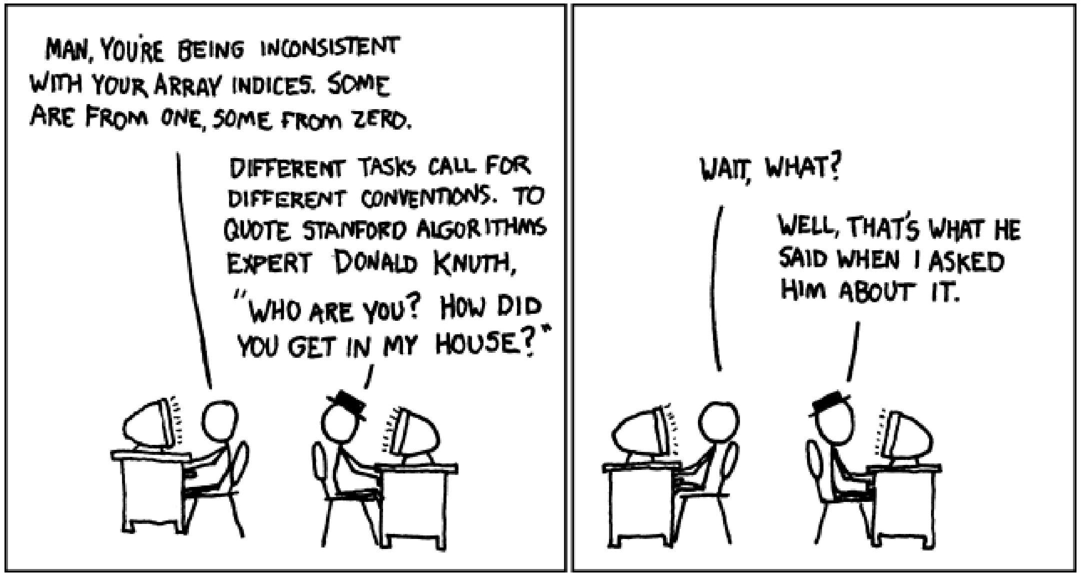

MAN, YOU'RE BEING INCONSISTENT WITH YOUR ARRAY INDICES. SOME ARE FROM ONE, SOME FROM ZERO.
DIFFERENT TASKS CALL FOR DIFFERENT CONVENTIONS. TO QUOTE STANFORD ALGORITHMS EXPERT DONALD KNUTH, "WHO ARE YOU? HOW DID YOU GET IN MY HOUSE?"
WAIT, WHAT?
WELL, THAT'S WHAT HE SAID WHEN I ASKED HIM ABOUT IT.

Arrays

Arrays

- An *array* is an ordered list of values.
- An array has a single name and holds several values.
  - Each value has a numeric index.
  - An array of size *n* is indexed from 0 to *n-1*.
- Arrays store elements of the same type.
  - Arrays can hold primitives or objects.

Arrays are Objects

- Remember that everything in Java is either a primitive or an object.
- Arrays are objects!
  - The variable name of an array holds a *pointer* to the place in memory where the elements are stored.

Declaring an Array

- To declare an array, you specify:
  - The type of the elements
  - Square brackets
  - The name of the array
- Examples:
  - `int[] scores;`
  - `String[] names;`
  - `Course[] catalog;`

`scores` is an object

the type of `scores` is an `int` array

each element in scores is a primitive

Declaring an Array

- To declare an array, you specify:
  - The type of the elements
  - Square brackets
  - The name of the array
- Examples:
  - `int[] scores;`
  - `String[] names;`
  - `Course[] catalog;`

`catalog` is an object

the type of `catalog` is a Course array

each element in scores is an object

Declaring an Array

- Note that declaring an array does not specify how many elements it can hold. It also doesn’t create any elements inside the array.

