

# Introduction to Software Development (ISD)

## Week 4

Autumn term 2012

# Aims of Week 4

- To become familiar with using Arrays and Array Lists to collect and process lists of values
- To use the Enhanced for loop for traversing arrays and array lists
- To learn about common algorithms for processing arrays and array lists
- To learn about how to use arrays and array lists with methods

# Arrays

- Programs often need to store a list of values and then process them
- For example, if you had this list of values of type `double`, how many variables would you need?
  - `double data1, data2, data3, . . . , data10`
- Instead, an **array** variable can be declared:  
`double[] data;`
- and it can then be initialised to hold these 10 values:  
`data = {32, 54, 67.5, 29, 35, 80, 115,  
44.5,  
100, 65}`
- Note that an array always holds a list of values of the same type (in this case, the type is `double`)

32
54
67.5
29
35
80
115
44.5
100
65

# Declaring and Initialising Arrays

- We can combine the above declaration and initialisation statements into one statement (as we have already seen for declaring and initialising other types of variables):

```
double[] data = {32, 54, 67.5, 29, 35, 80, 115,  
                 44.5, 100, 65}
```

- If we don't know the initial contents, we can declare an array of numbers and then initialise it to be of a specified length (10 below), in which case it will have an initial content of 0 in every entry:

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

- Or, combining these two statements into one:

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

# Declaring and Initialising Arrays

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

- initialises each array entry to the value “null”

```
char[] data = new char[20];
```

- initialises each array entry to the character with Unicode value 0

# Declaring and Initialising Arrays

```
1) double[] data;
```

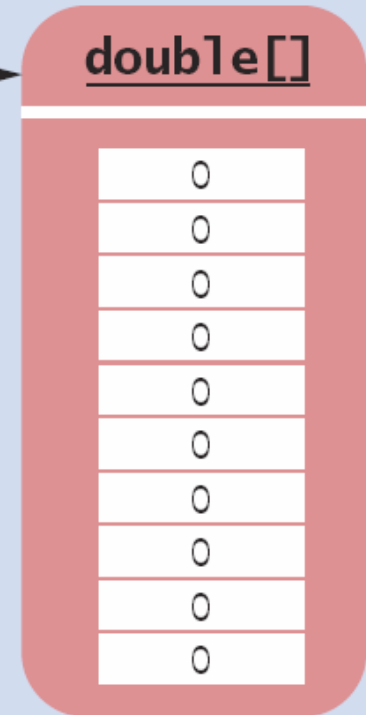
```
2) data = new double[10];
```

1 data =

Declare the array variable


2 data =

Initialize it with an array



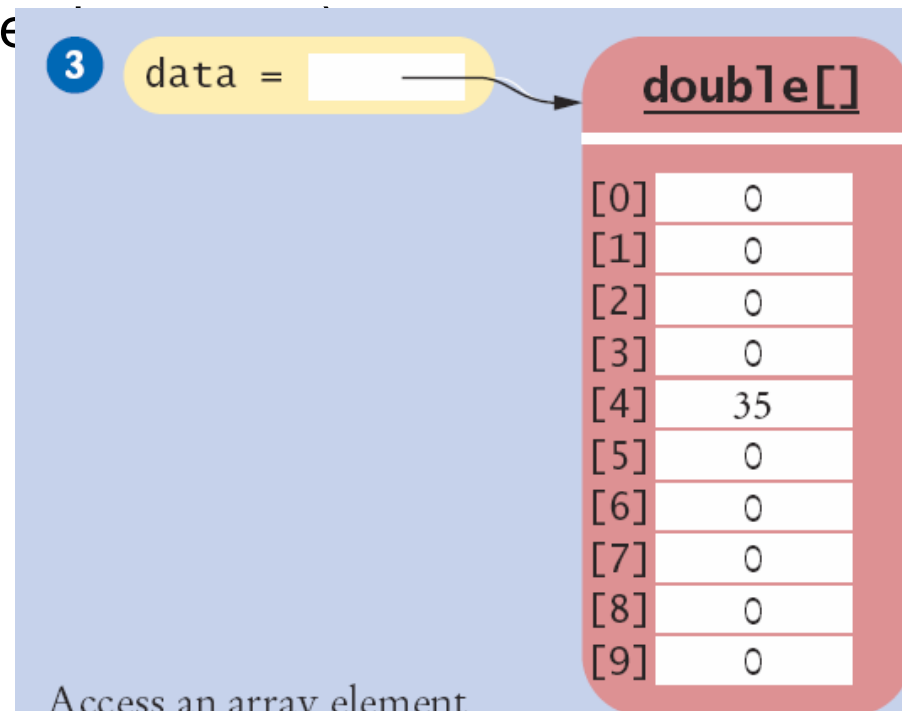
# Declaring and Initialising Arrays – summary

