Arrays

Lecture 7a

Topics (1 of 2)

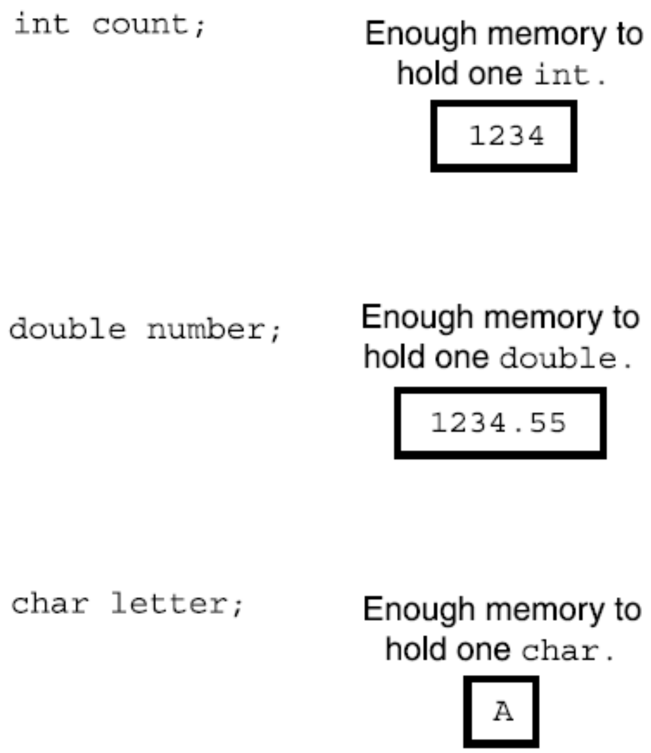
- Introduction to Arrays
- Processing Array Contents
- Passing Arrays as Arguments to Methods
- Some Useful Array Algorithms and Operations
- Returning Arrays from Methods
- String Arrays
- Arrays of Objects

Topics (2 of 2)

- Two-Dimensional Arrays
- Arrays with Three or More Dimensions
- Command-Line Arguments
- ArrayList

Introduction to Arrays (1 of 2)

- Primitive variables are designed to hold only one value at a time.


int count;
Enough memory to hold one int.
1234
double number;
Enough memory to hold one double.
1234.55
char letter;
Enough memory to hold one char.
A

Introduction to Arrays (2 of 2)

- Arrays allow us to create a collection of like values that are indexed.
- An array can store any type of data but only one type of data at a time.
- An array is a list of data elements.

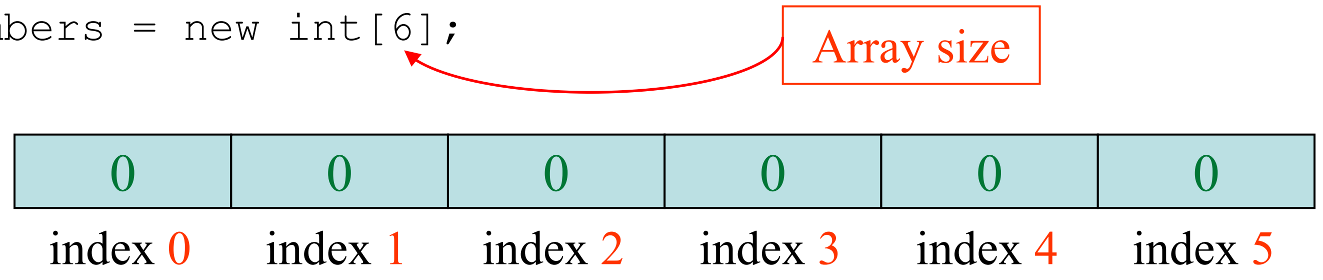
Creating Arrays (1 of 3)

- An array is an object, so it needs an object reference.

```
// Declare a reference to an array that will hold integers.
int[] numbers;
```

- The next step creates the array and assigns its address to the `numbers` variable.

```
// Create a new array that will hold 6 integers.
numbers = new int[6];
```



Array element values are initialized to 0.

Array indexes always start at 0.

Creating Arrays (2 of 3)

- It is possible to declare an array reference and create it in the same statement.

```
int[] numbers = new int[6];
```

- Arrays may be of any type.

```
float[] temperatures = new float[100];
char[] letters = new char[41];
long[] units = new long[50];
double[] sizes = new double[1200];
```

Creating Arrays (3 of 3)

- The array size must be a non-negative number.
- It may be a literal value, a constant, or variable.

```
final int ARRAY_SIZE = 6;
int[] numbers = new int[ARRAY_SIZE];
```

- Once created, an array size is fixed and cannot be changed.

Accessing the Elements of an Array

20	0	0	30	0	0
numbers[0]	numbers[1]	numbers[2]	numbers[3]	numbers[4]	numbers[5]

- An array is accessed by:
  - the reference name
  - a subscript that identifies which element in the array to access.

```
numbers[0] = 20;
numbers[3] = 30;
```

Review of Instance Fields and Methods (1 of 2)

- Point to ponder #1:

  What is the difference here?

```
int[] numbers = new int[4];
```

Declaring an array of size 4

```
numbers[4] = 10;
```

Assigning 10 to the 5th element of the array

Inputting and Outputting Array Elements

- Array elements can be treated as any other variable.
- They are simply accessed by the same name and a subscript.
- See example: ArrayDemo1.java
- Array subscripts can be accessed using variables (such as for loop counters).
- See example: ArrayDemo2.java

Bounds Checking

- Array indexes always start at zero and continue to (array length - 1).

```
int values = new int[10];
```

- This array would have indexes 0 through 9. Java does not allow a statement to use a subscript outside the range of valid subscripts for an array.
- See example: InvalidSubscript.java
- In `for` loops, it is typical to use *i, j,* and *k* as counting variables.
  - It might help to think of *i* as representing the word *index*.

Off-by-One Errors

- It is very easy to be off-by-one when accessing arrays.

```
// This code has an off-by-one error.
int[] numbers = new int[100];
for (int i = 1; i <= 100; i++)
   numbers[i] = 99;
```

- Here, the equal sign allows the loop to continue on to index 100, where 99 is the last index in the array (runtime error).
- This code would throw an `ArrayIndexOutOfBoundsException`.

