Lecture 3 - Java Fundamentals

CS260 – Android App Development

Paul Cao

Review

- Java variable types (especially String)
- Operators and expressions
- Input and output format
- Dialog boxes
- Flow control

Plan for today

- Arrays in Java
- File I/O
- methods for class

Writing Text To a File

- To open a file for text output you create an instance of the `PrintWriter` class.

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

Pass the name of the file that you wish to open as an argument to the `PrintWriter` constructor.

Warning: if the file already exists, it will be erased and replaced with a new file.

The `PrintWriter` Class

- The `PrintWriter` class allows you to write data to a file using the `print` and `println` methods, as you have been using to display data on the screen.
- Just as with the `System.out` object, the `println` method of the `PrintWriter` class will place a newline character after the written data.
- The `print` method writes data without writing the newline character.

The `PrintWriter` Class

Open the file.

```
PrintWriter outputFile = new PrintWriter("Names.txt");
outputFile.println("Chris");
outputFile.println("Kathryn");
outputFile.println("Jean");
outputFile.close();
```

Close the file.

Write data to the file.

The `PrintWriter` Class

- To use the `PrintWriter` class, put the following `import` statement at the top of the source file:

```
import java.io.*;
```

- See example: FileWriteDemo.java

Exceptions

- When something unexpected happens in a Java program, an *exception* is thrown.
- The method that is executing when the exception is thrown must either handle the exception or pass it up the line.
- To pass it up the line, the method needs a `throws` clause in the method header.

Exceptions

- To insert a `throws` clause in a method header, simply add the word *throws* and the name of the expected exception.
- `PrintWriter` objects can throw an `IOException`, so we write the `throws` clause like this:

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

Appending Text to a File

- To avoid erasing a file that already exists, create a `FileWriter` object in this manner:

```
FileWriter fw =
     new FileWriter("names.txt", true);
```

- Then, create a `PrintWriter` object in this manner:

```
PrintWriter fw = new PrintWriter(fw);
```

Specifying a File Location

- On a Windows computer, paths contain backslash (\) characters.
- Remember, if the backslash is used in a string literal, it is the escape character so you must use two of them:

```
PrintWriter outFile =
     new PrintWriter("A:\\PriceList.txt");
```

Specifying a File Location

- This is only necessary if the backslash is in a string literal.
- If the backslash is in a `String` object then it will be handled properly.
- Fortunately, Java allows Unix style filenames using the forward slash (/) to separate directories:

```
PrintWriter outFile = new
    PrintWriter("/home/rharrison/names.txt");
```

Reading Data From a File

- You use the `File` class and the `Scanner` class to read data from a file:

Pass the name of the file as an argument to the `File` class constructor.

```
File myFile = new File("Customers.txt");
Scanner inputFile = new Scanner(myFile);
```

Pass the `File` object as an argument to the `Scanner` class constructor.

Reading Data From a File

```
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter the filename: ");
String filename = keyboard.nextLine();
File file = new File(filename);
Scanner inputFile = new Scanner(file);
```

- The lines above:
  - Creates an instance of the `Scanner` class to read from the keyboard
  - Prompt the user for a filename
  - Get the filename from the user
  - Create an instance of the `File` class to represent the file
  - Create an instance of the `Scanner` class that reads from the file

Reading Data From a File

- Once an instance of `Scanner` is created, data can be read using the same methods that you have used to read keyboard input (`nextLine`, `nextInt`, `nextDouble`, etc).

```
// Open the file.
File file = new File("Names.txt");
Scanner inputFile = new Scanner(file);
// Read a line from the file.
String str = inputFile.nextLine();
// Close the file.
inputFile.close();
```

Exceptions

- The `Scanner` class can throw an `IOException` when a `File` object is passed to its constructor.
- So, we put a `throws IOException` clause in the header of the method that instantiates the `Scanner` class.
- See Example: ReadFirstLine.java

Detecting The End of a File

- The `Scanner` class's `hasNext()` method will return true if another item can be read from the file.

```
// Open the file.
File file = new File(filename);
Scanner inputFile = new Scanner(file);
// Read until the end of the file.
while (inputFile.hasNext())
{
   String str = inputFile.nextLine();
   System.out.println(str);
}
inputFile.close();// close the file when done.
```

Detecting the End of a File

- See example: FileReadDemo.java

Exercise

Write a program that asks the user for the name of a file. Your program should read the content of this file and output the file content with every character changed to upper case. And output the upper case texts to a file called “results.txt”

- Hint: You don't have to worry about the character by character input. Just read in line by line and change the whole line into upper case using the toUpperCase() member function from the String class

Introduction to Arrays

- Arrays in every language is similar
- 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.

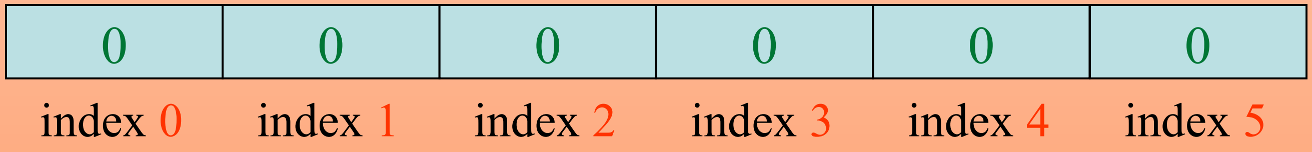
Creating Arrays

- 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];
```



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
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; //pronounced "numbers sub zero"
```

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,
  - etc.