Table 1 Declaring Arrays

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

# Accessing array elements

- Each element is numbered starting from 0 – this is its **index**
- An element is accessed by stating the name of the array and the index number e.g. `data[4]` is the fifth element of the array `data`
- N.B. you can't access an element of an array unless it has been initialised (compiled)
- An element of an array can be used just like any other variable of that type e.g.
  - in an assignment statement:  
`data[4] = 35;`
  - or in an expression:



Access an array element

`System.out.println(data[4]);`



# Bounds errors

- An array index must be at least 0 and less than the size of the array
- Be careful not to try to access array elements with an index value outside this range. For example, with the previous array, attempting to use `data[10]` causes a run-time error
- `xxx.length` is the size of the array named `xxx`
- it is equal to the index of the last element of `xxx` plus 1
- The following code ensures that the value of the variable `i` is valid before attempting to access the `ith` element of the array `xxx` :

```
if (0 <= i && i < xxx.length) {  
    // process xxx[i]  
}
```

## Visiting all elements of an array:

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

So, for example:

```
int[] monthDays =  
    {31,28,31,30,31,30,31,31,30,31,30,31};  
  
for (int i = 0; i < monthDays.length; i++) {  
    System.out.printf("Month %d has %d days\n",  
                      i,monthDays[i]);  
}
```

# Array Variables

- An array variable contains a *reference* to the array contents
- The reference is the memory location of the array contents

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

Array variable

scores =



Reference

Array contents

int[]

10

9

7

4

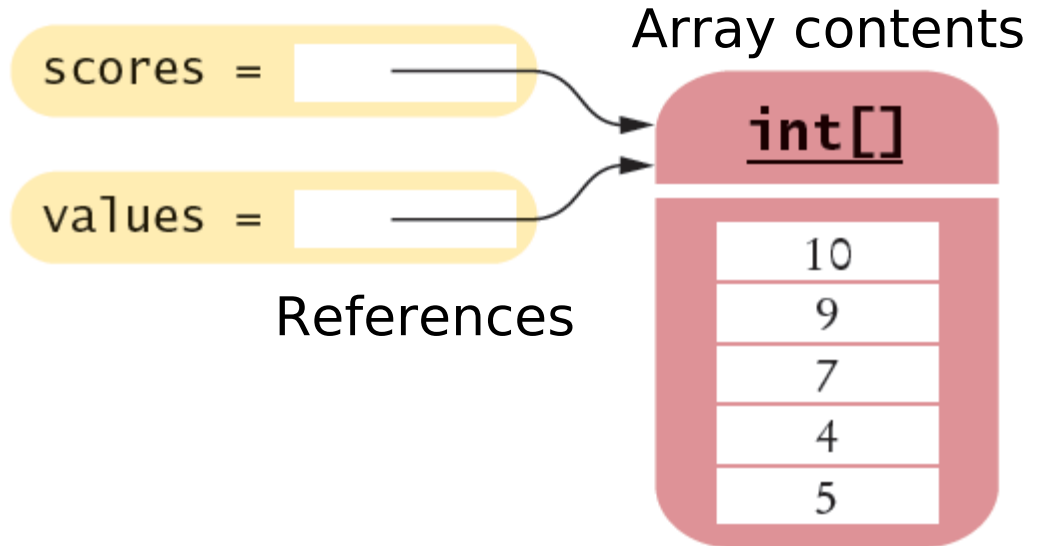
5

# Copying references to arrays

- You can make one array variable refer to the same contents as another:

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

```
int[] values = scores; // Copies the array  
reference
```