Array Initialization (2 of 2)

- When relatively few items need to be initialized, an initialization list can be used to initialize the array.

```
int[]days = {31, 28, 31, 30, 31, 30, 31, 31,
 30, 31, 30, 31};
```

- The numbers in the list are stored in the array in order:
  - `days[0]` is assigned 31,
  - `days[1]` is assigned 28,
  - `days[2]` is assigned 31,
  - `days[3]` is assigned 30,
  - etc.
- See example: ArrayInitialization.java

Array Initialization (1 of 2)

- Point to ponder #2:

  What is the other way to initialize this array?

```
int[]days = {31, 28, 31, 30, 31, 30, 31, 31, 30,
31, 30, 31};
```

```
int[]days = new int[12];
days[0] = 31
days[1] = 28
days[2] = 31
days[3] = 30 …
```

Alternate Array Declaration

- Previously we showed arrays being declared:

  ```
  int[] numbers;
  ```

  - However, the brackets can also go here:

    `int numbers[];` (similar to C/C++ style)

  - These are equivalent but the first style is typical.
- Multiple arrays can be declared on the same line.

  ```
  int[] numbers, codes, scores;
  ```

- With the alternate notation each variable must have brackets.

  ```
  int numbers[], codes[], scores;
  ```

  - The `scores` variable in this instance is simply an `int` variable.

Processing Array Contents (1 of 3)

- Processing data in an array is the same as any other variable.

```
int[] hours = {7, 8, 9, 10, 11};
int payRate = 5;
```

- Point to ponder #1:

  What is the value of `grossPay` below?

```
double grossPay = hours[3] * payRate;      50.0
double grossPay = hours[1] * payRate;      40.0
double grossPay = hours[5] * payRate;      error
```

See example: PayArray.java

Processing Array Contents (2 of 3)

- Pre and post increment works the same:

```
int[] score = {7, 8, 9, 10, 11};
int index = 2;
```

- Point to ponder #2:

  What are the values stored in the `score` array after those operations below?

```
++score[index];              {7, 8, 10, 10, 11}
score[4]--;                  {7, 8, 10, 10, 10}
int x = score[index--];      {7, 8, 10, 10, 10}
score[--index] = x;          {10, 8, 10, 10, 10}
```

Processing Array Contents (3 of 3)

- Array elements can be used in relational operations:

```
if(cost[20] < cost[0])
{
   //statements
}
```

- They can be used as loop conditions:

```
while(value[count] != 0)
{
   //statements
}
```

Array Length (1 of 4)

- Arrays are objects and provide a **public** field named `length` that is a **constant** that can be tested.

```
double[]   temperatures   =   new   double[25];
```

The length of temperatures is 25

- The length of an array can be obtained via its `length` constant.

```
int size = temperatures.length;
```

The variable size will contain 25

Array Length (2 of 4)

- The length field can be useful when processing the entire contents of an array.
- For example, the array's length field is used in the test expression as the upper limit for the loop control variable.

```
for (int i = 0; i < temperatures.length; i++)
    System.out.println(temperatures[i]);
```

Array Length (3 of 4)

- Point to ponder #3:

  What is the output below?

```
String word = "java";
int[] score = {7, 8, 9, 10, 11};

System.out.println(word.length());   4

System.out.println(score.length);    5
```

Array Length (4 of 4)

- Point to ponder #4:

  How to access the last element of the string word and of the array score (its largest subscript)?

```
String word = "Hello";
int[] score = {7, 8, 9, 10, 11};
```

```
System.out.println(word.charAt(word.length()-1)); //output: o

System.out.println(score[score.length-1]); //output: 11
```

- Be careful not to cause an off-by-one error when using the `length` field as the upper limit of a subscript

The Enhanced `for` Loop (1 of 5)

- Simplified array processing (read only)
- Always goes through all elements
- General format:

```
for(datatype elementVariable : array)
    statement;
```

Each time the loop iterates, it copies an array element to a variable

This variable must be of the same data type as the array elements, or a type that the elements can automatically be converted to.

The Enhanced `for` Loop (2 of 5)

Example:

```
int[] numbers = {3, 6, 9};
for(int val : numbers)
{
  System.out.println("The next value is " + val);
}
```

- Point to ponder #5:
  What is the output above?

```
The next value is 3
The next value is 6
The next value is 9
```

The Enhanced `for` Loop (3 of 5)

- Point to ponder #6:

  What is the difference between the outputs produced by the two loops below?

```
int[] numbers = {3, 6, 9};

for(int val : numbers)
{
  System.out.println("The next value is " + val);
}

for(int i = 0; i < numbers.length; i++)
{
  System.out.println("The next value is " + numbers[i]);
}
```

No difference!

The Enhanced `for` Loop (4 of 5)

- Point to ponder #7:

  So, what is the difference between for loop and enhanced for loop in Java?

  1. In enhanced for loop, the counter is always increased by one (for loop allows any step, e.g, i+=2);
  2. Enhanced for loop can only iterate in incremental order (for loop allows backward iteration, e.g., i--).
  3. In enhanced for loop, we don't have access to array index, which means we cannot replace the element at a given index (for loop allows, e.g., array[i]++)

The Enhanced `for` Loop (5 of 5)

- Point to ponder #8:

  So why “enhanced for loop”?

  In general, enhanced for loop is much easier to use and less error-prone than for loop, where you need to manage the steps manually. It just provides an alternative (simpler) approach to traverse the array or collection in Java

  However, for loop is much more powerful because you get the opportunity to control over looping process.

Array Size (1 of 2)

- The `length` constant can be used in a loop to provide automatic bounding.

Index subscripts start at 0 and end at one ***less than*** the array length.

```
for(int i = 0; i < temperatures.length; i++)
{
   System.out.println("Temperature " + i ": "
                                + temperatures[i]);
}
```

Array Size (2 of 2)

- You can let the user specify the size of an array:

```
int numTests;
int[] tests;

Scanner keyboard = new Scanner(System.in);
System.out.print("How many tests do you have? ");
numTests = keyboard.nextInt();
tests = new int[numTests];
```

- See example: DisplayTestScores.java

Reassigning Array References (1 of 3)

- It is possible to reassign an array reference variable to a different array.

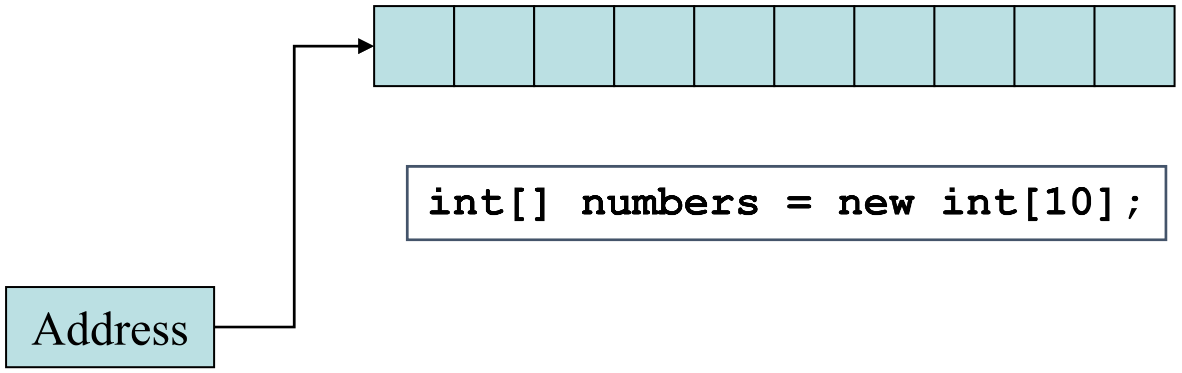
```
// Create an array referenced by the numbers variable.
int[] numbers = new int[10];

// Reassign numbers to a new array.
numbers = new int[5];
```

Two different objects!