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.

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

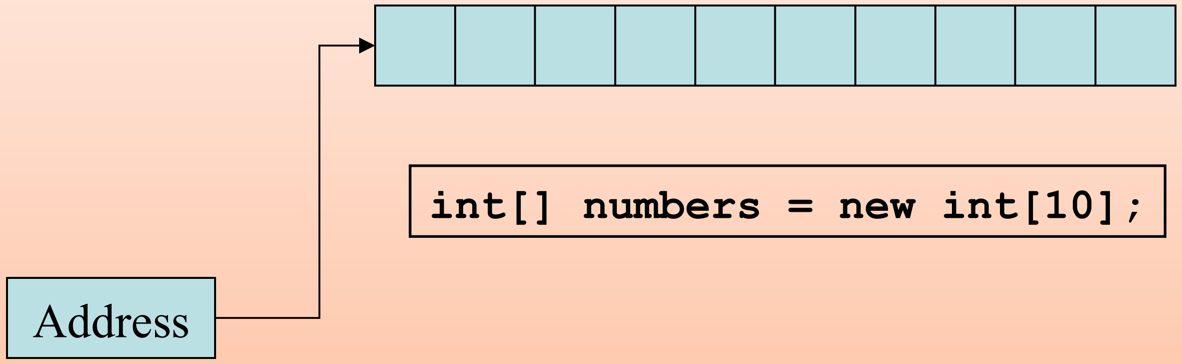
Reassigning Array References

- 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

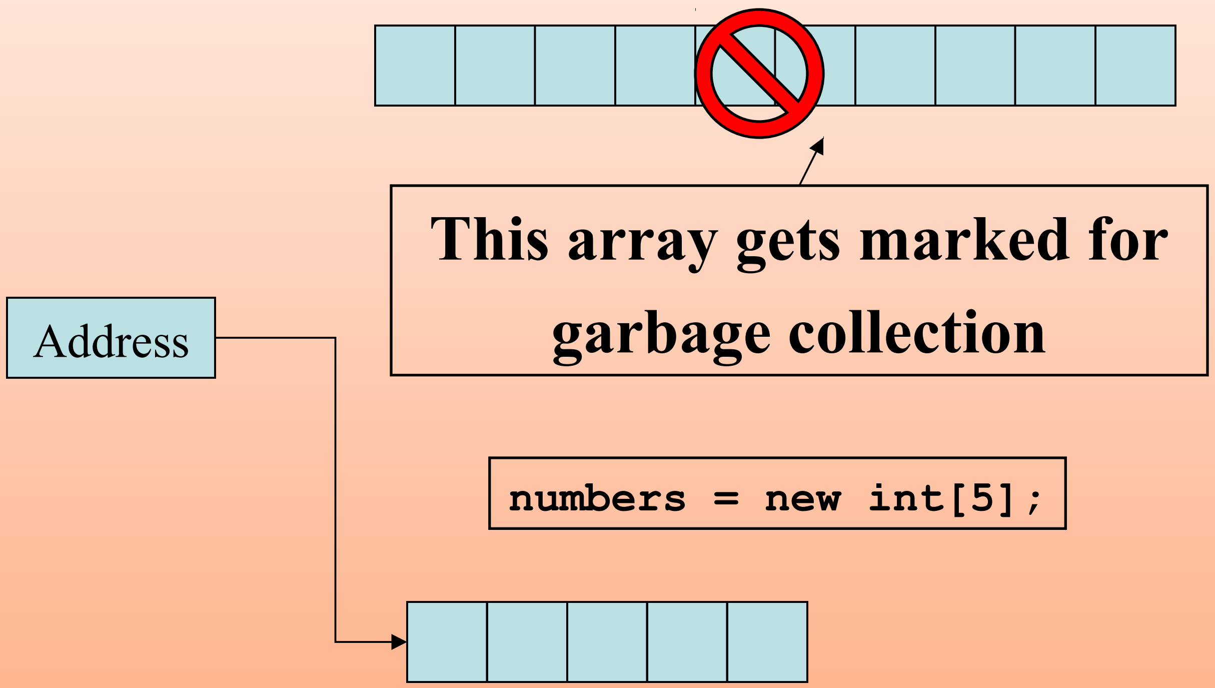


```
int[] numbers = new int[10];
```

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

Reassigning Array References

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

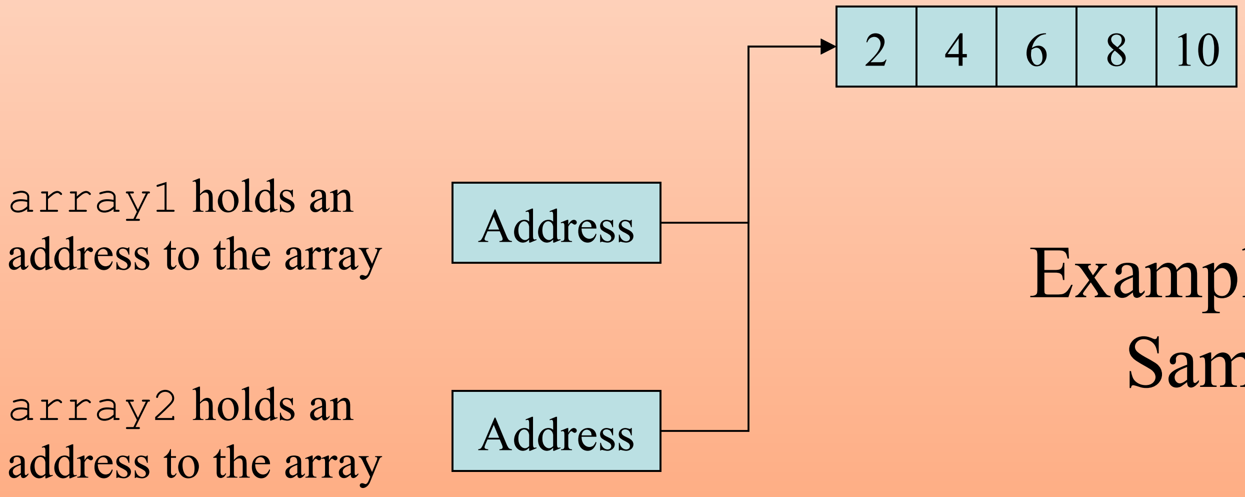


```
numbers = new int[5];
```

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.
```


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

- 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`.

- Or you can simply call the clone method of the array object

Comparing Arrays

- The == operator determines only whether array references point to the same array object. You have to compare the contents.

```
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.");
```

Exercise

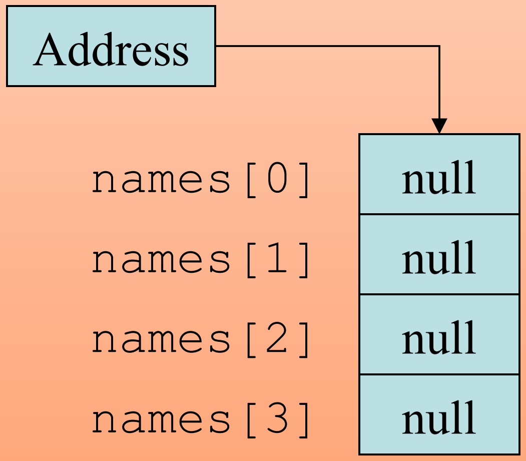
Write a program that reads in two double arrays from user. The user will specify the size of the two arrays and also enter the data into each array via keyboard.

Your program is supposed to output the content of each array and in the end, state whether the contents of the two arrays are the same or not.

`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.

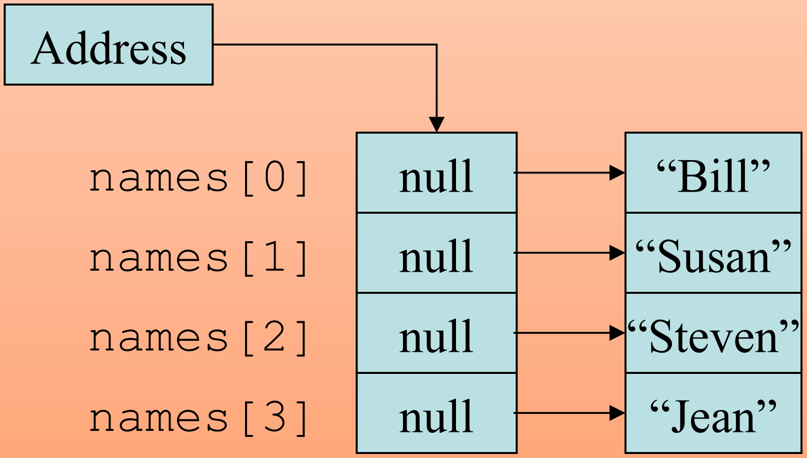


`String` Arrays

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

```
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);
```

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 do write the parentheses after the name of the `String` class’s `length` method.

Two-Dimensional Arrays

- 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];
```