# Copying the *contents* of an array

- The method `Arrays.copyOf` creates a new array and returns its reference
- The value of a new array variable (e.g. `scores1` below) can be set to this reference:

```
int[] scores = {56, 75, 35, 81, 44};  
... // process the scores array  
int[] scores1 =  
    Arrays.copyOf(scores, scores.length);
```

- You need to import the `Arrays` class from the `java.util` package in order to use its methods:

```
import java.util.Arrays;
```

# Growing an array

- If you need to create a bigger array than the original one, you can specify the required size of the new array in the second parameter to `Arrays.copyOf`:

```
int newLen = 10;  
int[] scores2 = Arrays.copyOf(scores, newLen);
```

- `scores2` now has contents:

56, 75, 35, 81, 44, 0, 0, 0, 0, 0

# Growing an array

- If the specified size of the new array, N, is smaller than the array that is being copied, then only the first N elements are copied into the new array e.g.

```
int[] scores3 = Arrays.copyOf(scores, 4);
```

- means that scores3 contains just the first four numbers from scores i.e.

56, 75, 35, 81

# Printing out arrays

- `Arrays.toString()` can be used to print out an array e.g.  
`System.out.println(Arrays.toString(scores));`
- This will print out the array in this form:  
`[56, 75, 35, 81, 44]`
- For printing it out formatted differently, use a “for” loop e.g.:

```
for (int i = 0; i < scores.length; i++) {  
    if (i > 0) {  
        System.out.print(" | ");  
    }  
    System.out.print(scores[i]);  
}
```
- outputs: 56 | 75 | 35 | 81 | 44



# Reading user input into arrays

- If we know how many input values are required (e.g. as specified by the value of `NUMBER_OF_INPUTS` in the example below), we can initialise an array of that size and set its elements one-by-one according to the user's input:

```
final int NUMBER_OF_INPUTS = 20;

System.out.printf("Please input %d
    numbers:\n", NUMBER_OF_INPUTS);

double[] inputs = new
    double[NUMBER_OF_INPUTS];

for (int i = 0; i < inputs.length; i++) {
    inputs[i] = in.nextDouble();
}
```

# Reading user input into arrays

- If we don't know how many input values are required, we can initialise an array of the *maximum possible size*, fill its elements one-by-one, but also keep a count `currentSize` of the number of values that were input:

```
final int MAX_INPUTS = 200;

System.out.println("Please input the numbers:");

double[] inputs = new double[MAX_INPUTS];
int currentSize = 0;

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

# Searching through an array to find a specific value:

```
int searchedForValue = 57;
int pos = 0;
boolean found = false;
while (pos < data.length && !found) {
    if (data[pos] == searchedForValue) {
        found = true;
    }
    else {
        pos++;
    }
}
if (found) {
    System.out.println("Found at position: " + pos);
}
else {
    System.out.println("Not found");
}
```

# The “enhanced for” loop

- Often we need to visit all elements of an array
- The “enhanced for” loop makes this simpler than using an ordinary for loop. E.g. to find the sum of an array of numbers:

```
double[] data = {32, 54, 67.5, 29, 35};  
double sum = 0;  
for (double element : data) {  
    sum = sum + element;  
}
```

- You cannot get a bounds error using the “enhanced for” loop
- But, unlike with an ordinary “for” loop, the “enhanced for” loop does not allow you to modify the contents of the array

# Other common array algorithms

- Filling arrays
- Calculating the sum, average, maximum, minimum of an array of numbers
- Inserting and removing elements
- Swapping elements:
  - Read Section 6.3 of *Java for Everyone*

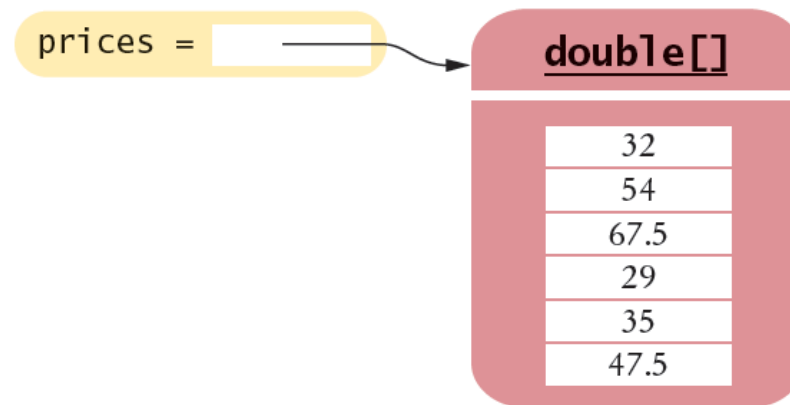
# Using Arrays with Methods

- Arrays can be used as method parameters in the same way as other types of values e.g. the following method computes the sum of an array of numbers:

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

# Using Arrays with Methods

- Suppose `prices` is an array variable of type `double[]`



- We can find out the total of its contents by calling the `sum` method:

```
double total = sum(prices);
```

- Notice that it is the **reference** to the array that is passed as the parameter, *not* a copy of the array contents!

# Updating array contents using methods

- Passing the array reference allows the called method to access the array's elements. So it can change their values!
- Example: multiply each element in the passed array by the value passed in the second parameter:

```
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;
    }
}
```



**1**values = data = factor = double[]

32

54

67.5

29

35

Method call

**2**values = data = factor = double[]

32

54

67.5

29

35

Initializing method parameters

- The parameter variables data and factor are created. **1**
- The parameter variables are initialized with the values that are passed in the call. In our case, data is set to values and factor is set to 10. Note that values and data are references to the *same* array. **2**

3

values =

---

data =

factor =

double[]

320

540

675

290

350

About to return to the caller

4

values =

double[]

320

540

675

290

350

After method call

- The method multiplies all array elements by 10. 3
- The method returns. Its parameter variables are removed. However, values still refers to the array with the modified values. 4

# Methods can return arrays

- For example, this method constructs and returns an array of  $n$  square numbers,  $0^2$  up to  $(n-1)^2$ :

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

# Two-Dimensional Arrays

- Arrays can also be used to store data in two dimensions (2D), which can be visualised as *rows* and *columns*, similarly to matrices in maths. E.g. to store this data about the number of medals won by each country:

<u>Countries</u>	<u>Gold</u>	<u>Silver</u>	<u>Bronze</u>
Canada	0	0	1
China	0	1	1
Japan	1	0	0
Russia	3	0	1
Switzerland	0	1	0
Ukraine	0	0	1
United States	0	2	0

```
final int COUNTRIES = 7;
```

```
final int MEDALS = 3;
```

```
String[] countries = {"Canada", "China", "Japan",  
    "Russia",  
    "Switzerland", "Ukraine", "United  
    States"};
```

```
int[][] counts =
```

```
{
```

```
    { 0, 0, 1 },
```

```
    { 0, 1, 1 },
```

```
    { 1, 0, 0 },
```

```
    { 3, 0, 1 },
```

```
    { 0, 1, 0 },
```

```
    { 0, 0, 1 },
```

```
    { 0, 2, 0 }
```

```
};
```

# Two-Dimensional Arrays

- Notice that a 2D array is actually an array of arrays:

```
int[][] counts =  
{  
    { 0, 0, 1 },  
    { 0, 1, 1 },  
    { 1, 0, 0 },  
    { 3, 0, 1 },  
    { 0, 1, 0 },  
    { 0, 0, 1 },  
    { 0, 2, 0 }  
};
```

- So `counts.length` returns 7, and `counts[0].length` returns 4

# Two-Dimensional Arrays

- If we didn't know the initial contents, we could have initialised the two dimensional array with the default content of 0 in each element:

```
int[][] counts = new int[COUNTRIES]  
[MEDALS];
```

- We can access a specific element by specifying the two index values, specifying first the row and then the column:

```
int chinaBronzes = counts[1][2];
```

(don't forget that both the rows and the columns are numbered starting from 0!)

# Printing out 2D arrays

- Use nested for loops
- Outer loop processes rows (*i*) , inner loop processes columns (*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 end of  
    the row  
}
```



# Array Lists

- In contrast to arrays, **array lists** can grow and shrink as needed
- The `ArrayList` class (also in the `java.util` package) supplies methods for common tasks such as inserting and removing elements in array lists
- You need to use this import statement in order to use array lists in your program:

```
import java.util.ArrayList;
```

- An array list of strings, called “names”, is declared as follows, for example:

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

# Array Lists

- The `ArrayList` class is an example of a **generic class** in Java
- Generic classes are designed to handle many types of objects
- You need to specify the type of the objects when declaring a variable of that class:
  - write the type inside `< >` - this is the 'type parameter' which is passed to the generic class
  - the type must be a Class
  - it cannot be a primitive type (int, double...)