- If the first (10 element) array no longer has a reference to it, it will be garbage collected.

Reassigning Array References (2 of 3)

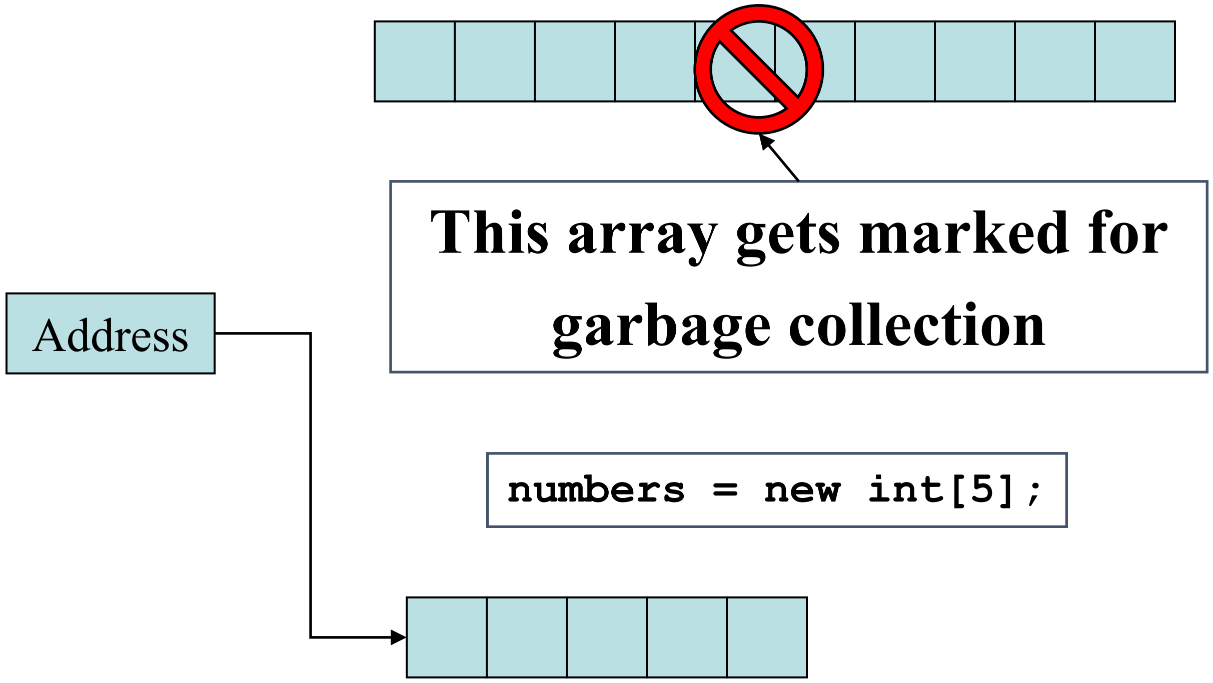


```
int[] numbers = new int[10];
```

The `numbers` variable holds the address of an `int` array.

Reassigning Array References (3 of 3)

The `numbers` variable holds the address of an `int` array.

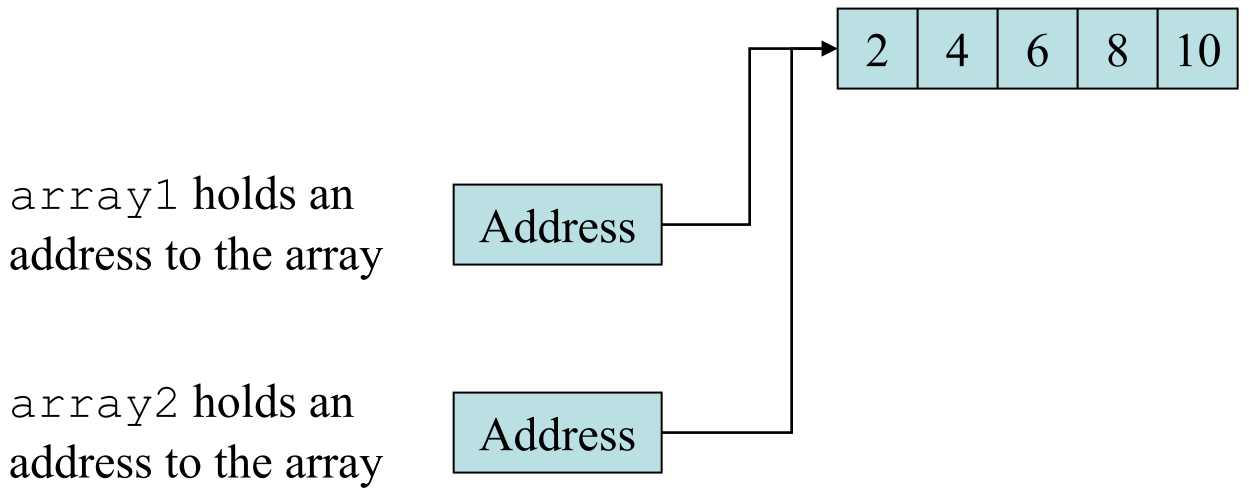


```
numbers = new int[5];
```

Copying Arrays (1 of 2)

- This is *not* the way to copy an array.

```
int[] array1 = { 2, 4, 6, 8, 10 };
int[] array2 = array1; // This does not copy array1.
```



2
4
6
8
10
array1 holds an address to the array
Address
array2 holds an address to the array
Address

Example: SameArray.java

Copying Arrays (2 of 2)

- You cannot copy an array by merely assigning one reference variable to another.
- You need to copy the individual elements of one array to another.

```
int[] firstArray = {5, 10, 15, 20, 25 };
int[] secondArray = new int[5];
for (int i = 0; i < firstArray.length; i++)
  secondArray[i] = firstArray[i];
```

- This code copies each element of `firstArray` to the corresponding element of `secondArray`.

Passing Array Elements to a Method

- When a single element of an array is passed to a method it is handled like any other variable.
- See example: PassElements.java
- More often you will want to write methods to process array data by passing the entire array, not just one element at a time.