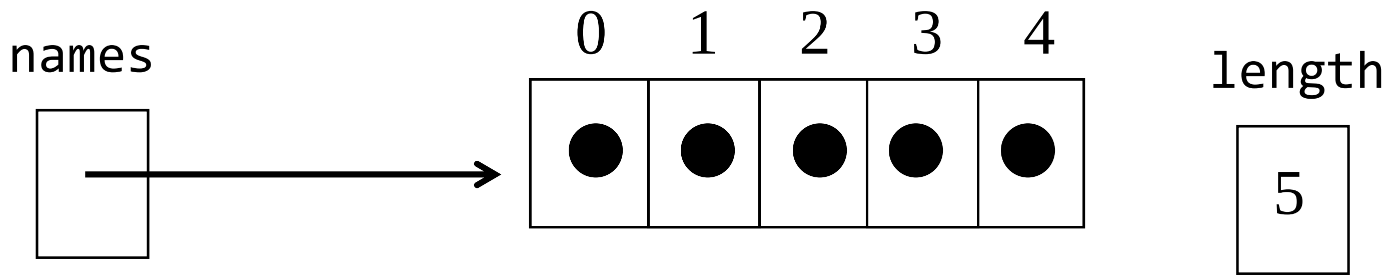
Initializing Arrays

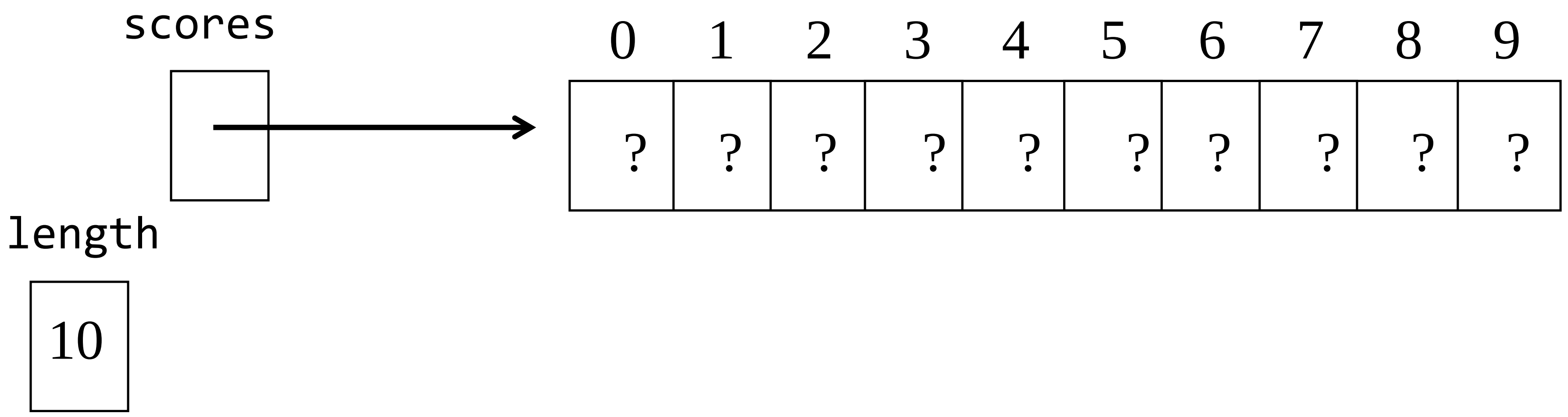
- Because arrays are objects, they need to be initialized with the `new` operator.
- To initialize, specify:
  - `new` operator
  - the type
  - the size of the array inside square brackets
- Examples:
  - `names = new String[5];`
  - `scores = new int[10];`
  - `catalog = new Course[100];`

Initializing Arrays

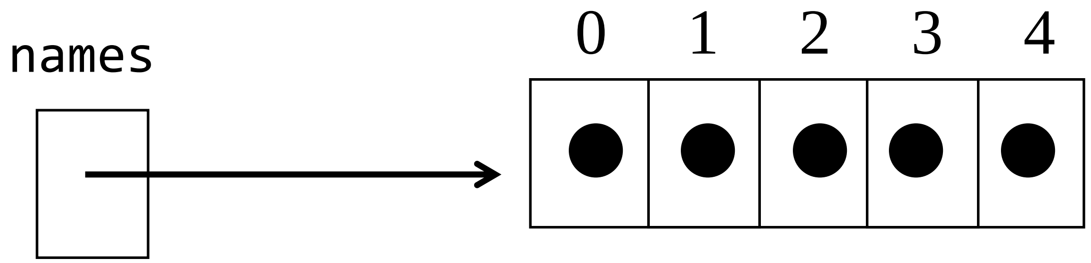
- Initializing the array sets up a place in memory.
  - there are no elements in the array yet!
- It also sets up an instance variable called `length`.
  - `length` holds the number of elements that can be stored in the array.
  - The largest index in the array will be… what?