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

# Array Lists

- The ArrayList class provides many useful methods:
  - add: add an element
  - get: return an element
  - remove: delete an element
  - set: change an element
  - size: return current length of the array list
- When an array list is first constructed, it has size 0:

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

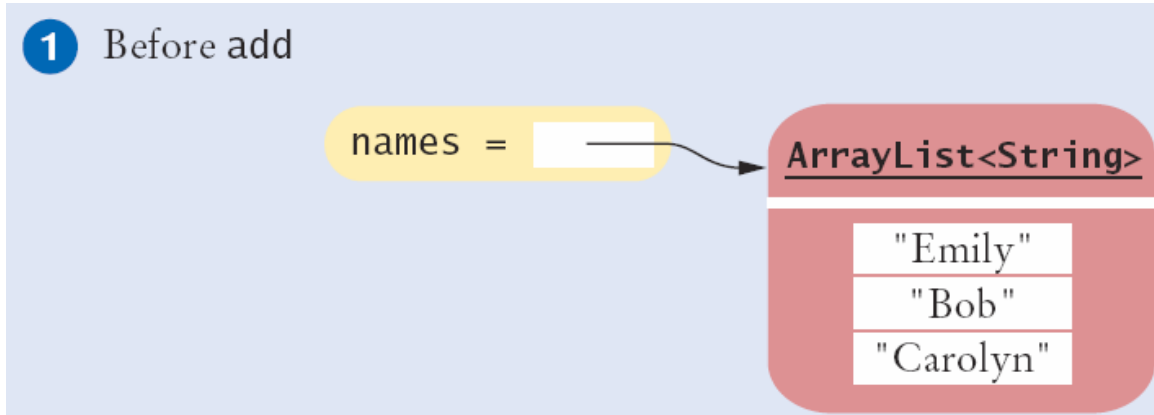
```
names.add("Emily"); // now has size 1
```

```
names.add("Bob"); // now has size 2, "Bob" is  
second
```

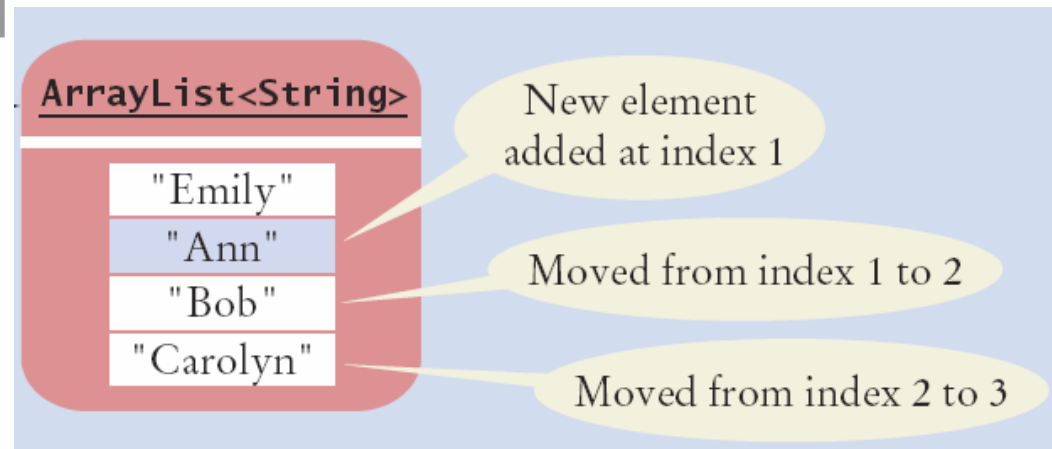
```
names.add("Carolyn"); // now size 3, "Carolyn" is  
third
```