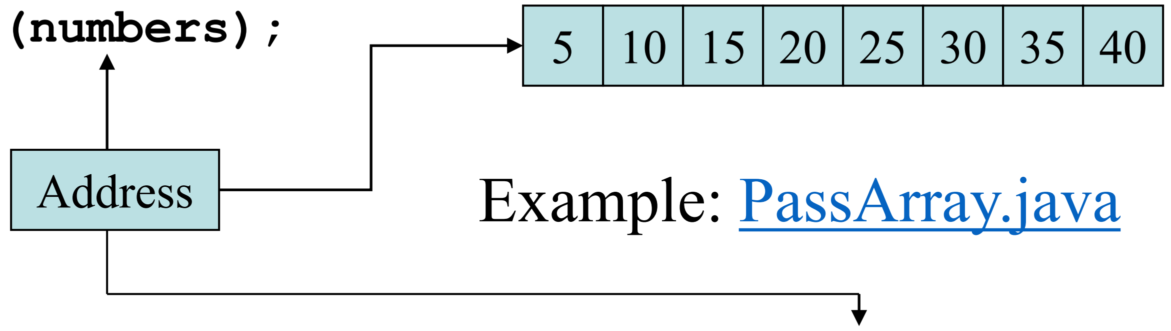
```
int[] numbers = {5, 10, 15, 20, 25, 30, 35, 40};
showValue(numbers[2]);
public static void showValue(int n)
{
   System.out.print(n);
}
```

Output? 15

Passing Arrays as Arguments

- Arrays are objects.
- Their references can be passed to methods like any other object reference variable.

```
showArray(numbers);
```



Example: PassArray.java

```
public static void showArray(int[] array)
{
   for (int i = 0; i < array.length; i++)
      System.out.print(array[i] + " ");
}
```

Comparing Arrays

- The == operator determines only whether array references point to the same array object.

```
int[] firstArray = { 5, 10, 15, 20, 25 };
int[] secondArray = { 5, 10, 15, 20, 25 };

if (firstArray == secondArray) // This is a mistake.
   System.out.println("The arrays are the same.");
else
   System.out.println("The arrays are not the same.");
```

Comparing Arrays: Example

```
int[] firstArray = { 2, 4, 6, 8, 10 };
int[] secondArray = { 2, 4, 6, 8, 10 };
boolean arraysEqual = true;
int i = 0;

// First determine whether the arrays are the same size.
if (firstArray.length != secondArray.length)
   arraysEqual = false;

// Next determine whether the elements contain the same data.
while (arraysEqual && i < firstArray.length)
{
  if (firstArray[i] != secondArray[i])
     arraysEqual = false;
  i++;
}

if (arraysEqual)
  System.out.println("The arrays are equal.");
else
  System.out.println("The arrays are not equal.");
```

Useful Array Operations (1 of 2)

- Summing Array Elements:

```
int total = 0; // Initialize accumulator
for (int i = 0; i < units.length; i++)
    total += units[i];
```

units

4	2	5	1

total = 12

- Averaging Array Elements:

```
double total = 0; // Initialize accumulator
double average; // Will hold the average
for (int i = 0; i < scores.length; i++)
        total += scores[i];
average = total / scores.length;
```

scores

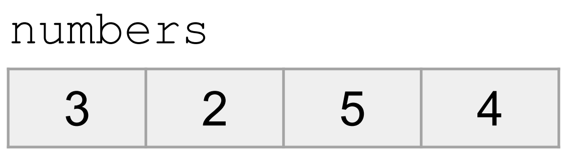
4	2	5	1

total = 12
average = 3

Useful Array Operations (2 of 2)

- Finding the Highest Value

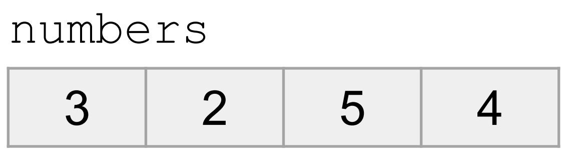
```
int highest = numbers[0];
for (int i = 1; i < numbers.length; i++)
{
        if (numbers[i] > highest)
                highest = numbers[i];
}
```



highest = 5

- Finding the Lowest Value

```
int lowest = numbers[0];
for (int i = 1; i < numbers.length; i++)
{
        if (numbers[i] < lowest)
                lowest = numbers[i];
}
```



lowest = 2

Example: SalesData.java, Sales.java

Partially Filled Arrays (1 of 3)

- Typically, if the amount of data that an array must hold is unknown:
    - set the size the array to the largest expected number of elements.
    - use a counting variable to keep track of how much valid data is in the array.

```
…
int[] array = new int[100];
int count = 0;
…
  System.out.print("Enter a number or -1 to quit: ");
  number = keyboard.nextInt();
  while (number != -1 && count <= 99)
  {
    array[count] = number;
    count++;
    System.out.print("Enter a number or -1 to quit: ");
    number = keyboard.nextInt();
  }
…
```

`number` and `keyboard` were previously declared, and `keyboard` references a `Scanner` object

Partially Filled Arrays (2 of 3)

Example:

```
…
int[] array = new int[5];
int count = 0;
…
  System.out.print("Enter a number or -1 to quit: ");
  number = keyboard.nextInt();
  while (number != -1 && count <= 4)
  {
    array[count] = number;
    count++;
    System.out.print("Enter a number or -1 to quit: ");
    number = keyboard.nextInt();
  }
…
```

array

0	0	0	0	0

User input

2	5	3	-1

array

2	5	3	0	0

Partially Filled Arrays (3 of 3)

Displaying all the valid items in the array:

```
for (int index = 0; index < count; index++)
{
   System.out.println(array[index]);
}
```

- Point to ponder #8:
  Why iterate over `count` instead of `array.length`?

  To show valid items only.

`array`

2	5	3	0	0

Output:
```
2
5
3
```

Arrays

Lecture 7b

Topics (1 of 2)

- Introduction to Arrays
- Processing Array Contents
- Passing Arrays as Arguments to Methods
- Some Useful Array Algorithms and Operations
- Returning Arrays from Methods
- String Arrays
- Arrays of Objects

Topics (2 of 2)

- Two-Dimensional Arrays
- Arrays with Three or More Dimensions
- Command-Line Arguments
- ArrayList

Returning an Array Reference (1 of 2)

- A method can return a reference to an array.
- The return type of the method must be declared as an array of the right type.

```
public static double[] getArray()
{
  double[] array = { 1.2, 2.3, 4.5, 6.7, 8.9 };
  return array;
}
```

- The `getArray` method is a public static method that returns an array of doubles.
- See example: [ReturnArray.java]

Returning an Array Reference (2 of 2)

- Point to ponder #1:

  What type of variable (data type) can receive the output of the getArray() method?

  double[]