two dimensional array

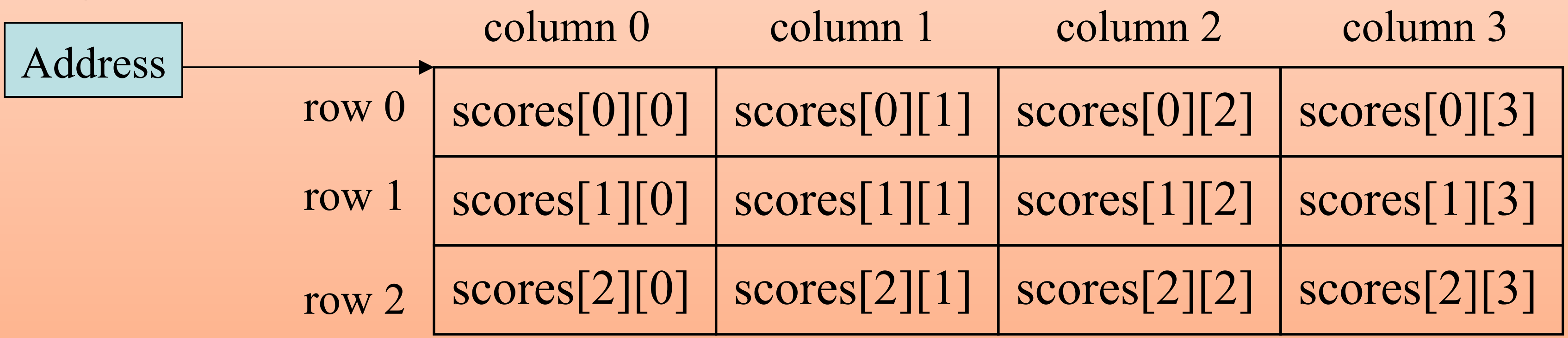
rows

columns

- 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

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



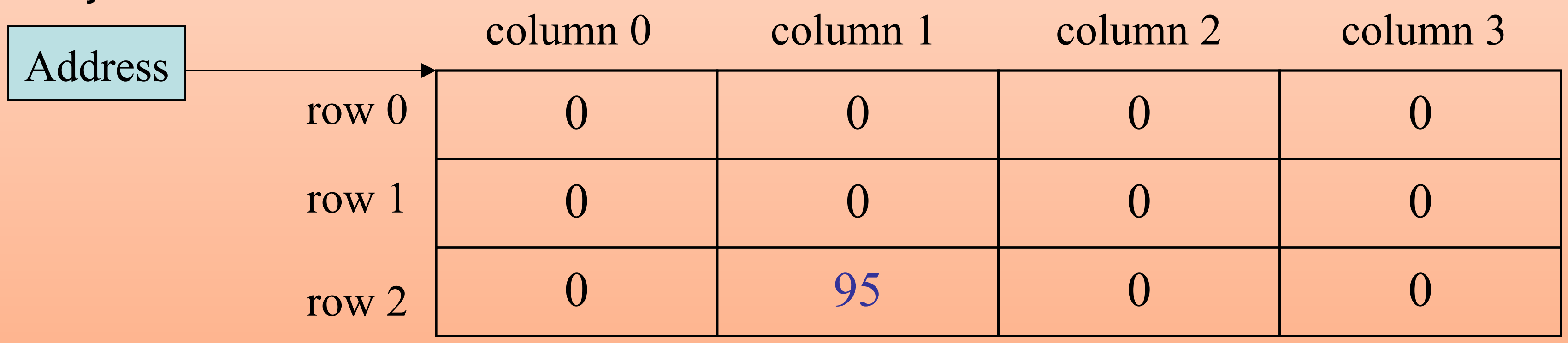
	column 0	column 1	column 2	column 3
row 0	scores[0][0]	scores[0][1]	scores[0][2]	scores[0][3]
row 1	scores[1][0]	scores[1][1]	scores[1][2]	scores[1][3]
row 2	scores[2][0]	scores[2][1]	scores[2][2]	scores[2][3]

Accessing Two-Dimensional Array Elements

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

- 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

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];
```

