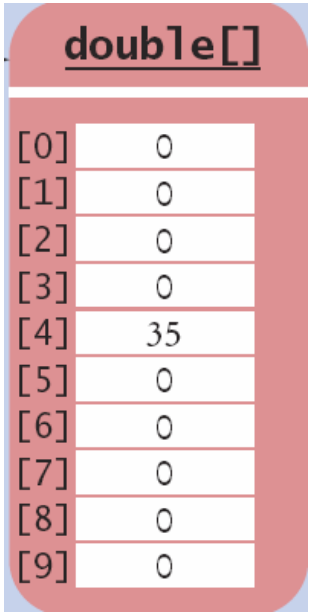


Common Error : Array Bounds Errors

- Accessing a nonexistent element is very common error
- Array indexing starts at 0
- Your program will stop at run time

```
public class OutOfBounds
{
  public static void main(String[] args)
  {
    double values[];
    values = new double[10];
    values[10] = 100;
  }
}
```

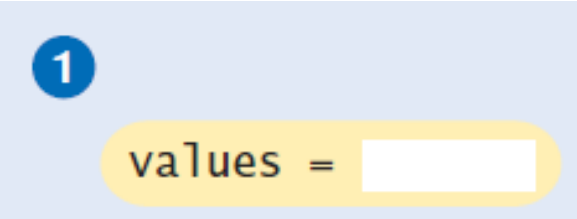
The is no element 10:




```
java.lang.ArrayIndexOutOfBoundsException: 10
        at OutOfBounds.main(OutOfBounds.java:7)
```

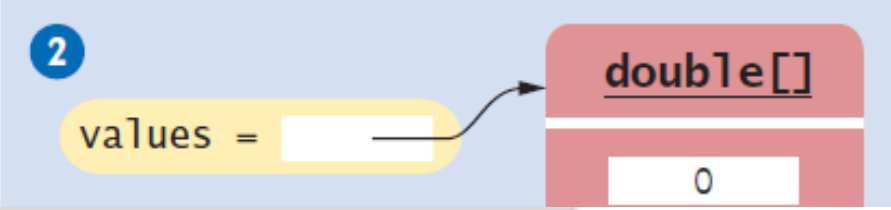
Common Error: Uninitialized Arrays

- Don't forget to initialize the array variable!
- The compiler will catch this error




```
double[] values;
...
values[0] = 29.95; // Error—values not initialized
```

```
Error: D:\Java\Unitialized.java:7:
variable values might not have been initialized
```




```
double[] values;
values = new double[10];
values[0] = 29.95; // No error
```

The for Loop

- Using for loops to 'walk' arrays is very common

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

The Enhanced for Loop

- Using for loops to 'walk' arrays is very common
  - The enhanced for loop simplifies the process
  - Also called the "for each" loop
  - Read this code as:
    - "For each element in the array"

- As the loop proceeds, it will:
  - Access each element in order (0 to length-1)
  - Copy it to the element variable
  - Execute loop body

- NOTE!!: Not possible to:
  - Change elements
  - Get bounds error

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

Common Array Algorithms

- Filling an Array
- Sum and Average Values
- Find the Maximum or Minimum
- Output Elements with Separators
- Linear Search
- Removing an Element
- Inserting an Element
- Swapping Elements
- Copying Arrays
- Reading Input

Common Algorithms: Sum and Average

- Sum and Average
  - Use enhanced `for` loop, and make sure not to divide by zero

```
double total = 0, average = 0;
for (int i = 0; i < values.length; i++nt : values)
{
  total = total + element;
}
if (values.length > 0) { average = total / values.length; }
```

Common Algorithms: Find Max or Min

- Maximum and Minimum
  - Set largest to first element
  - Use `for` or enhanced `for` loop
  - Use the same logic for minimum

NOTE: start from 1

```
double largest = values[0];
for (int i = 1; i < values.length; i++)
{
  if (values[i] > largest)
  {
    largest = values[i];
  }
}
```

Typical `for` loop to find maximum

Common Algorithms: Find Max or Min

- Maximum and Minimum
  - Set largest to first element
  - Use `for` or enhanced `for` loop
  - Use the same logic for minimum

```
double largest = values[0];
for (double element : values)
{
  if element > largest)
    largest = element;
}
```

Enhanced `for` to find maximum

```
double smallest = values[0];
for (double element : values)
{
  if element < smallest)
    smallest = element;
}
```

Enhanced `for` to find minimum

Common Algorithms: Linear Search

- Linear Search
  - Search for a specific value in an array
  - Uses a boolean found flag to stop loop if found