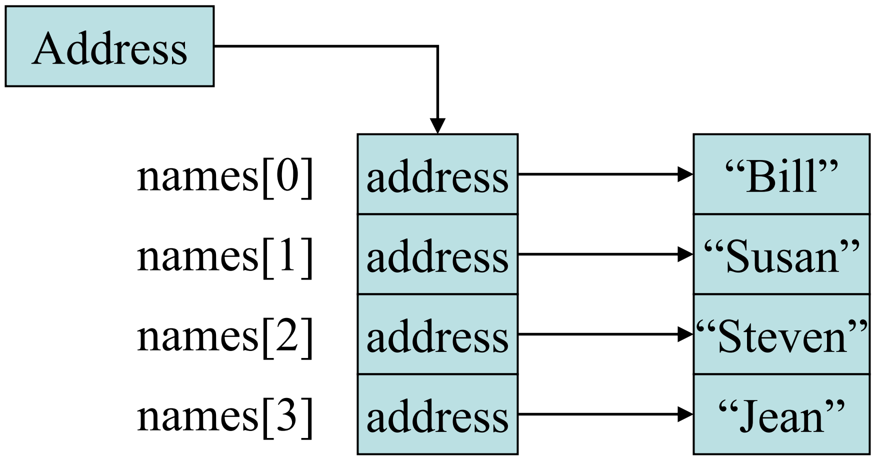
String Arrays (1 of 3)

- Arrays are not limited to primitive data.
- An array of String objects can be created:

```
String[] names = { "Bill", "Susan",
 "Steven", "Jean" };
```

The names variable holds the address to the array.

A String array is an array of references to String objects.



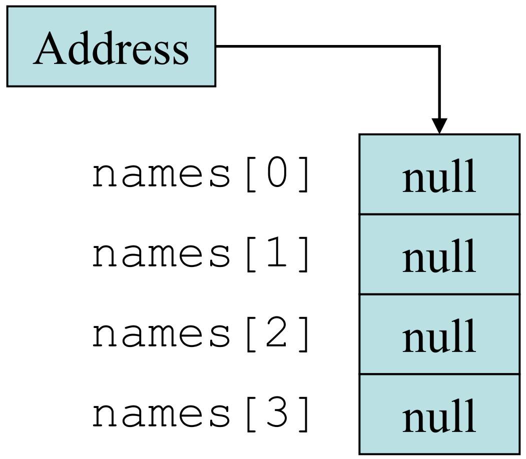
Example:
[MonthDays.java]

`String` Arrays (2 of 3)

- If an initialization list is not provided, the `new` keyword must be used to create the array:

```
String[] names = new String[4];
```

The `names` variable holds the address to the array.

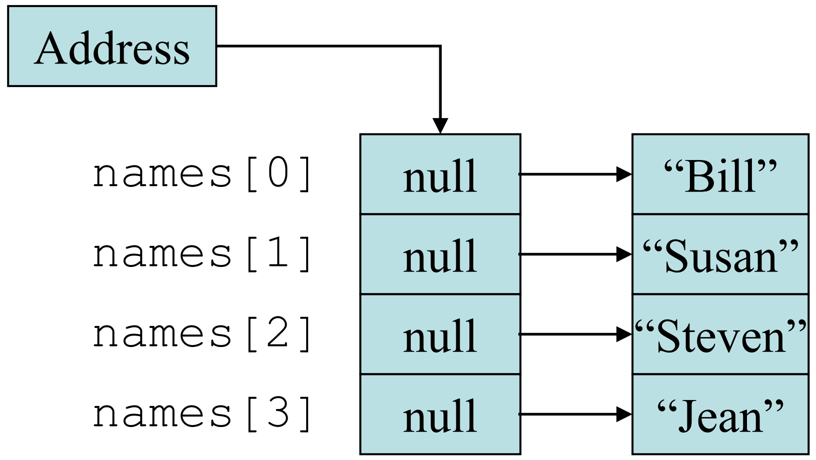


`String` Arrays (3 of 3)

- When an array is created in this manner, each element of the array must be initialized.

```
names[0] = "Bill";
names[1] = "Susan";
names[2] = "Steven";
names[3] = "Jean";
```

The `names` variable holds the address to the array.



Calling `String` Methods On Array Elements

- `String` objects have several methods, including:
  - `toUpperCase`
  - `compareTo`
  - `equals`
  - `charAt`
- Each element of a `String` array is a `String` object.
- Methods can be used by using the array name and index as before.

```
System.out.println(names[0].toUpperCase());
char letter = names[3].charAt(0);
```

Combining the `length` Field & The `length` Method

- To display the length of each string held in a `String` array:

```
for (int i = 0; i < names.length; i++)
    System.out.println(names[i].length());
```

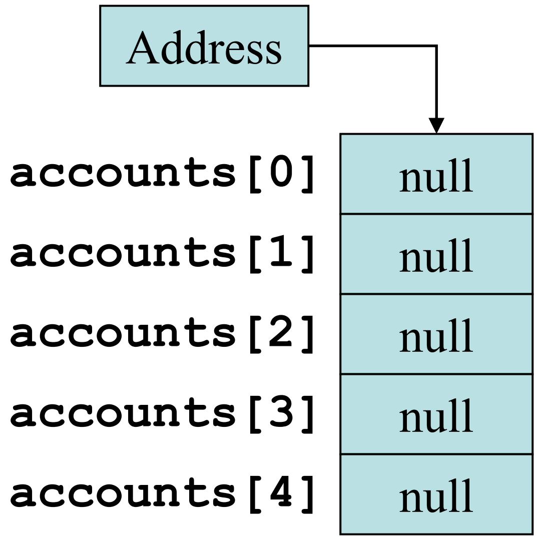
- An array's `length` is a **field**
  - You do not write a set of parentheses after its name.
- A `String`'s `length` is a **method**
  - You do write the parentheses after the name of the `String` class's `length` method.

Arrays of Objects (1 of 2)

- Because `String`s are objects, we know that arrays can contain objects.

```
BankAccount[] accounts = new BankAccount[5];
```

The `accounts` variable holds the address of an `BankAccount` array.



The array is an array of references to `BankAccount` objects.