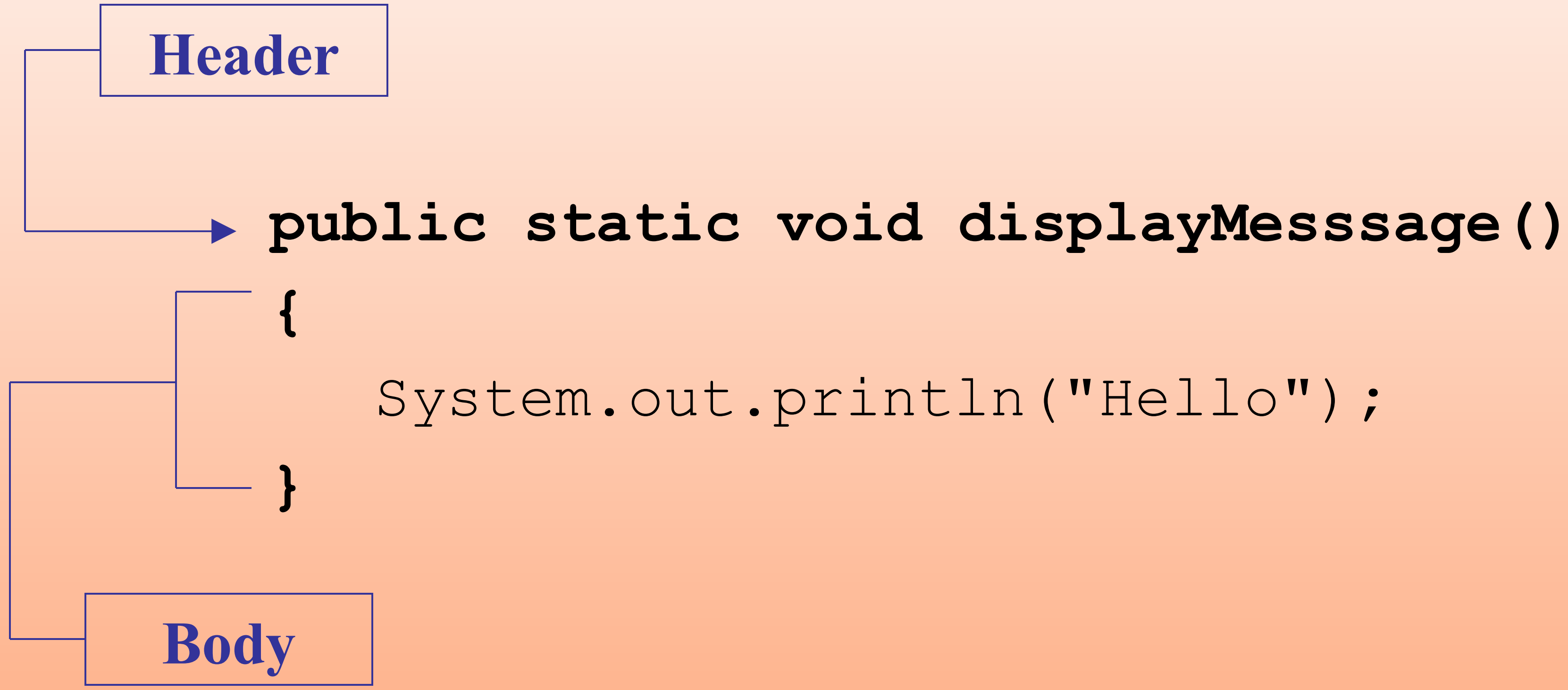
Exercise

Declare a 2D array of size 10 by 5. Then read in all the data from a file called data.txt (available on angel). In the end, output the whole array and the sum of every element in the array on the console.

Methods == Member Functions

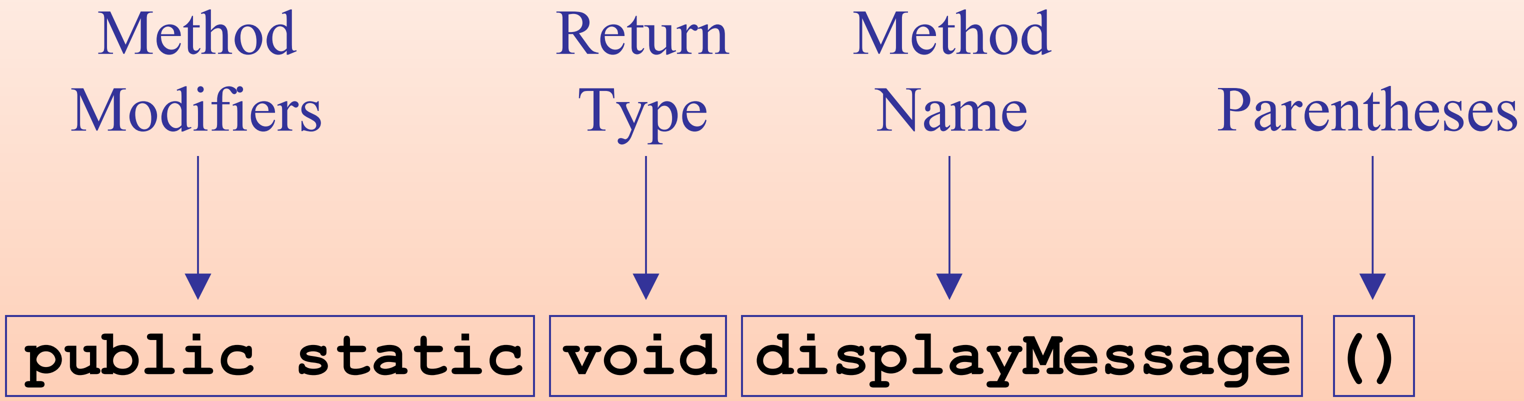
- Methods are commonly used to break a problem down into small manageable pieces. They are basically member functions in other languages.
- No need to do prototype anymore. Just the header and body directly.
- Void can be used as return value. If no parameter list is needed, just use (), not (void)

Two Parts of Method Declaration