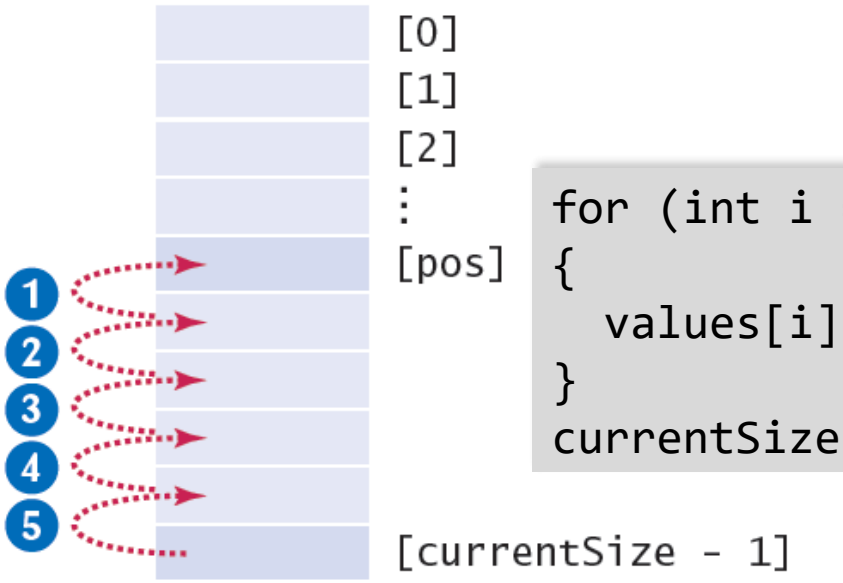
```
int searchedValue = 100;  int pos = 0;
boolean found = false;
while (pos < values.length && !found)
{
  if (values[pos] == searchedValue)
  {
    found = true;
  }
  else
  {
    pos++;
}
```

```
if (found)
{
  System.out.println("Found at position: " + pos);
}
else { System.out.println("Not found"); }
```

Compound condition to prevent bounds error if value not found.

Common Algorithms: Removing an element and maintaining order

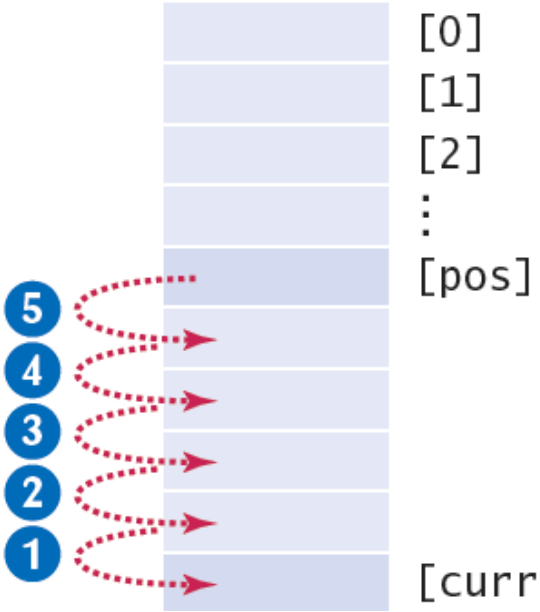
- Requires tracking the 'current size' (# of valid elements)
- But don't leave a 'hole' in the array!
- Solution depends on if you have to maintain 'order'
  - If so, move all of the valid elements after 'pos' up one spot, update .




```
for (int i = pos; i < currentSize - 1; i++)
{
  values[i] = values[i + 1];
}
currentSize--;
```

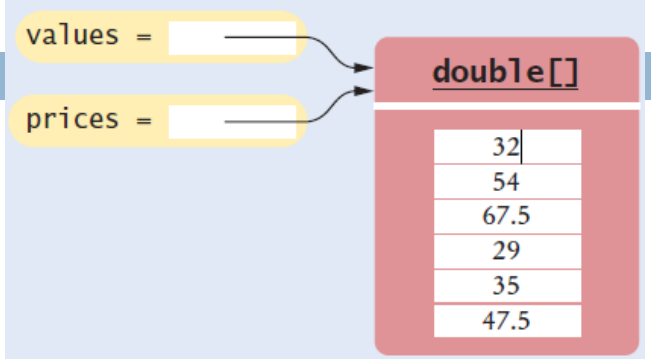
Common Algorithms: Inserting an Element

- Solution depends on if you have to maintain 'order'
  - If not, just add it to the end and update the size
  - If so, find the right spot for the new element, move all of the valid elements after 'pos' down one spot, insert the new element, and update size



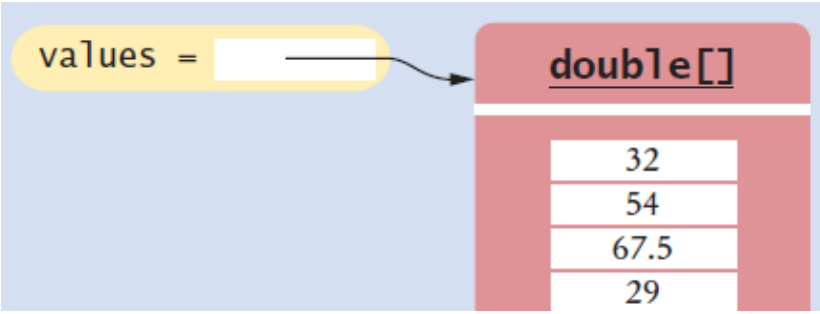

```
if (currentSize < values.length)
{
  currentSize++;
  for (int i = currentSize - 1; i > pos; i--)
  {
    values[i] = values[i - 1];  // move down
  }
  values[pos] = newElement;     // fill hole
}
```