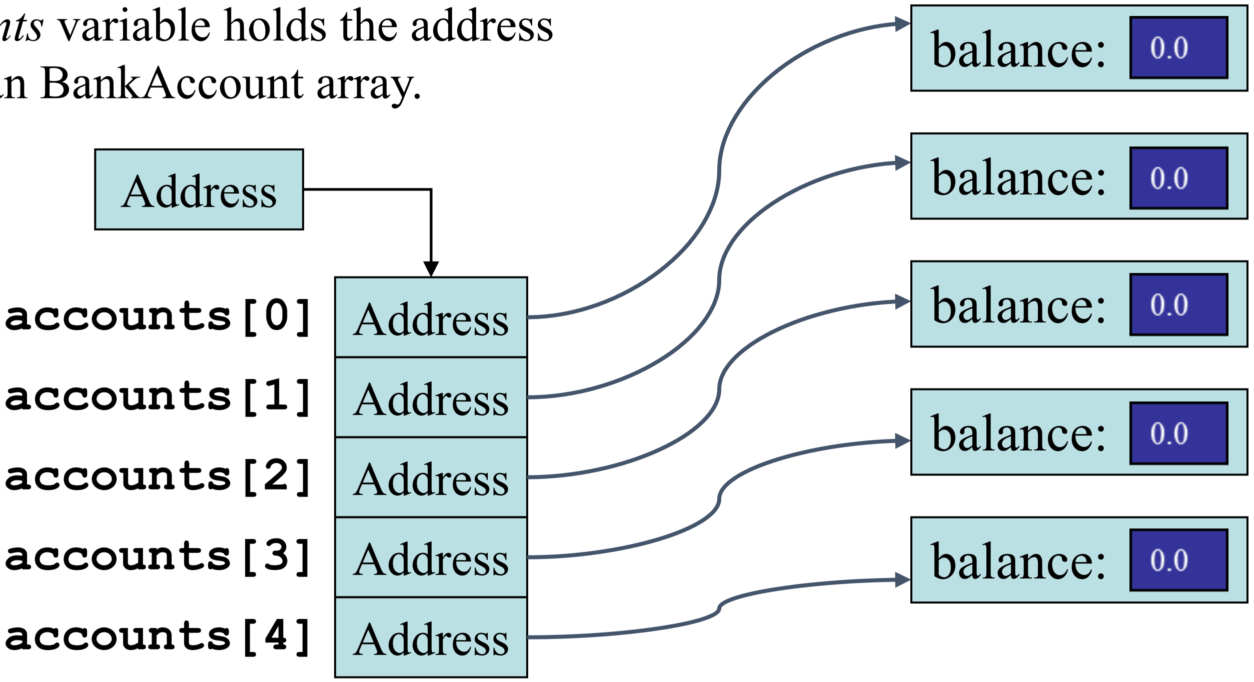
Arrays of Objects (2 of 2)

- Each element needs to be initialized.

```
for (int i = 0; i < accounts.length; i++)
  accounts[i] = new BankAccount();
```

- See example: ObjectArray.java

The *accounts* variable holds the address of an BankAccount array.



Two-Dimensional Arrays (1 of 2)

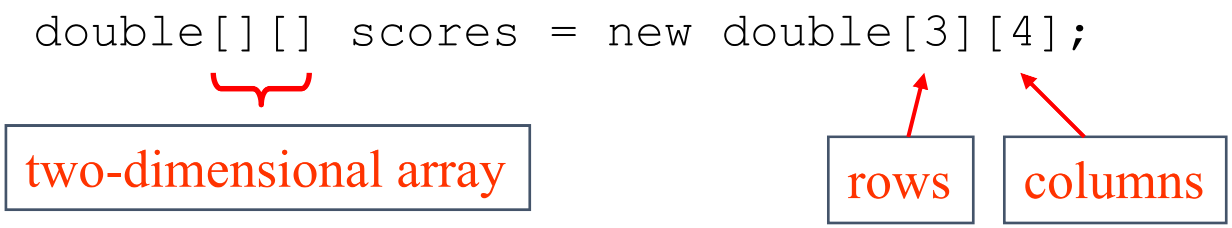
- A two-dimensional array is an array of arrays.
- It can be thought of as having rows and columns.

	column 0	column 1	column 2	column 3
row 0				
row 1				
row 2				

3 rows and 4 columns

Two-Dimensional Arrays (2 of 2)

- Declaring a two-dimensional array requires two sets of brackets and two size declarators
  - The first one is for the number of rows
  - The second one is for the number of columns.



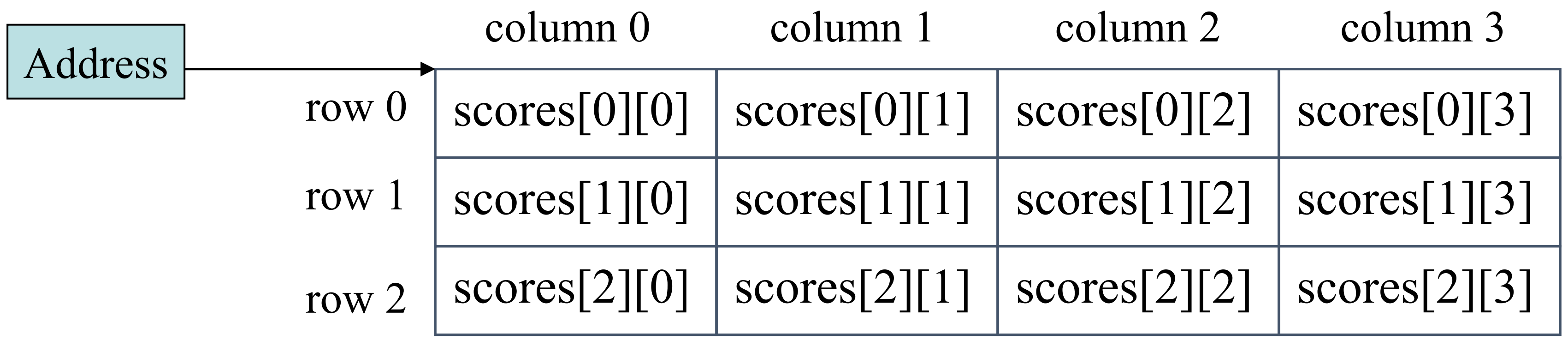
```
double[][] scores = new double[3][4];
```

- The two sets of brackets in the data type indicate that the scores variable will reference a two-dimensional array.
- Notice that each size declarator is enclosed in its own set of brackets.

Accessing Two-Dimensional Array Elements (1 of 4)

- When processing the data in a two-dimensional array, each element has two subscripts:
  - one for its row and
  - another for its column.

The `scores` variable holds the address of a 2D array of `double`s.