# Adding an element in the middle

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

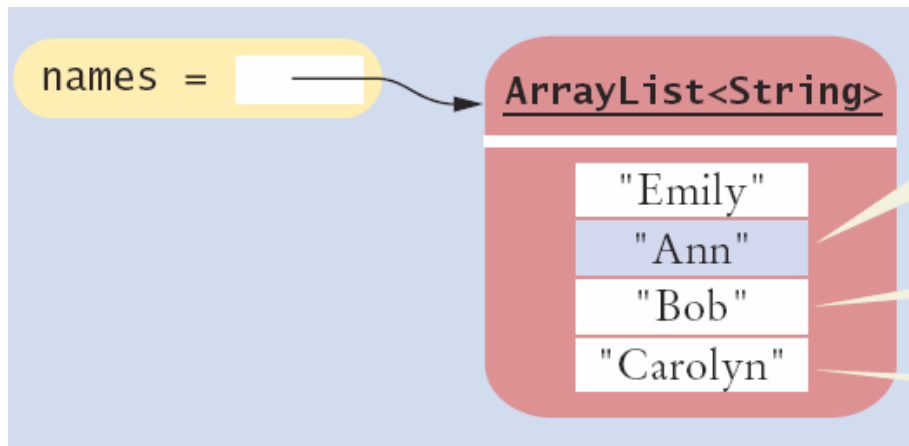


```
names.add(1, "Ann");
```

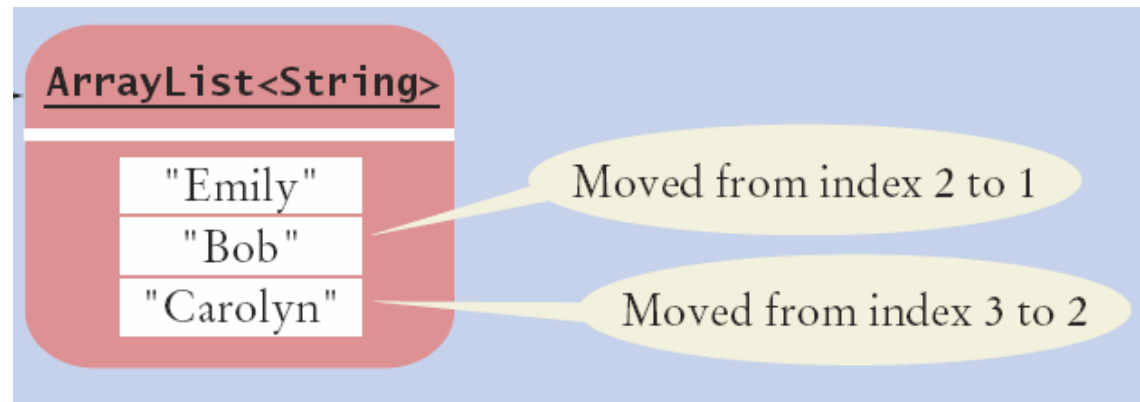


# Removing an element

- Pass the location (index) of the element to be removed:



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



**Table 2** Working with Array Lists

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

## Note on “length” versus “size”

- Unfortunately, the Java syntax for determining the number of elements in an array, an array list, and a string is not consistent!
- You just have to remember the correct syntax for each data type:

Data Type	Number of Elements
Array	<code>a.length</code>
Array list	<code>a.size()</code>
String	<code>a.length()</code>

# Using the “enhanced for” loop with Array Lists

- The enhanced for loop can be used for visiting all elements of an array list e.g.

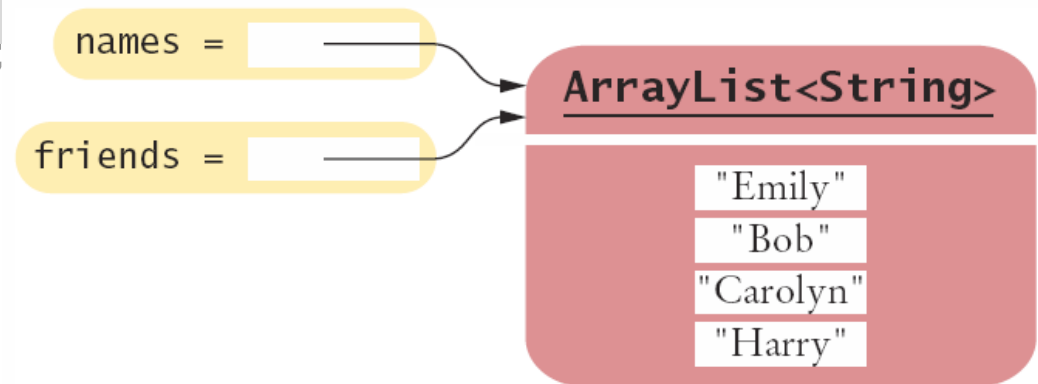
```
ArrayList<String> names = . . .  
for (String name : names) {  
    System.out.println(name);  
}
```