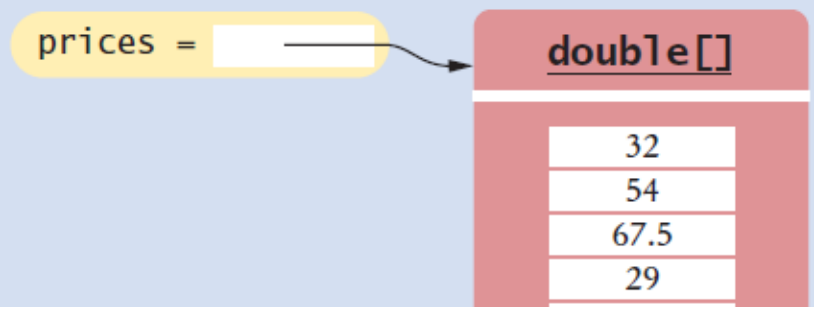
Common Algorithms: Copying Arrays



- Not the same as copying only the reference
  - Copying creates two set of contents!
- Use the new (Java 6) `Arrays.copyOf` method

```
double[] values = new double[6];
. . . // Fill array
double[] prices = values;    // Only a reference so far
double[] prices = Arrays.copyOf(values, values.length);
// copyOf creates the new copy, returns a reference
```





Common Algorithms: Growing an Array

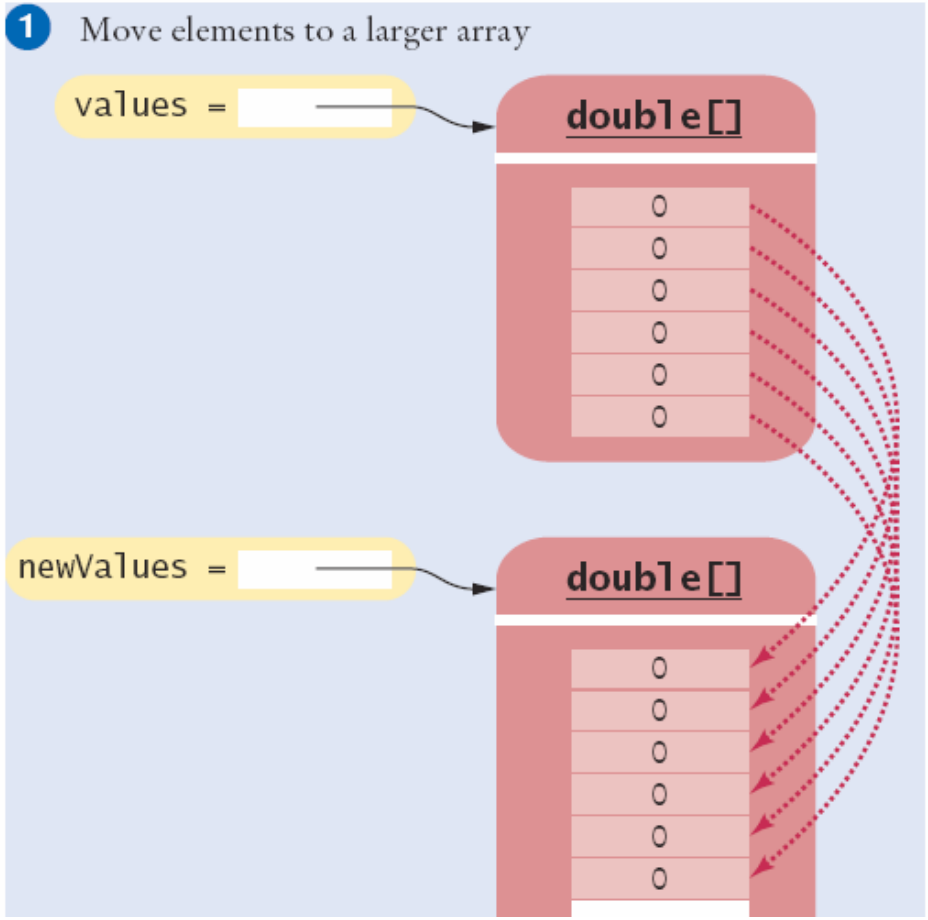
- Copy the contents of one array to a larger one
- Change the reference of the original to the larger one

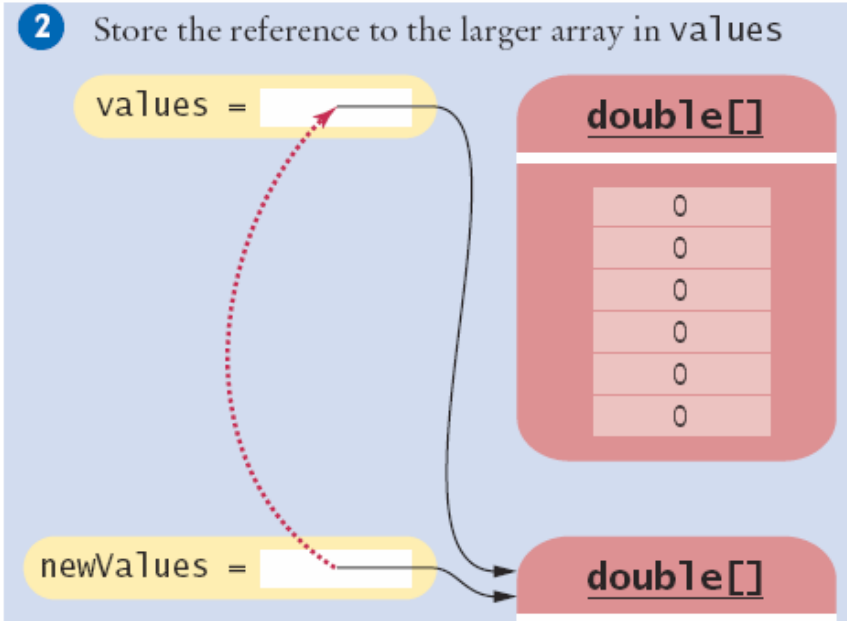
- Example: Double the size of an existing array
  - Use the `Arrays.copyOf` method
  - Use '`2 *`' in the second parameter

```
double[] values = new double[6];
. . . // Fill array
double[] newValues = Arrays.copyOf(values, 2 * values.length);
values = newValues;
```