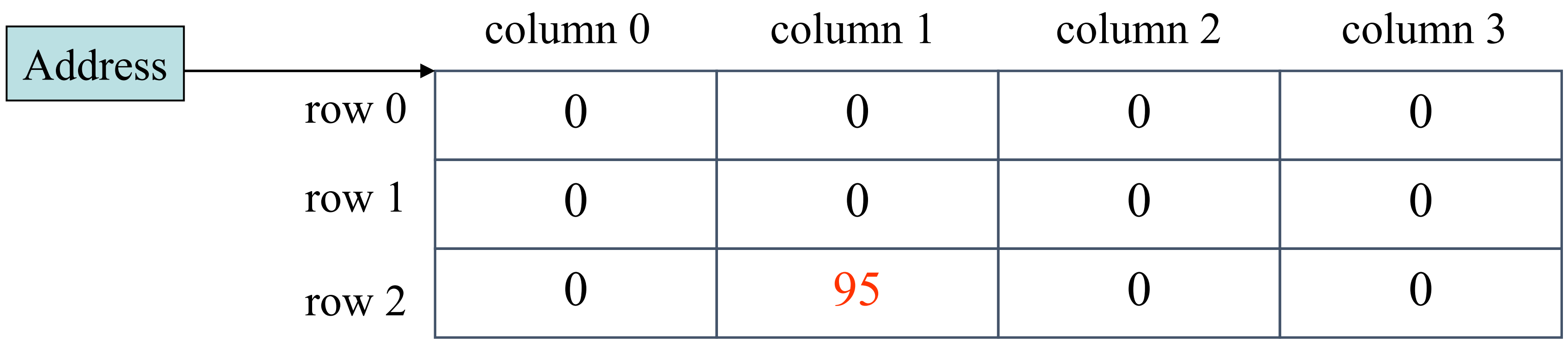
	column 0	column 1	column 2	column 3
row 0	scores[0][0]	scores[0][1]	scores[0][2]	scores[0][3]
row 1	scores[1][0]	scores[1][1]	scores[1][2]	scores[1][3]
row 2	scores[2][0]	scores[2][1]	scores[2][2]	scores[2][3]

Accessing Two-Dimensional Array Elements (2 of 4)

Accessing one of the elements in a two-dimensional array requires the use of both subscripts.

The `scores` variable holds the address of a 2D array of `double`s.

```
scores[2][1] = 95;
```



	column 0	column 1	column 2	column 3
row 0	0	0	0	0
row 1	0	0	0	0
row 2	0	95	0	0

Accessing Two-Dimensional Array Elements (3 of 4)

- Programs that process two-dimensional arrays can do so with nested loops.
- To fill the scores array:

```
for (int row = 0; row < 3; row++)
{
  for (int col = 0; col < 4; col++)
  {
    System.out.print("Enter a score: ");
    scores[row][col] = keyboard.nextDouble();
  }
}
```

Number of rows, not the largest subscript

Number of columns, not the largest subscript

`keyboard` references a `Scanner` object

Accessing Two-Dimensional Array Elements (4 of 4)

- To print out the `scores` array:

```
for (int row = 0; row < 3; row++)
{
   for (int col = 0; col < 4; col++)
    {
      System.out.println(scores[row][col]);
   }
}
```

- See example: CorpSales.java

Initializing a Two-Dimensional Array (1 of 2)

- Initializing a two-dimensional array requires enclosing each row’s initialization list in its own set of braces.

```
int[][] numbers = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
```

- Java automatically creates the array and fills its elements with the initialization values.
  - row 0 {1, 2, 3}
  - row 1 {4, 5, 6}
  - row 2 {7, 8, 9}

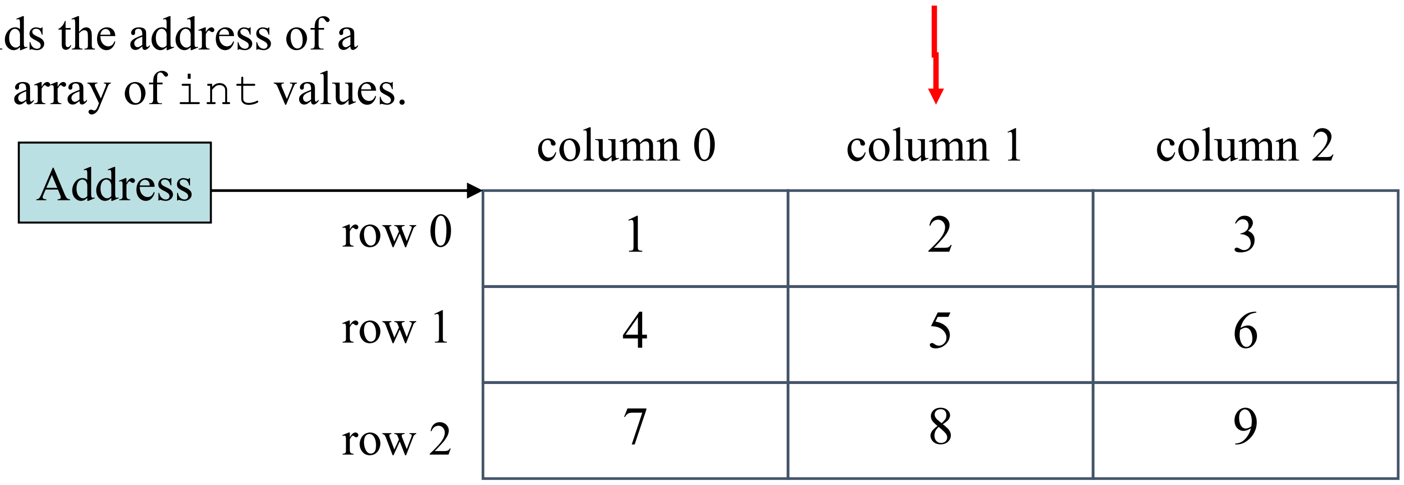
Initializing a Two-Dimensional Array (2 of 2)

For more clarity, the same statement could also be written as follows:

```
int[][] numbers = {{1, 2, 3},
                   {4, 5, 6},
                   {7, 8, 9}};
```

produces:

The `numbers` variable holds the address of a 2D array of `int` values.



The `length` Field (1 of 2)

- Two-dimensional arrays are arrays of one-dimensional arrays.
- The length field of the array gives the number of rows in the array.
- Each row has a length constant tells how many columns is in that row.
- Each row can have a different number of columns.

The `length` Field (2 of 2)

- To access the `length` fields of the array:

```
int[][] numbers = { { 1, 2, 3, 4 },
                    { 5, 6, 7 },
                    { 9, 10, 11, 12 } };

for (int row = 0; row < numbers.length; row++)
{
  for (int col = 0; col < numbers[row].length; col++)
    System.out.println(numbers[row][col]);
}
```

Number of rows Number of columns in this row.

- See example: Lengths.java

The array can have variable length rows.

