

starting out with >>>

**JAVA**<sup>TM</sup> **EARLY OBJECTS**

FIFTH EDITION

## CHAPTER 7

# Arrays and the ArrayList Class



TONY GADDIS

Addison-Wesley  
is an imprint of

PEARSON

Copyright © 2015 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

# Topics

- 🍓 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 (cont'd)

- 🍓 The Sequential Search Algorithm
- 🍓 The Selection Sort and the Binary Search
- 🍓 Two-Dimensional Arrays
- 🍓 Arrays with Three or More Dimensions
- 🍓 Command-Line Arguments
- 🍓 The `ArrayList` Class

# Introduction to Arrays

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

- 🍓 **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];
```

0	0	0	0	0	0
index 0	index 1	index 2	index 3	index 4	index 5

Array element values are initialized to 0.

Array indexes always start at 0.

# Creating Arrays

- 🍷 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

- 🍷 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	0	0	0
<code>numbers[0]</code>	<code>numbers[1]</code>	<code>numbers[2]</code>	<code>numbers[3]</code>	<code>numbers[4]</code>	<code>numbers[5]</code>

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

`numbers[0] = 20; // pronounced "numbers sub zero"`



# 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.
- 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.
- 🍓 This code would throw an **ArrayIndexOutOfBoundsException**.

# Array Initialization

- 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

- And so forth...

- See example: [ArrayInitialization.java](#)

# Alternate Array Declaration

- Previously we showed arrays being declared:

```
int[] numbers;
```

- However, the brackets can also go here:

```
int numbers[];
```

- 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

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

```
grossPay = hours[3] * payRate;
```

- Pre and post increment works the same:

```
int[] score = {7, 8, 9, 10, 11};  
++score[2]; // Pre-increment operation  
score[4]++; // Post-increment operation
```

- See example: [PayArray.java](#)

# Processing Array Contents

 **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

- 🍓 **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 this array is 25.

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

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

- 🍓 The variable `size` will contain 25.



# The Enhanced for Loop

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

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

## 🍷 Example:

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

# Array Size

- 🍓 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

- 🍓 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

- 🍷 An array reference can be assigned to another array of the same type.

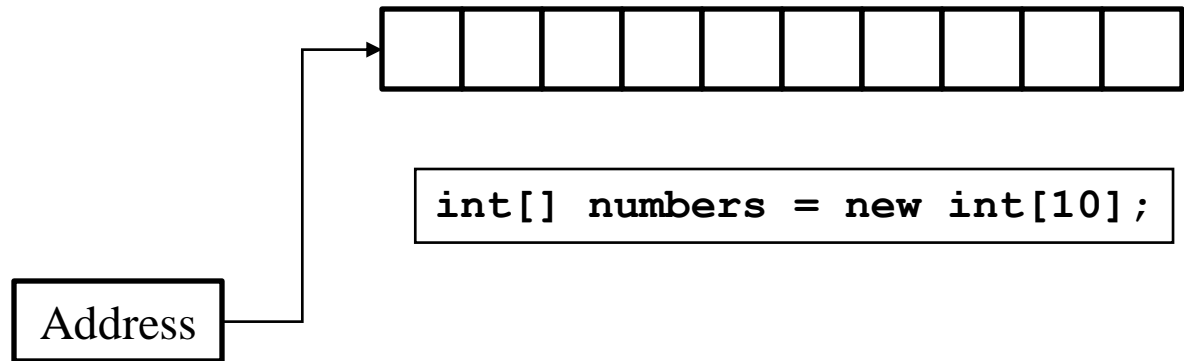
```
// Create an array referenced by the numbers variable.  
int[] numbers = new int[10];
```