`Arrays.copyOf` second parameter is the length of the new array

Common Algorithms: Growing an Array





- Then copy `newValues` reference over `values` reference

Common Algorithms: Reading Input

- Reading Input
  - A: ***Known*** number of values to expect
    - Make an array that size and fill it one-by-one

```
double[] inputs = new double[NUMBER_OF_INPUTS];
for (i = 0; i < values.length; i++)
{
  inputs[i] = in.nextDouble();
}
```

Common Algorithms: Reading Input

- B: ***Unknown*** number of values
  - Make maximum sized array, maintain as partially filled array

```
double[] inputs = new double[MAX_INPUTS];
int currentSize = 0;
while (in.hasNextDouble() && currentSize < inputs.length)
{
  inputs[currentSize] = in.nextDouble();
  currentSize++;
}
```

Special Topic: Sorting Arrays

- When you store values into an array, you can choose to either:
  - Keep them unsorted (random order)

[0]	[1]	[2]	[3]	[4]
11	9	17	5	12

  - Sort them (Ascending or Descending…)

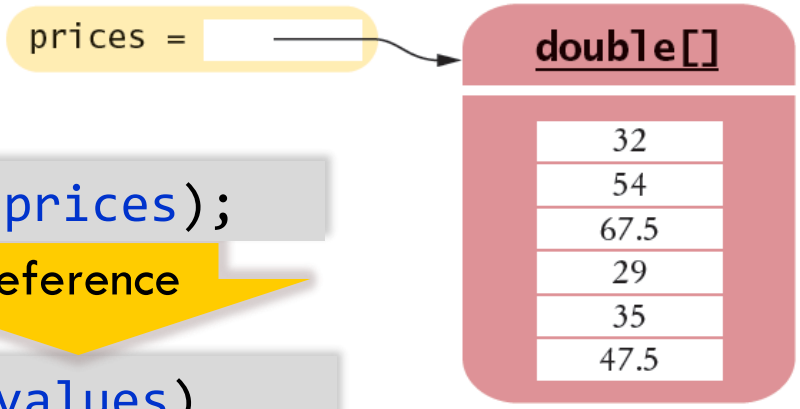
[0]	[1]	[2]	[3]	[4]
5	9	11	12	17

- A sorted array makes it much easier to find a specific value in a large data set
- The Java API provides an efficient sort method:

```
Arrays.sort(values);        // Sort all of the array
Arrays.sort(values, 0, currentSize);  // partially filled
```

Using Arrays with Methods

- Methods can be declared to receive references as parameter variables
- What if we wanted to write a method to sum all of the elements in an array?
  - Pass the array *reference* as an argument!




```
priceTotal = sum(prices);
```

reference

```
public static double sum(double[] values)
{
  double total = 0;
  for (double element : values)
    total = total + element;
  return total;
}
```

Arrays can be used as method arguments and method return values.

Passing Array References: See Reverse.java

- Passing a reference gives the called method access to all of the data elements
  - It CAN change the values!
- Example: Multiply each element in the passed array by the value passed in the second parameter

- The parameter variables values and factor are created. 1

```
multiply(values,    10);
```

reference

value

```
public static void multiply(double[] data, double factor)
{
  for (int i = 0; i < data.length; i++)
    data[i] = data[i] * factor;
}
```

Method Returning an Array

- Methods can be declared to return an array (see Reverse.java)
- To Call: Create a compatible array reference:
  - Call the method

```
int[] numbers = squares(10);
```

value

```
public static int[] squares(int n)
{
  int[] result = new int[n];
  for (int i = 0; i < n; i++)
  {
    result[i] = i * i;
  }
  return result;
}
```

reference

Two-Dimensional Arrays

- Arrays can be used to store data in two dimensions (2D) like a spreadsheet
  - Rows and Columns
  - Also known as a 'matrix'

	Gold	Silver	Bronze
Canada	1	0	1
China	1	1	0
Germany	0	0	1
Korea	1	0	0
Japan	0	1	1
Russia	0	1	1
United States	1	1	0

Figure 12 Figure Skating Medal Counts

Declaring Two-Dimensional Arrays

- Use two 'pairs' of square braces

```
const int COUNTRIES = 7;
const int MEDALS = 3;
int[][] counts = new int[COUNTRIES][MEDALS];
```