```
public static void displayMesssage()
{
    System.out.println("Hello");
}
```

Parts of a Method Header



```
public static void displayMessage ()
{
   System.out.println("Hello");
}
```

Calling a Method

- A method executes when it is called.
- The `main` method is automatically called when a program starts, but other methods are executed by method call statements.

```
    displayMessage();
```

- It is the same as in other languages to call methods.
- Examples: CreditCard.java

Documenting Methods

- A method should always be documented by writing comments that appear just before the method's definition.
- The comments should provide a brief explanation of the method's purpose.
- The documentation comments begin with /** and end with */.

Passing Arguments to a Method

- Values that are sent into a method are called arguments.

```
System.out.println("Hello");
number = Integer.parseInt(str);
```

- The data type of an argument in a method call must correspond to the variable declaration in the parentheses of the method declaration. The parameter is the variable that holds the value being passed into a method.
- By using parameter variables in your method declarations, you can design your own methods that accept data this way. See example: PassArg.java

Passing 5 to the **`displayValue`** Method

```
displayValue(5);
```

The argument 5 is copied into the parameter variable **`num`**.

```
public static void displayValue(int num)
{
  System.out.println("The value is " +
  num);
}
```

The method will display **`The value is 5`**

Argument and Parameter Data Type Compatibility

- When you pass an argument to a method, be sure that the argument's data type is compatible with the parameter variable's data type.
- Avoid type mismatch!

Passing Multiple Arguments

The argument 5 is copied into the **num1** parameter.

The argument 10 is copied into the **num2** parameter.

NOTE: Order matters!

```
ShowSum(5.0, 10.0);