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

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

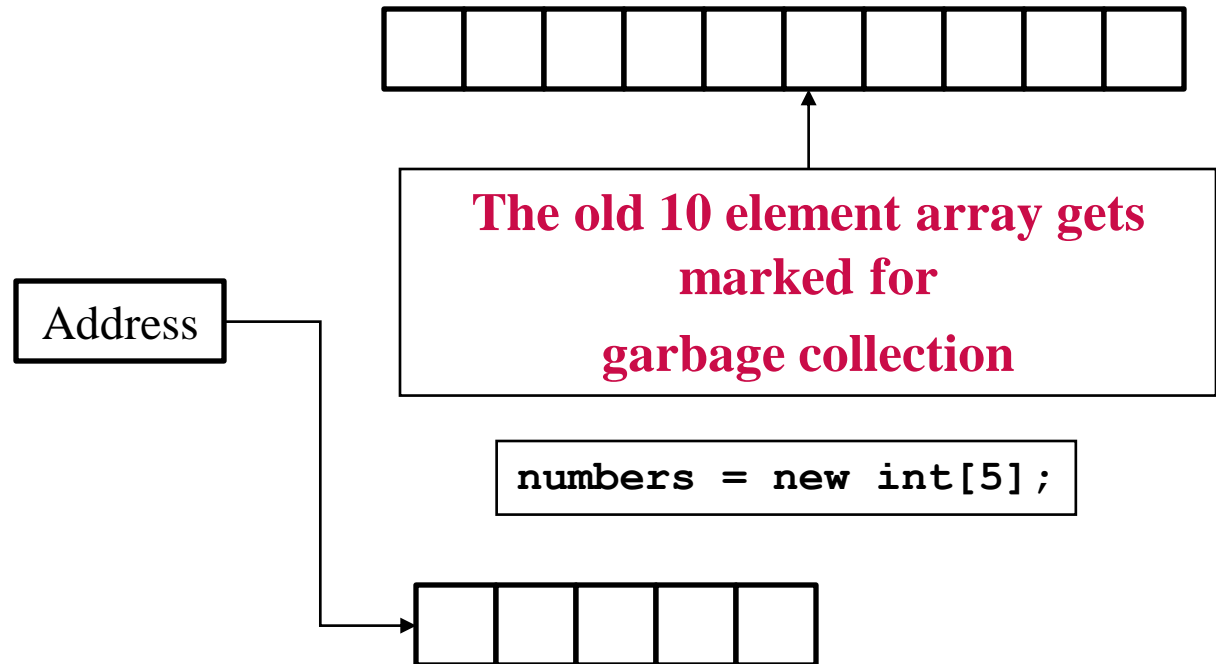
# Reassigning Array References

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



# Reassigning Array References

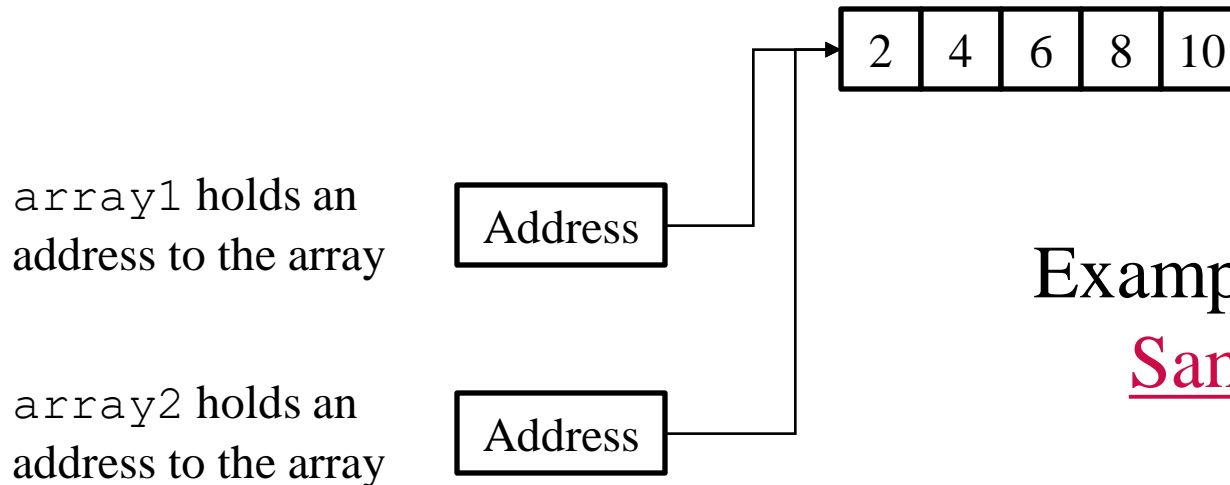
The `numbers` variable holds the address of an `int` array.



# Copying Arrays

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

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



Example:

[SameArray.java](#)

# Copying Arrays

- 🍓 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 each corresponding element of `secondArray`.

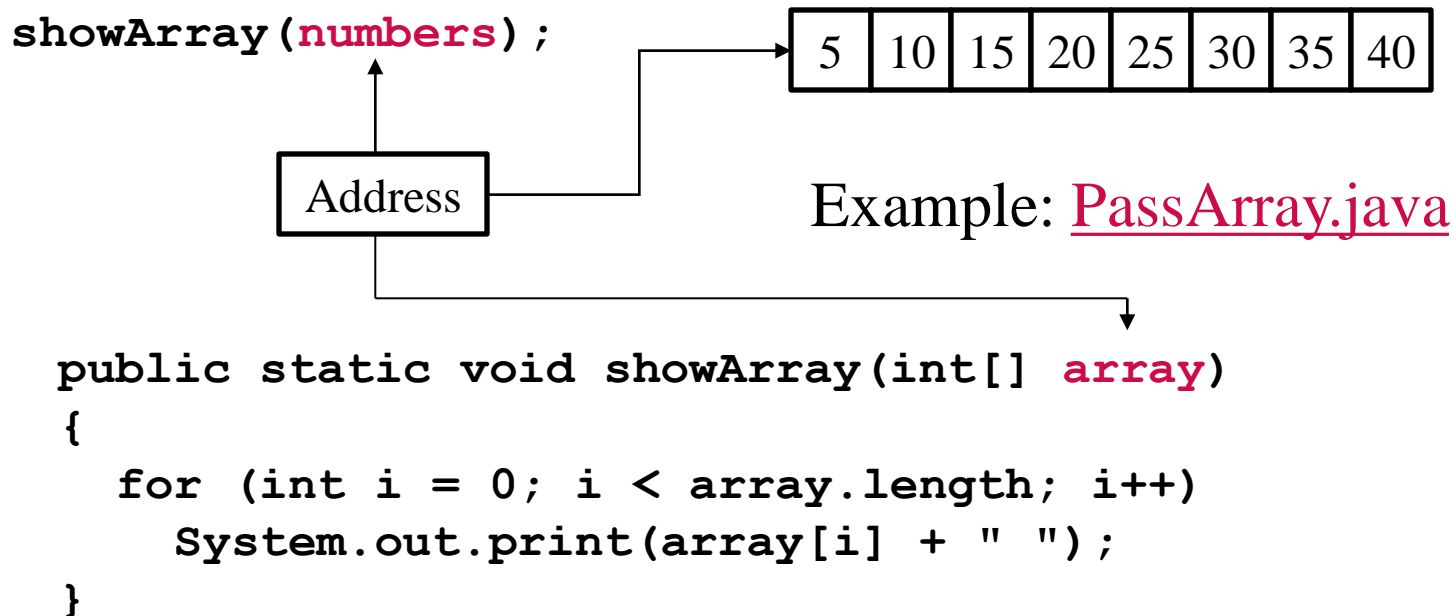


# Passing Array Elements to a Method

- 🍓 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.

# Passing Arrays as Arguments

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



# Comparing Arrays

- 🍓 The == operator determines only whether array references point to the same array object.
- 🍓 To compare the contents of an array:

```
int[] firstArray = { 2, 4, 6, 8, 10 };
int[] secondArray = { 2, 4, 6, 8, 10 };
boolean arraysEqual = true;
int i = 0;
if (firstArray.length != secondArray.length)
    arraysEqual = false;
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



## Finding the Highest Value:

```
int [] numbers = new int[50];  
int highest = numbers[0];  
for (int i = 1; i < numbers.length; i++)  
{  
    if (numbers[i] > highest)  
        highest = numbers[i];  
}
```



## Finding the Lowest Value:

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

# Useful Array Operations

## Summing Array Elements:

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

## 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;
```

## Example: [SalesData.java](#), [Sales.java](#)

# Partially Filled Arrays

- Typically, if the amount of data that an array must hold is unknown:
  - 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();
}
```

... *input, number and keyboard were previously declared and keyboard references a Scanner object.*

# Arrays and Files

🍓 **Saving the contents of an array to a file:**

```
int[] numbers = {10, 20, 30, 40, 50};
```

```
PrintWriter outputFile = new PrintWriter  
("Values.txt");
```

```
for (int i = 0; i < numbers.length; i++)  
    outputFile.println(numbers[i]);
```

```
outputFile.close();
```

# Arrays and Files

## Reading the contents of a file into an array:

```
final int SIZE = 5; // Assuming we know the size.
int[] numbers = new int[SIZE];
int i = 0;
File file = new File ("Values.txt");
Scanner inputFile = new Scanner(file);
while (inputFile.hasNext() && i < numbers.length)
{
    numbers[i] = inputFile.nextInt();
    i++;
}
inputFile.close();
```



# Returning an Array Reference

- 🍷 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](#)

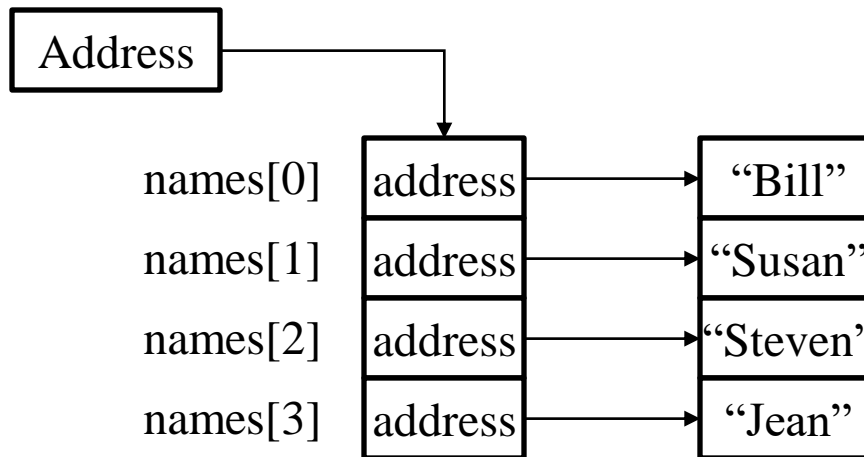
# String Arrays

- 🍷 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:

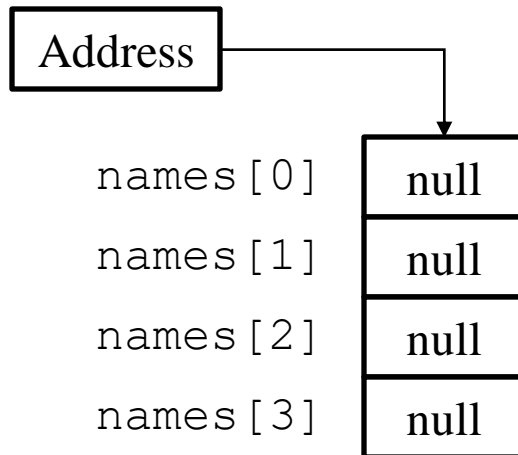
[MonthDays.java](#)

# String Arrays

- 🍷 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.

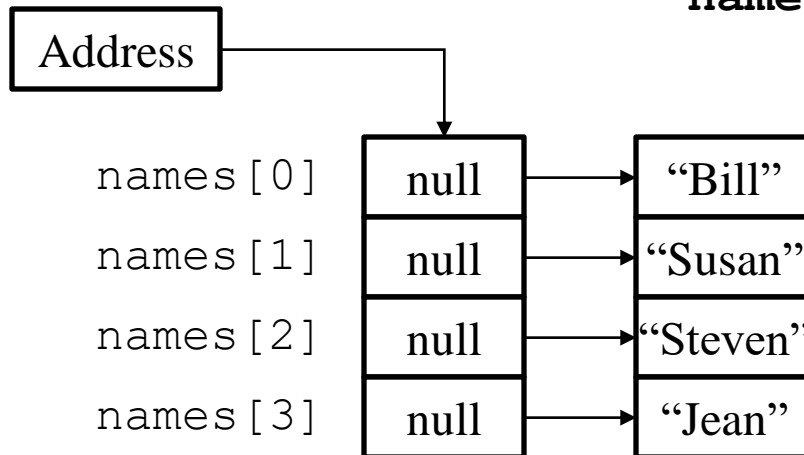


# String Arrays

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

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

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



# 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);
```

# The length Field & The length Method

- Arrays have a `final` field named `length`.
- String objects have a method named `length`.
- 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());
```