Gold	Silver	Bronze
1	0	1
1	1	0
0	0	1
1	0	0
0	1	1
0	1	1
1	1	0

- You can also initialize the array

```
const int COUNTRIES = 7;
const int MEDALS = 3;
int[][] counts =
{
  { 1, 0, 1 },
  { 1, 1, 0 },
  { 0, 0, 1 },
  { 1, 0, 0 },
  { 0, 1, 1 },
  { 0, 1, 1 },
  { 1, 1, 0 }
};
```

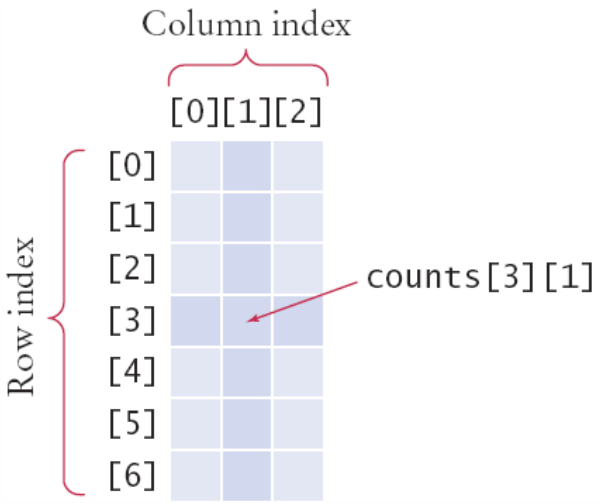
Note the use of two 'levels' of curly braces. Each row has braces with commas separating them.

Accessing Elements

- Use two index values:

Row then Column

```
int value = counts[3][1];
```



- To print
  - Use nested for loops
  - Outer row(i) , inner column(j) :

```
for (int i = 0; i < COUNTRIES; i++)
{
  // Process the ith row
  for (int j = 0; j < MEDALS; j++)
  {
    // Process the jth column in the ith row
    System.out.printf("%8d", counts[i][j]);
  }
  System.out.println(); // Start a new line at the end of the row
}
```

Locating Neighboring Elements

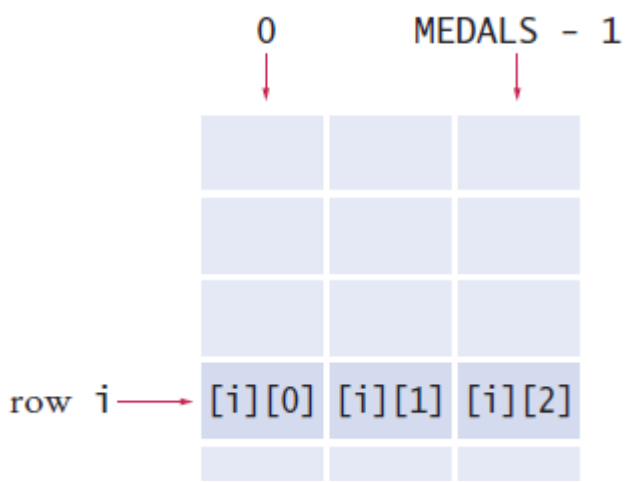
- Some programs that work with two-dimensional arrays need to locate the elements that are adjacent to an element
- This task is particularly common in games
- You are at loc `i, j`
- Watch out for edges!
  - No negative indexes!
  - Not off the 'board'

[i - 1][j - 1]	[i - 1][j]	[i - 1][j + 1]
[i][j - 1]	[i][j]	[i][j + 1]
[i + 1][j - 1]	[i + 1][j]	[i + 1][j + 1]

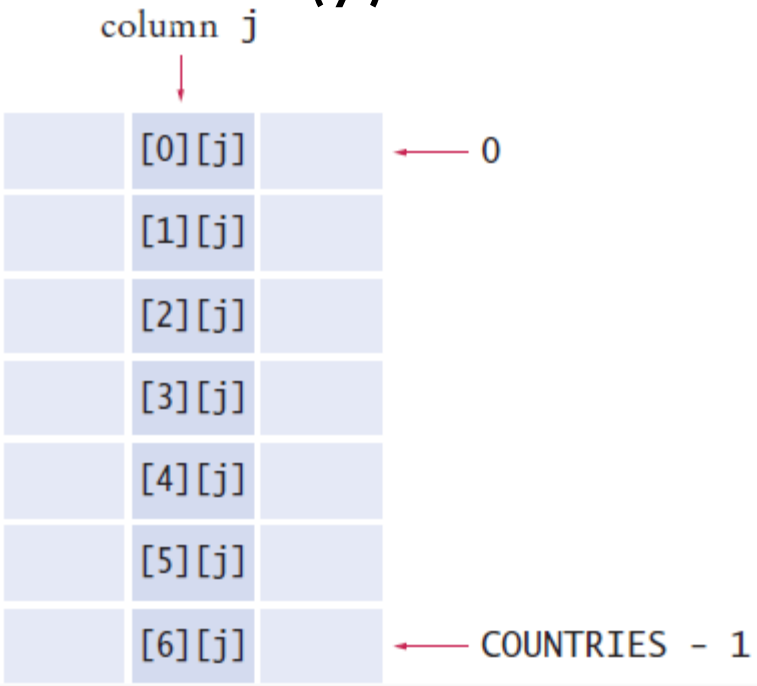
Adding Rows and Columns: Medals.java

- Rows (x)

```
int total = 0;
for (int j = 0; j < MEDALS; j++)
{
  total = total + counts[i][j];
}
```