public static void showSum(double num1, double num2)
{
   double sum;    //to hold the sum
   sum = num1 + num2;
   System.out.println("The sum is " + sum);
}
```

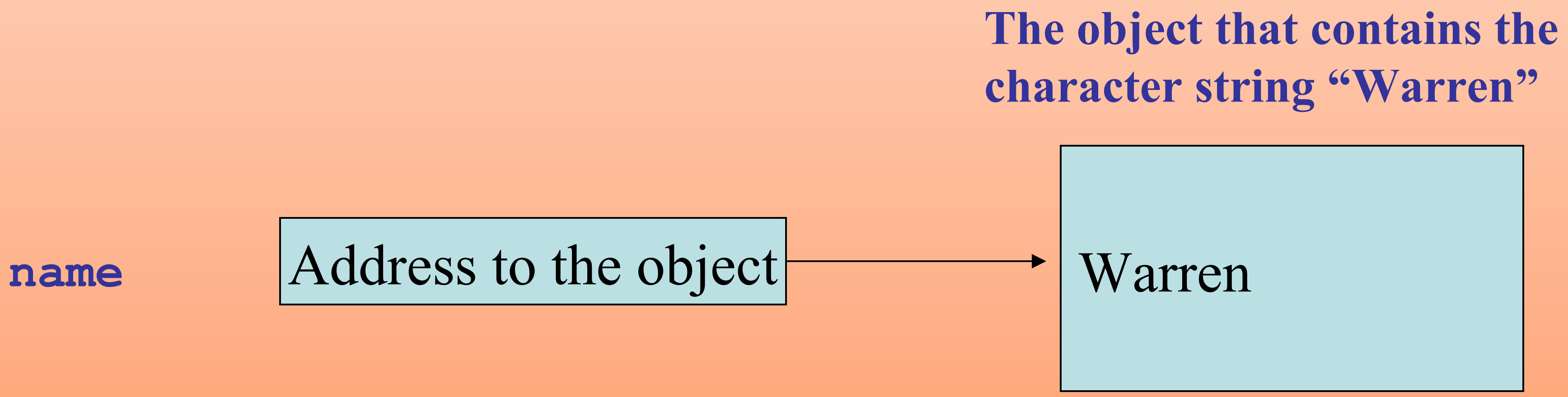
Arguments are Passed by Value

- In Java, all arguments of the primitive data types are *passed by value*, which means that only a copy of an argument’s value is passed into a parameter variable.
- A method’s parameter variables are separate and distinct from the arguments that are listed inside the parentheses of a method call.
- If a parameter variable is changed inside a method, it has no affect on the original argument.
- See example: PassByValue.java

Classes

- A reference variable contains the address of an object.

```
String name = "Warren";
```



Passing Object References to a Method

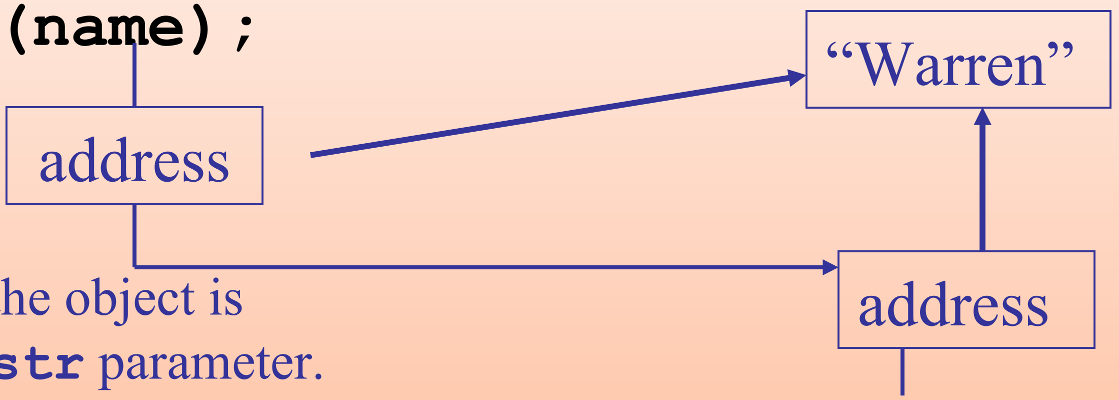
- A class type variable does not hold the actual data item that is associated with it, but holds the memory address of the object. A variable associated with an object is called a reference variable.

- When an object such as a `String` is passed as an argument, it is actually a reference to the object that is passed.

Passing a Reference as an Argument

Both variables reference the same object

```
showLength(name);
```



The address of the object is copied into the **`str`** parameter.

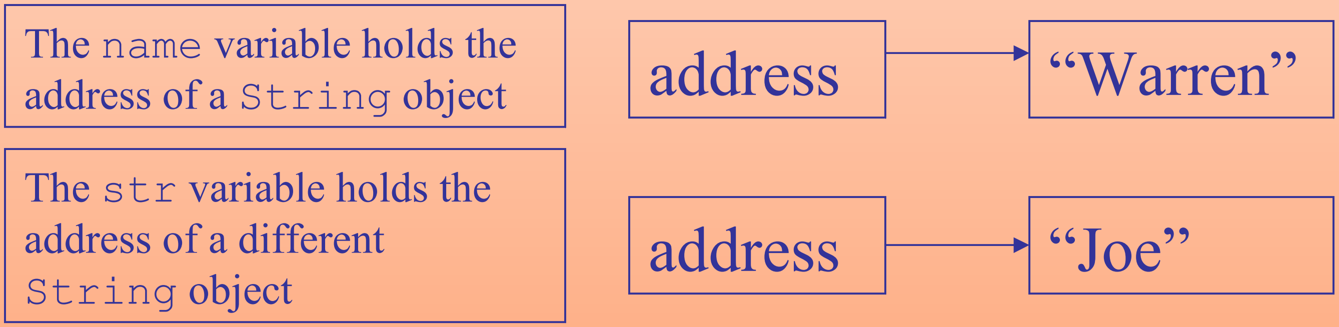
```
public static void showLength(String str)
{
  System.out.println(str + " is " +
  str.length()        + " characters long.");
  str = "Joe" // see next slide
}
```

`String`s are Immutable Objects

- `String`s are immutable objects, which means that they cannot be changed. When the line