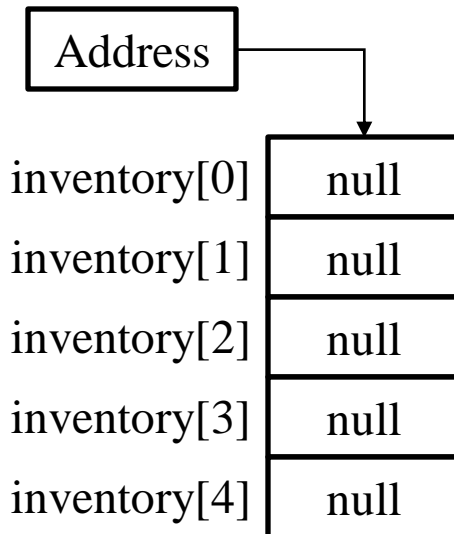
- An array's `length` is a field
  - You ***do not*** write a set of parentheses after its name.
- A `String`'s `length` is a method
  - You write the parentheses after the name of the `String` class's `length` method.

# Arrays of Objects

- 🍷 Since strings are objects, we know that arrays can contain objects.

```
InventoryItem[] inventory = new InventoryItem[5];
```

The `inventory` variable holds the address of an `InventoryItem` array.



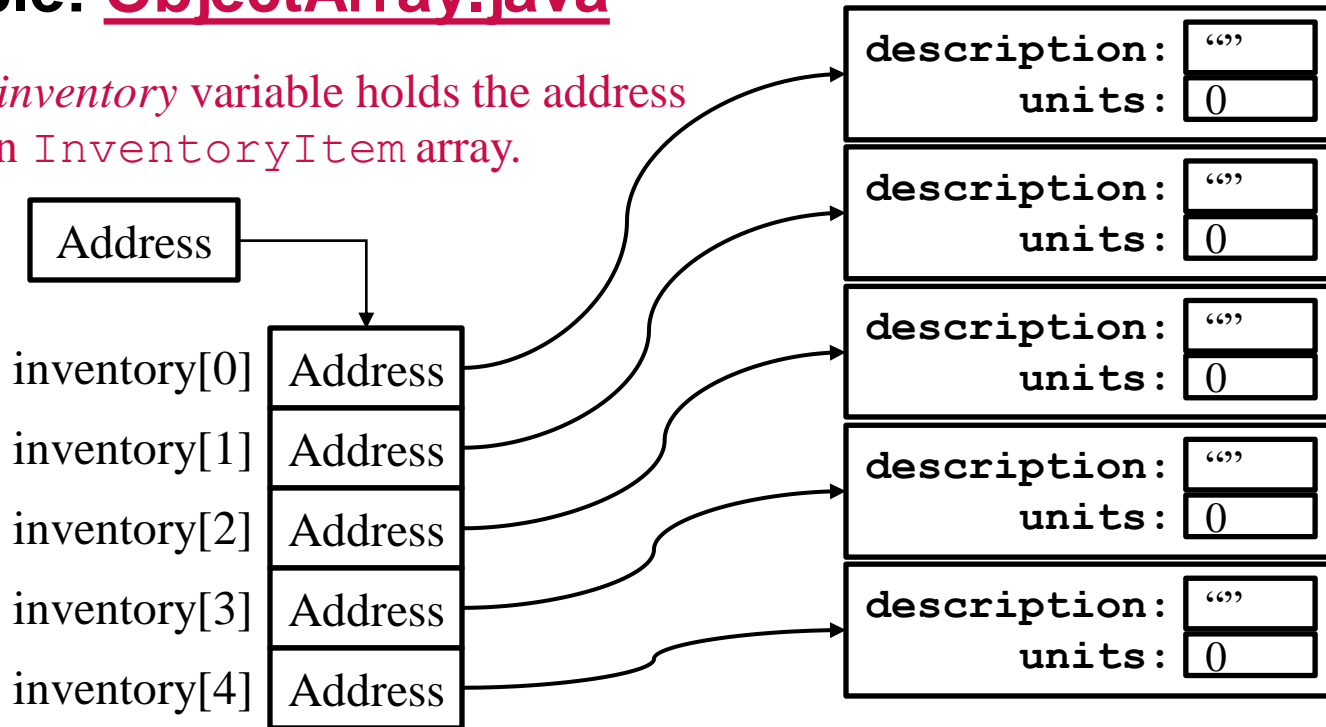
# Arrays of Objects

- Each element needs to be initialized.