Columns (y)




```
int total = 0;
for (int i = 0; i < COUNTRIES; i++)
{
  total = total + counts[i][j];
}
```

Array Lists

Array Lists

- When you write a program that collects values, you don't always know how many values you will have.
- In such a situation, a Java Array List offers two significant advantages:
  - Array Lists can grow and shrink as needed.
  - The `ArrayList` class supplies methods for common tasks, such as inserting and removing elements.

An Array List expands to hold as many elements as needed

Declaring and Using Array Lists

- The `ArrayList` class is part of the `java.util` package
  - It is a *generic* class
    - Designed to hold many types of objects
  - Provide the type of element during declaration
    - Inside `<` `>` as the 'type parameter':
    - The type must be a Class!!
    - Cannot be used for primitive types (`int`, `double`...)!!!

```
ArrayList<String> names = new ArrayList<String>();
```

```
ArrayList<Integer> nums = new ArrayList<Integer>();
```

ArrayLists

- `ArrayList` provides many useful methods:
  - `add`: add an element
  - `get`: return an element
  - `remove`: delete an element

Array Lists

Variable type

Variable name

An array list object of size 0

```
ArrayList<String> friends = new ArrayList<String>();
```

```
friends.add("Cindy");
String name = friends.get(i);
friends.set(i, "Harry");
```

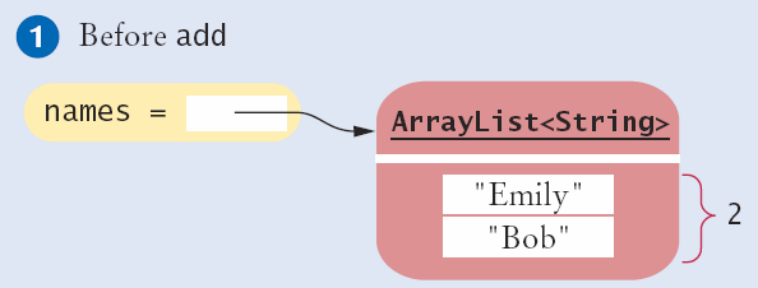
The add method appends an element to the array list, increasing its size.

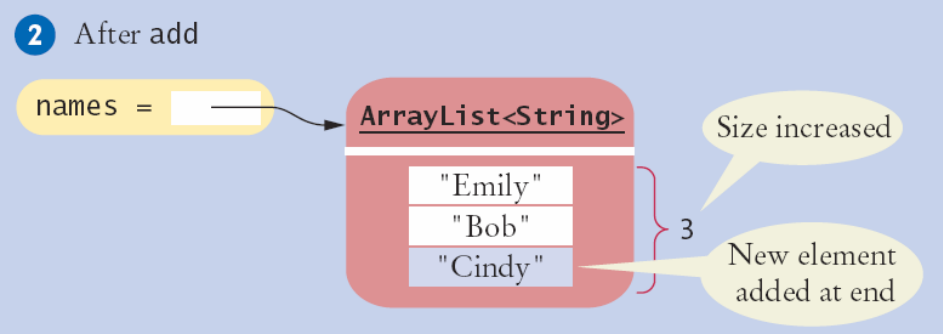
Use the get and set methods to access an element.

The index must be ≥ 0 and < friends.size().

- `set`: change an element
- `size`: current length

Adding an element with add( )