  ```
  str = "Joe";
  ```

  is executed, it cannot change an immutable object, so creates a new object.


The name variable holds the address of a String object
address
"Warren"
The str variable holds the address of a different String object
address
"Joe"

- See example: PassString.java

`@param` Tag in Documentation Comments

- You can provide a description of each parameter in your documentation comments by using the `@param` tag.
- General format

```
@param parameterName Description
```

- See example: TwoArgs2.java
- All `@param` tags in a method's documentation comment must appear after the general description. The description can span several lines.

More About Local Variables

- A local variable is declared inside a method and is not accessible to statements outside the method.
- Different methods can have local variables with the same names because the methods cannot see each other's local variables.
- A method's local variables exist only while the method is executing. When the method ends, the local variables and parameter variables are destroyed and any values stored are lost.
- Local variables are not automatically initialized with a default value and must be given a value before they can be used.

Returning a Value from a Method

- Data can be passed into a method by way of the parameter variables. Data may also be returned from a method, back to the statement that called it.

```
    int num = Integer.parseInt("700");
```

- The string “700” is passed into the `parseInt` method.
- The `int` value 700 is returned from the method and assigned to the `num` variable.

Defining a Value-Returning Method

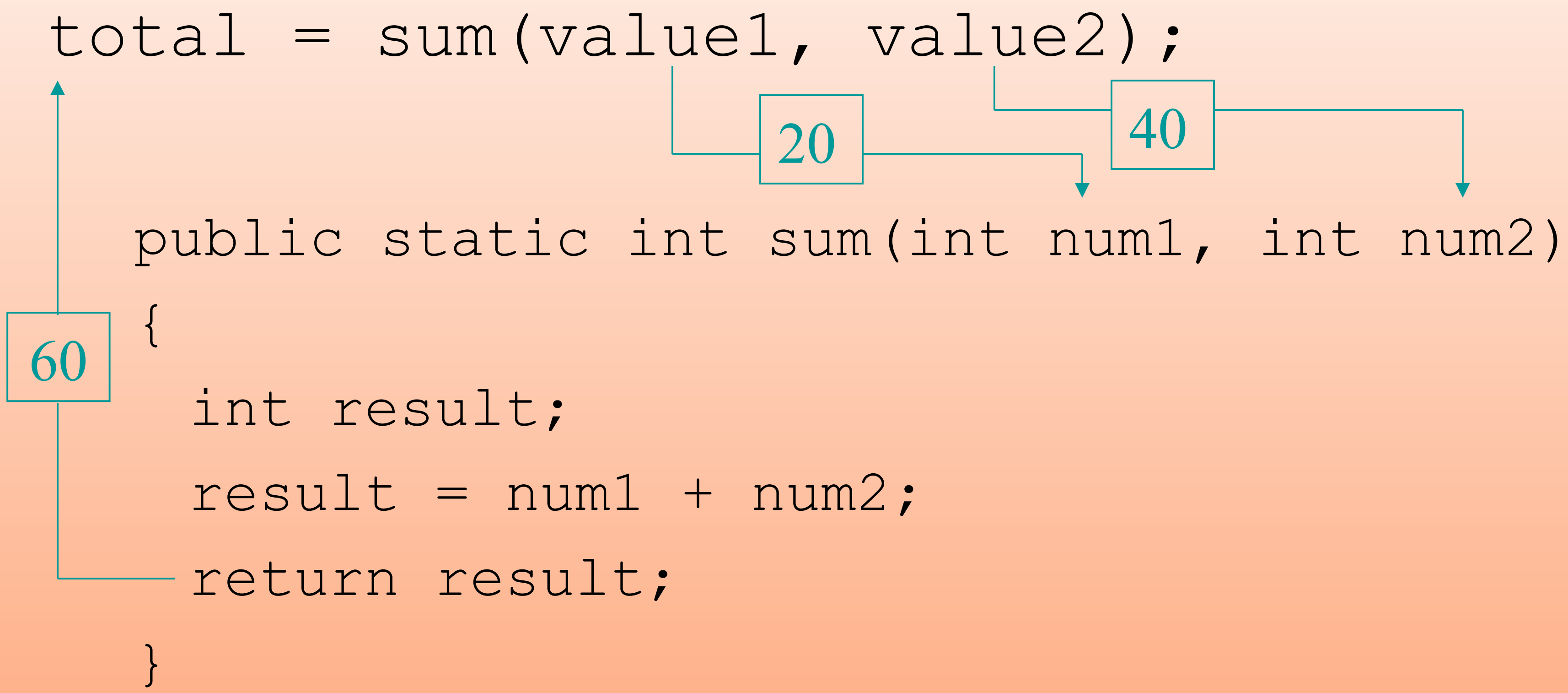
```
public static int sum(int num1, int num2)
{
   int result;
   result = num1 + num2;
   return result;
}
```

Return type

The `return` statement causes the method to end execution and it returns a value back to the statement that called the method.

This expression must be of the same data type as the return type

Calling a Value-Returning Method