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

- Example: ObjectArray.java

The *inventory* variable holds the address of an *InventoryItem* array.









# The Sequential Search Algorithm

- 🍷 A search algorithm is a method of locating a specific item in a larger collection of data.
- 🍷 The *sequential search algorithm* uses a loop to:
  - 🍷 sequentially step through an array,
  - 🍷 compare each element with the search value, and
  - 🍷 stop when
    - 🍷 the value is found or
    - 🍷 the end of the array is encountered.
- 🍷 See example: [SearchArray.java](#)








# Selection Sort

## In a selection sort:

-  The smallest value in the array is located and moved to element 0.
-  Then the next smallest value is located and moved to element 1.
-  This process continues until all of the elements have been placed in their proper order.
-  See example: [SelectionSortDemo.java](#)

# Binary Search

## A binary search:

-  requires an array sorted in ascending order.
-  starts with the element in the middle of the array.
-  If that element is the desired value, the search is over.
-  Otherwise, the value in the middle element is either greater or less than the desired value
-  If it is greater than the desired value, search in the first half of the array.
-  Otherwise, search the last half of the array.
-  Repeat as needed while adjusting start and end points of the search.

 See example: [BinarySearchDemo.java](#)

# Two-Dimensional Arrays

- 🍓 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				
row 3				

# Two-Dimensional Arrays

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

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

Two-dimensional array

Rows Columns

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

# Accessing Two-Dimensional Array Elements

 **When processing the data in a two-dimensional array, each element has two subscripts:**

 one for its row

 one for its column

# Accessing Two-Dimensional Array Elements

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

Address		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

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 `doubles`.

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

Address		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

🍷 Programs that process two-dimensional arrays can do so with nested loops.

🍷 To fill the scores array:

```
for (int row = 0; row < 3; row++)
```

Number of rows, not the largest subscript

```
{
```

```
    for (int col = 0; col < 4; col++)
```

Number of columns, not the largest subscript

```
{
```

```
    System.out.print("Enter a score: ");
```

```
    scores[row][col] = keyboard.nextDouble();
```

```
}
```

```
}
```

keyboard references a Scanner object

# Accessing Two-Dimensional Array Elements

 **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

- ❖ 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}

- ❖ Declares an array with three rows and three columns.

# Initializing a Two-Dimensional Array

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

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

Address		column 0	column 1	column 2
row 0		1	2	3
row 1		4	5	6
row 2		7	8	9

# The length Field

- 🍓 **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

 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.
- 🍓 See example: [Pass2Darray.java](#)

# Ragged Arrays

- ❶ When the rows of a two-dimensional array are of different lengths, the array is known as a *ragged array*.
- ❷ 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][];
```

- ❸ Then create the individual rows.