- The add method has two versions:
  - Pass a new element to add to the end

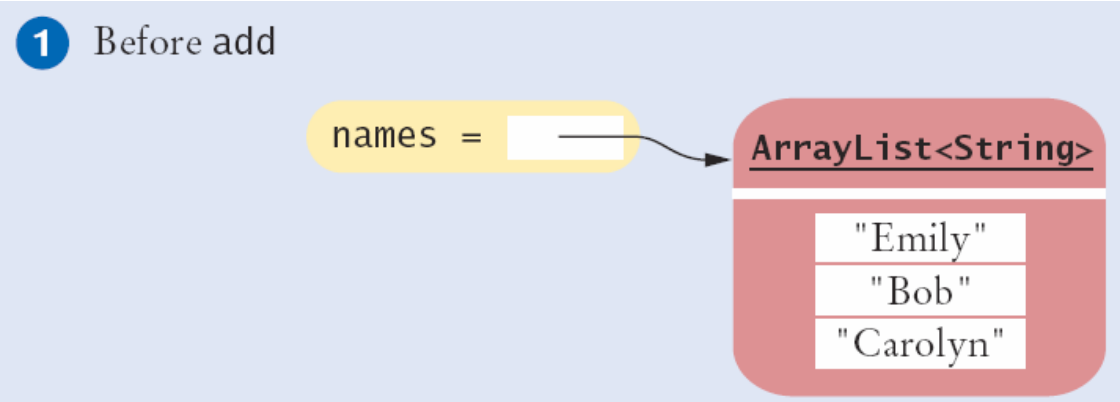
```
names.add("Cindy");
```

  - Pass a location (index) and the new value to add

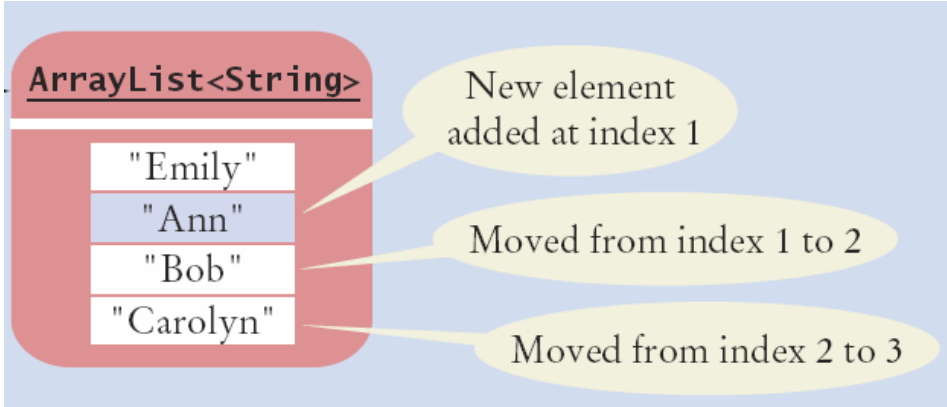
Moves all other elements

```
names.add(1, "Cindy");
```

Adding an Element in the Middle



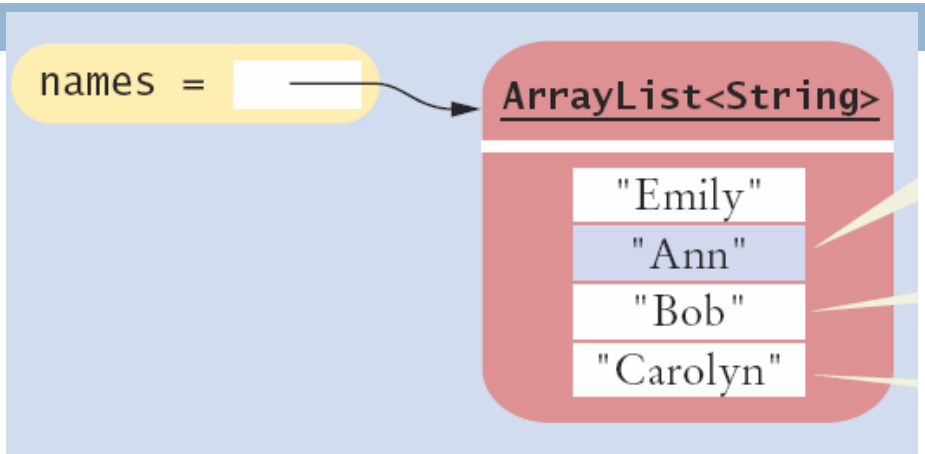

```
names.add(1, “Ann”);
```



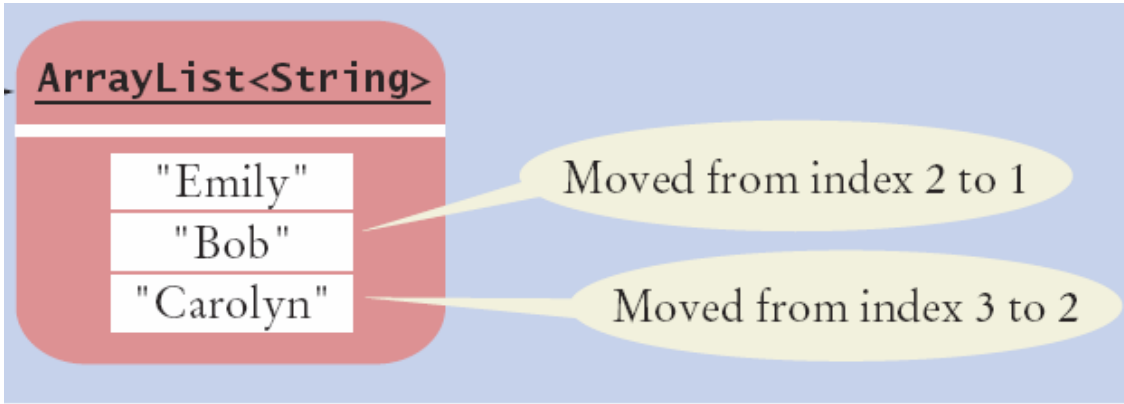
- Pass a location (index) and the new value to add

  Moves all other elements

Removing an Element




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



- Pass a location (index) to be removed
  Moves all other elements

Using Loops with Array Lists

- You can use the enhanced for loop with Array Lists:

```
ArrayList<String> names = . . . ;
for (String name : names)
{
  System.out.println(name);
}
```

- Or ordinary loops:

```
ArrayList<String> names = . . . ;
for (int i = 0; i < names.size(); i++)
{
  String name = names.get(i);
  System.out.println(name);
}
```