```
total = sum(value1, value2);

   public static int sum(int num1, int num2)
   {
     int result;
     result = num1 + num2;
     return result;
   }
```

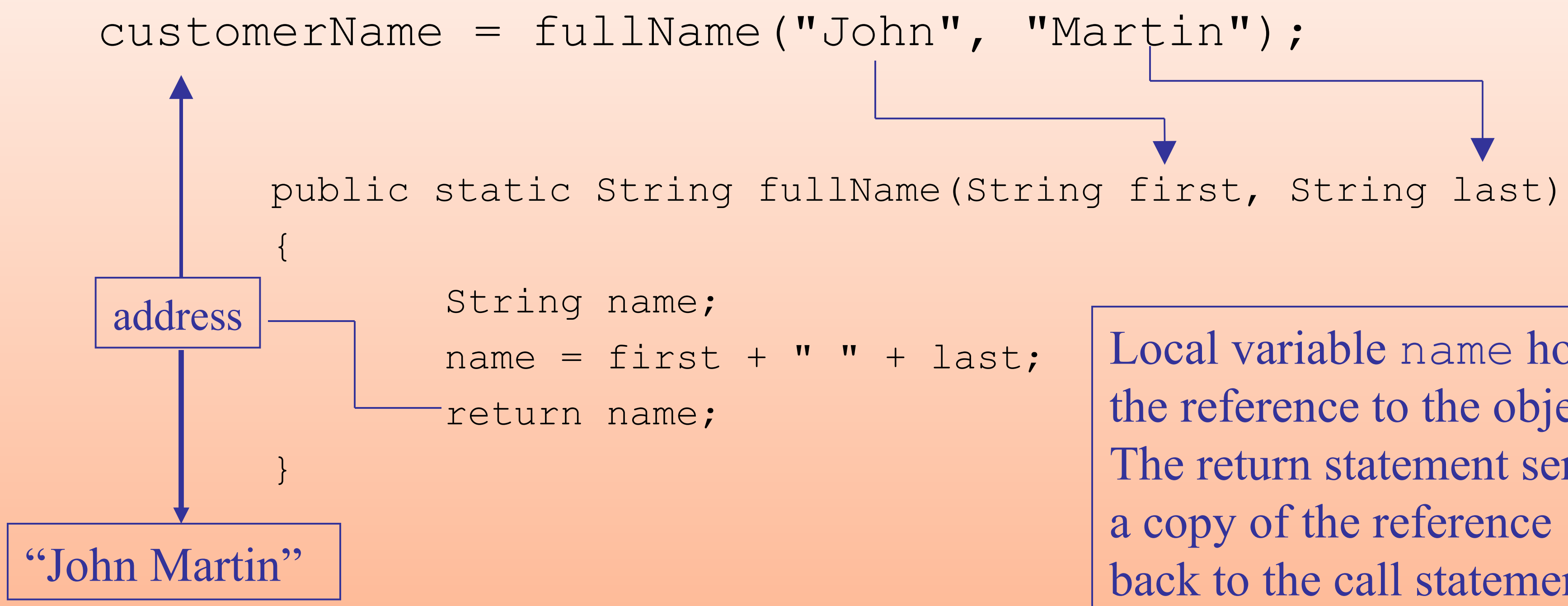
`@return` Tag in Documentation Comments

- You can provide a description of the return value in your documentation comments by using the `@return` tag.
- General format

```
      @return Description
```

- See example: ValueReturn.java
- The `@return` tag in a method's documentation comment must appear after the general description. The description can span several lines.

Returning a Reference to a `String` Object



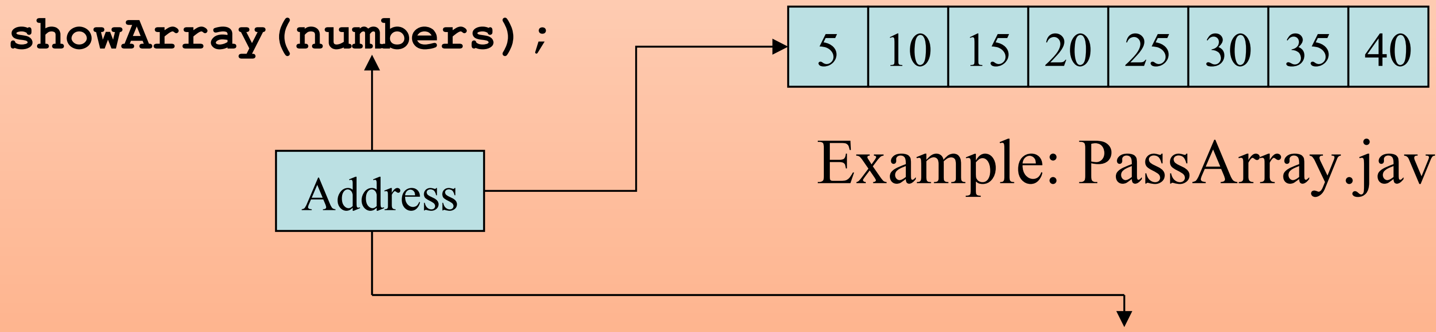
```
customerName = fullName("John", "Martin");

public static String fullName(String first, String last)
{
        String name;
        name = first + " " + last;
        return name;
}
```

Local variable `name` holds the reference to the object. The return statement sends a copy of the reference back to the call statement and it is stored in `customerName`.

Passing Arrays as Arguments

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



Example: PassArray.java

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

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

Exercise

Write a Java program that defines the following methods

- findMax: accepts a double array and returns the max of that array
- square: accepts a double array and returns a double array where the new array's element is the square of the original array. For example, if the old array is -.5 .3 .4, the new array's content is .25 .09 and .16.
- Test your program using the main method