```
ragged[0] = new int [3];
```

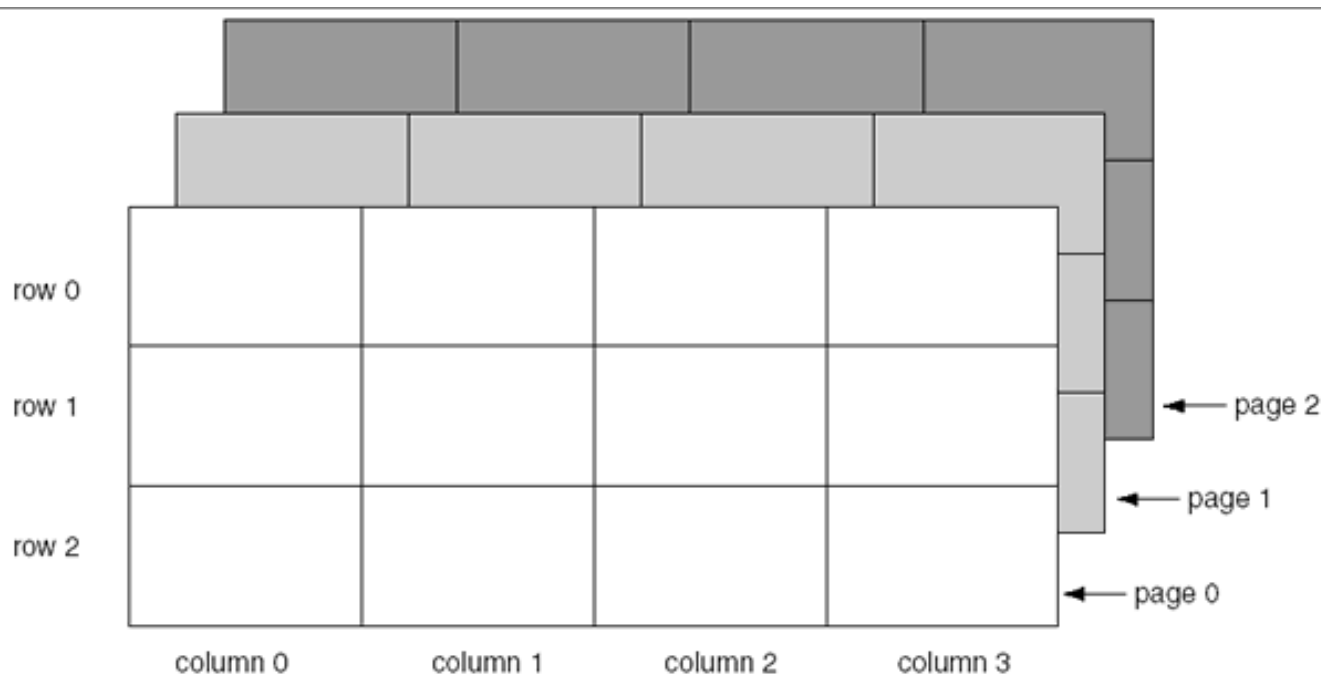
```
ragged[1] = new int [4];
```

```
ragged[2] = new int [5];
```

```
ragged[3] = new int [6];
```

# More Than Two Dimensions

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



# Command-Line Arguments

- 🍷 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

 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`.

# Variable-Length Argument Lists

## Special type parameter – vararg...

-  vararg parameters are actually arrays

-  Examples: [VarArgsDemo1.java](#), [VarargsDemo2.java](#)

```
public static int sum(int... numbers)
{
    int total = 0; // Accumulator
    // Add all the values in the numbers array.
    for (int val : numbers)
        total += val;
    // Return the total.
    return total;
}
```

# 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

🍓 **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
```

# Creating and Using an ArrayList

🍓 **To access items in an ArrayList, use the get method**

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

In this statement 1 is the index of the item to get.

🍓 **Example: ArrayListDemo1.java**

# Creating and Using an ArrayList

 **You can use the enhanced for loop to iterate over each item in an ArrayList.**

```
// Create an ArrayList of names.  
ArrayList<String> nameList = new ArrayList<String>();  
nameList.add("James");  
nameList.add("Catherine");  
nameList.add("Bill");  
  
// Display the items in the ArrayList.  
for (String name : nameList)  
    System.out.println(name);
```

# Using an ArrayList

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

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

This statement yields :

```
[ James, Catherine ]
```

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

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

This statement removes the second item.

- 🍓 **See example: [ArrayListDemo3.java](#)**

# Using an ArrayList

- 🍓 The `ArrayList` class's `add` method with one argument adds new items to the end of the `ArrayList`
- 🍓 To insert items at a location of choice, use the `add` method with two arguments:

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

This statement inserts the `String` "Mary" at index 1

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

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

This statement replaces "Mary" with "Becky"

- 🍓 See example: [ArrayListDemo4.java](#)

# Using an ArrayList

- 🍷 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

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

```
ArrayList<InventoryItem> accountList =  
    ↑ new ArrayList<InventoryItem>();  
    ↑
```

This creates an ArrayList that can hold  
InventoryItem objects.



# Using an ArrayList

```
// Create a list to hold InventoryItem objects.
ArrayList list = new ArrayList();

// Add three InventoryItem objects to the ArrayList.
list.add(new InventoryItem("Nuts", 100));
list.add(new InventoryItem("Bolts", 150));
list.add(new InventoryItem("Washers", 75));

// Display each item.
for (int index = 0; index < list.size(); index++)
{
    InventoryItem item = (InventoryItem)list.get(index);
    System.out.println("Item at index " + index +
        "\nDescription: " + item.getDescription() +
        "\nUnits: " + item.getUnits());
}
```

See: [ArrayListDemo6.java](#)