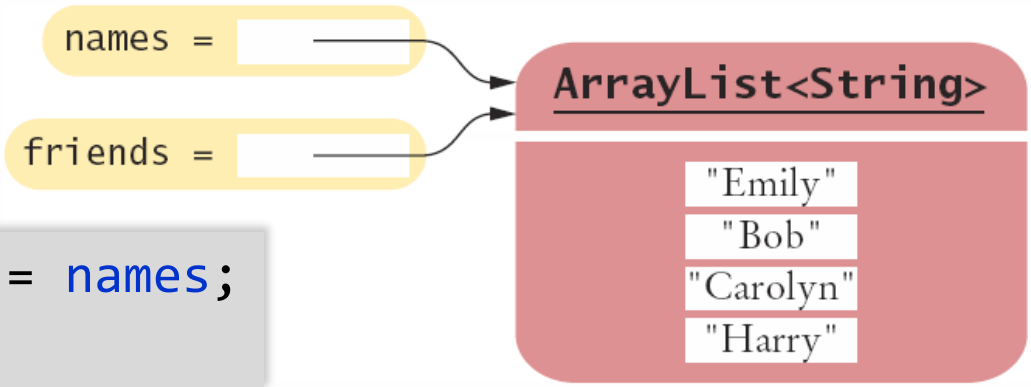
Working with Array Lists

Table 2 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]`.
`String name = names.get(i);`	Gets an element.
`String last = names.get(names.size() - 1);`	Gets the last element.

Copying an ArrayList

- Remember that `ArrayList` variables hold a ***reference*** to an `ArrayList` (just like arrays)
- Copying a reference:




```
ArrayList<String> friends = names;
friends.add("Harry");
```

- To make a true copy, pass the reference of the original `ArrayList` to the constructor of the new one:

reference

```
ArrayList<String> newNames = new ArrayList<String>(names);
```

Array Lists and Methods

- Like arrays, Array Lists can be method parameter variables and return values.
- Here is an example: a method that receives a list of `String`s and returns the reversed list.

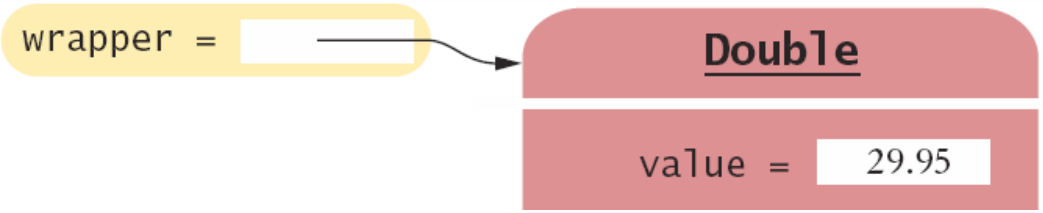
reference

```
public static ArrayList<String> reverse(int xist<String> names)
{
  // Allocate a list to hold the method result
  ArrayList<String> result = new ArrayList<String>();
  // Traverse the names list in reverse order (last to first)
  for (int i = names.size() - 1; i >= 0; i--)
  {
    // Add each name to the result
    result.add(names.get(i));
  }
  return result;
}
```

Wrappers and Auto-boxing

- Java provides *wrapper* classes for primitive types
  - Conversions are automatic using ***auto-boxing***
    - Primitive to wrapper Class

```
double x = 29.95;
Double wrapper;
wrapper = x;   // boxing
```



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

    - Wrapper Class to primitive

```
double x;
Double wrapper = 29.95;
x = wrapper;   // unboxing
```

Wrappers and Auto-boxing

- You cannot use primitive types in an `ArrayList`, but you can use their wrapper classes
  - Depend on auto-boxing for conversion

- Declare the `ArrayList` with wrapper classes for primitive types
  - Use `ArrayList<Double>`
    - Add primitive double variables
    - Or double values

```
double x = 19.95;
ArrayList<Double> values = new ArrayList<Double>();
values.add(new Double(29.95));                // boxing
values.add(new Double(x));
                // boxing
double x = values.get(0);     // unboxing
```

Choosing Arrays or Array Lists

- Use an Array if:
  - The size of the array never changes
  - You have a long list of primitive values
    - For efficiency reasons
  - Your instructor wants you to

- Use an Array List:
  - For just about all other cases
  - Especially if you have an unknown number of input values

Array and Array List Operations

Table 3 Comparing Array and Array List Operations

Operation	Arrays	Array Lists
Get an element.	`x = values[4];`	`x = values.get(4)`
Replace an element.	`values[4] = 35;`	`values.set(4, 35);`
Number of elements.	`values.length`	`values.size()`
Number of filled elements.	`currentSize` (companion variable, see Section 6.1.3)	`values.size()`
Remove an element.	See Section 6.3.6	`values.remove(4);`
Add an element, growing the collection.	See Section 6.3.7	`values.add(35);`
Initializing a collection.	`int[] values = { 1, 4, 9 };`	No initializer list syntax; call `add` three times.

Common Error

- Length versus Size
  - Unfortunately, the Java syntax for determining the number of elements in an array, an `ArrayList`, and a `String` is not consistent.
  - It is a common error to confuse these. You just have to remember the correct syntax for each data type.

Data Type	Number of Elements
Array	`a.length`
Array list	`a.size()`
`String`	`a.length()`