# Copying an ArrayList

- ArrayList variables hold a *reference* to an array list (just like arrays)
- Assigning the contents of one array variable to another just

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



- To make a new copy of the contents of an array list, pass the **reference** of the original ArrayList to the constructor of a new one:

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

# Array Lists and Methods

- Like arrays, Array Lists can be method parameters or method return values
- Here is an example: a method, `rev`, that receives an array list of Strings and returns the reversed list

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

# Wrapper Classes and Auto-boxing

- Can we store primitive values in array lists?
- Yes:
  - Java provides **wrapper** classes for the primitive types
  - Conversions between a primitive type and its wrapper class are automatic (no need for explicit 'casting' in programs)
  - This automatic conversion from a primitive type to its wrapper class is called **auto-boxing**

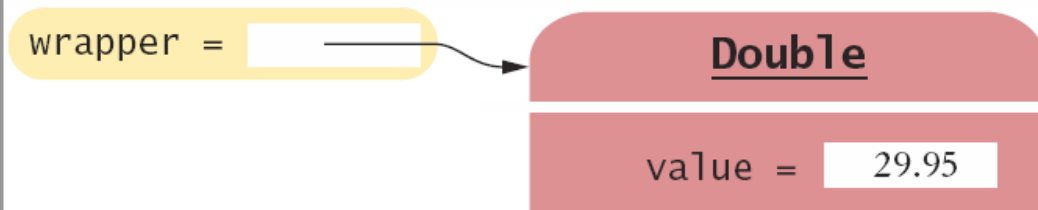
Primitive Type	Wrapper Class
byte	Byte
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

# Wrapper Classes and Auto-boxing

- Converting a value of a primitive type to a value of the Wrapper Class:

```
double value = 29.95;  
Double wrapper;
```

```
wrapper = value; //  
boxing
```



- Converting a value of the Wrapper Class to a value of the

```
Double wrapper = 29.95;  
double value;
```

```
value = wrapper; //  
unboxing
```

# Wrapper Classes and Auto-boxing

- You cannot use primitive types in an array list, but you *can* use the *wrapper classes* corresponding to the primitive types
  - auto-boxing does the necessary type conversions
- Declare the array list with the wrapper class corresponding to the required primitive type
- E.g. with an array list of type `ArrayList<Double>`
- we can add to it variables or values of the primitive type `double`

```
ArrayList<Double> data = new ArrayList<Double>();  
  
data.add(29.95);           // boxing  
  
double x = 19.95;  
data.add(x);               // boxing  
  
double y = data.get(1);    // unboxing
```

# ArrayList Algorithms are similar to Arrays

- Converting implementations that use Arrays to implementations that use ArrayLists requires the following changes:

- instead of `[i]` use the method `get()`
- instead of `data.length` use the method `size()`

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

```
double largest = data.get(0);
for (int i = 1; i < data.size(); i++) {
    if (data.get(i) > largest) {
        largest = data.get(i);
    }
}
```

# When to use Arrays or Array Lists

- Use an Array if:
  - The size of the array never changes
  - You have a long list of primitive values (in which case arrays are more efficient than array lists, since they don't require autoboxing)
- Use an ArrayList:
  - For just about all other cases
  - And especially if you have an unknown number of input values

# Aims of Week 4

- To become familiar with using Arrays and Array Lists to collect and process lists of values
- To use the “enhanced for” loop for traversing arrays and array lists
- To learn about common algorithms for processing arrays and array lists
- To learn about how to use arrays and array lists with methods