Summing The Elements of a Two-Dimensional Array

```
int[][] numbers = { { 1, 2, 3, 4 },
                    {5, 6, 7, 8},
                    {9, 10, 11, 12} };
int total;
total = 0;
for (int row = 0; row < numbers.length; row++)
{
  for (int col = 0; col < numbers[row].length; col++)
    total += numbers[row][col];
}

System.out.println("The total is " + total);
```

Summing The Rows of a Two-Dimensional Array

```
int[][] numbers = { { 1, 2, 3, 4 },
                    {5, 6, 7, 8},
                    {9, 10, 11, 12}};
int total;

for (int row = 0; row < numbers.length; row++)
{
  total = 0;
  for (int col = 0; col < numbers[row].length; col++)
      total += numbers[row][col];
  System.out.println("Total of row " + row + " is " +
                                 total);
}
```

Summing The Columns of a Two-Dimensional Array

```
int[][] numbers = {{1, 2, 3, 4},
                   {5, 6, 7, 8},
                   {9, 10, 11, 12}};
int total;

for (int col = 0; col < numbers[0].length; col++)
{
  total = 0;
  for (int row = 0; row < numbers.length; row++)
    total += numbers[row][col];
  System.out.println("Total of column "
                     + col + " is " + total);
}
```

Passing and Returning Two-Dimensional Array References

- There is no difference between passing a single or two-dimensional array as an argument to a method.
- The method must accept a two-dimensional array as a parameter.

```
private static void showArray(int[][] array)
```

- See example: Pass2Darray.java

Ragged Arrays

- When the rows of a two-dimensional array are of different lengths, the array is known as a *ragged array*.
- You can create a ragged array by creating a two-dimensional array with a specific number of rows, but no columns.

```
    int [][] ragged = new int [4][];
```

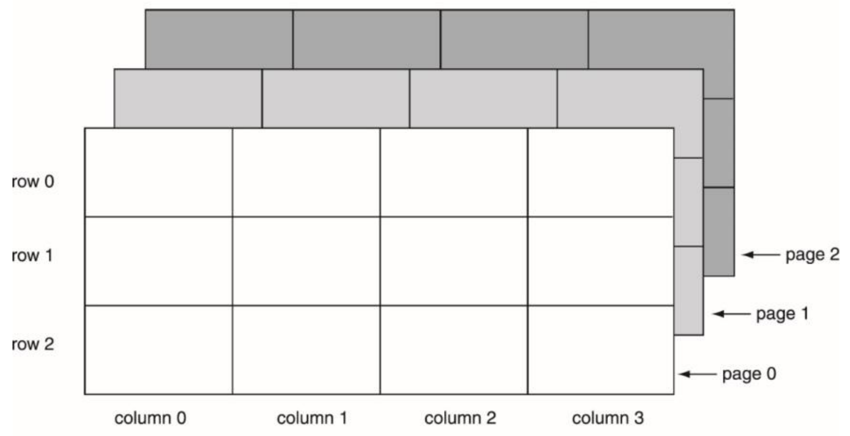
- Then create the individual rows.

```
   ragged[0] = new int [3];
   ragged[1] = new int [4];
   ragged[2] = new int [5];
   ragged[3] = new int [6];
```

More Than Two Dimensions

- Java does not limit the number of dimensions that an array may be.
- More than three dimensions is hard to visualize but can be useful in some programming problems.

```
double[][][] seats = new double[3][5][8]; (3D-array)
```



Command-Line Arguments (1 of 2)

- A Java program can receive arguments from the operating system command-line.
- The `main` method has a header that looks like this:

```
public static void main(String[] args)
```

- The `main` method receives a `String` array as a parameter.
- The array that is passed into the `args` parameter comes from the operating system command-line.

Command-Line Arguments (2 of 2)

- To run the example:

```
java CommandLine How does this work?
    args[0] is assigned "How"
    args[1] is assigned "does"
    args[2] is assigned "this"
    args[3] is assigned "work?"
```

- Example: CommandLine.java
- It is not required that the name of `main`'s parameter array be `args`.

The `ArrayList` Class

- Similar to an array, an `ArrayList` allows object storage
- Unlike an array, an `ArrayList` object:
  - automatically expands when a new item is added
  - automatically shrinks when items are removed
- Requires:

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

Creating an `ArrayList`

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

Notice the word `String` written inside angled brackets <>

This specifies that the `ArrayList` can hold `String` objects.

If we try to store any other type of object in this `ArrayList`, an error will occur.

Using an `ArrayList` (1 of 9)

- To populate the `ArrayList`, use the `add` method:
    - `nameList.add("James");`
    - `nameList.add("Catherine");`
- To get the current size, call the `size` method
    - `nameList.size();   // returns 2`

Using an `ArrayList` (2 of 9)

- To access items in an `ArrayList`, use the `get` method

```
        nameList.get(1);
```

  - In this statement, 1 is the index of the item.

- The index specifies the item's location in the ArrayList, so it is much like an array subscript.

- The first item that is added to an ArrayList is stored at index 0.

- Example: ArrayListDemo1.java

Using an `ArrayList` (3 of 9)

- The `ArrayList` class's `toString` method returns a string representing all items in the `ArrayList`

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

This statement yields :

```
[ James, Catherine ]
```

Using an `ArrayList` (4 of 9)

- The `ArrayList` class's `remove` method removes designated item from the `ArrayList`

```
nameList.remove(1);
```

This statement removes the second item.

- When an item is removed from an `ArrayList`, the items that come after it are shifted downward in position to fill the empty space.
- This means that the index of each item after the removed item will be decreased by one.

- See example: ArrayListDemo3.java

Using an `ArrayList` (5 of 9)

- To insert items at a location of choice, use the `add` method (overload) with two arguments:

  ```
  nameList.add(1, "Emily");
  ```

  This statement inserts the `String` “Emily" at index 1
- When an item is added at a specific index, the items that come after it are shifted upward in position to accommodate the new item.
- This means that the index of each item after the new item will be increased by one.
- See example: ArrayListDemo4.java

Using an `ArrayList` (6 of 9)

- To replace an existing item, use the `set` method:

```
nameList.set(1, "Becky");
```

This statement replaces “Mary” with “Becky”

- See example: ArrayListDemo5.java

Using an `ArrayList` (7 of 9)

- An `ArrayList` has a capacity, which is the number of items it can hold without increasing its size.
- The default capacity of an `ArrayList` is 10 items.
- To designate a different capacity, use a parameterized constructor:

```
ArrayList<String> list = new ArrayList<String>(100);
```

Using an `ArrayList` (8 of 9)

- You can store any type of *object* in an `ArrayList`

```
ArrayList<BankAccount> accountList =
                 new ArrayList<BankAccount>();
```

This creates an `ArrayList` that can hold `BankAccount` objects.

Using an `ArrayList` (9 of 9)

```
// Create an ArrayList to hold BankAccount objects.
ArrayList<BankAccount> list = new ArrayList<BankAccount>();

// Add three BankAccount objects to the ArrayList.
list.add(new BankAccount(100.0));
list.add(new BankAccount(500.0));
list.add(new BankAccount(1500.0));

// Display each item.
for (int index = 0; index < list.size(); index++)
{
   BankAccount account = list.get(index);
   System.out.println("Account at index " + index +
                "\nBalance: " + account.getBalance());
}
```

See example: ArrayListDemo6.java