Initializing Arrays



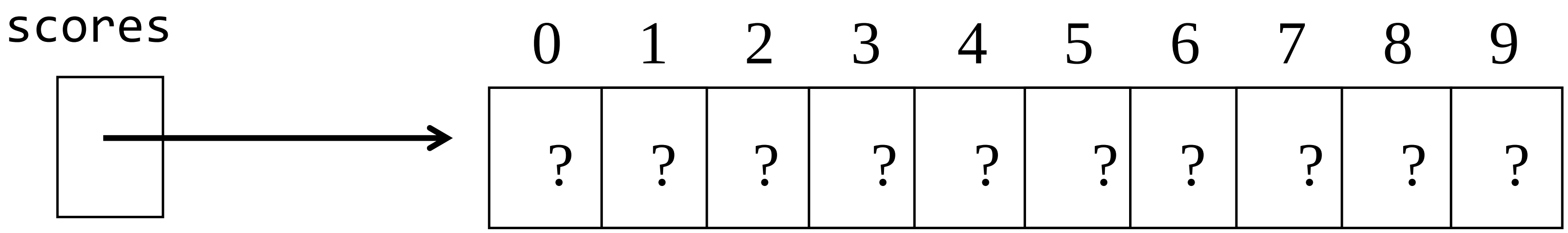


Initializing Arrays

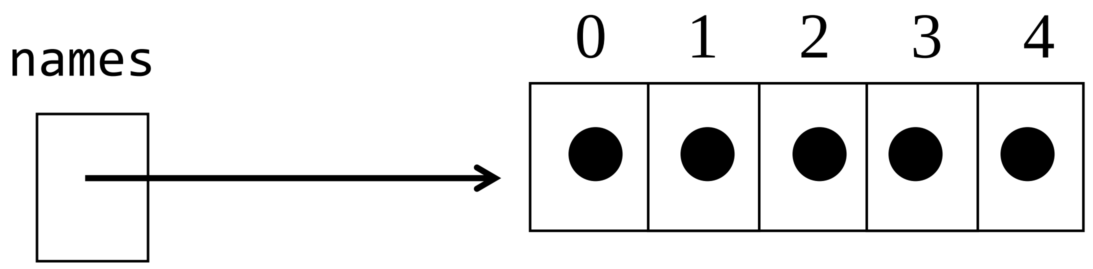


names is an object

the names array will hold objects

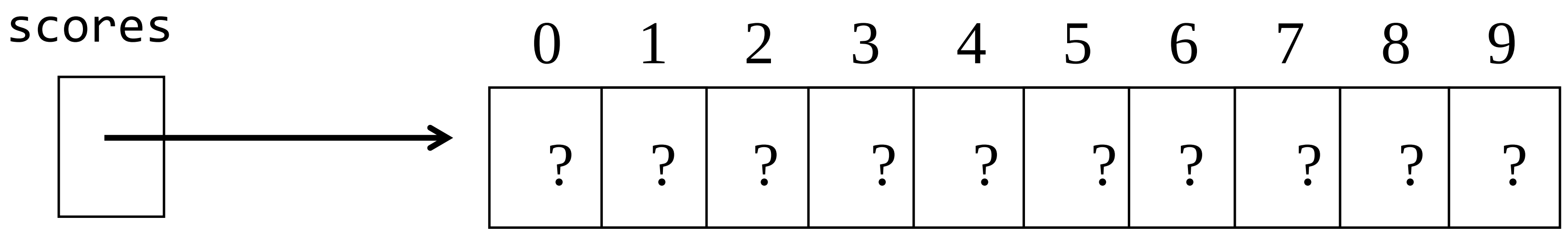


Initializing Arrays



scores is an object

the scores array will hold primitives



Declaring and Initializing

- Just like with other variables, you can declare an initialize separately or in one line.
- Examples:
  - `double[] costs = new double[100];`
  - `boolean[] flags = new boolean[5];`

Practice

- Write code to declare and initialize:
  - An array that will store the first 4 prime numbers
  - An array that will store three names
  - An array that will store the possible letter grades A, B, C, D, and F

Array Size

- The size of an array cannot be changed once it is set during initialization.
- You can change what is stored in the array, but you **cannot** change *how many* elements can be stored in that contiguous space in memory.

Array Size

- You can always find out how many elements an array can hold by accessing the length variable.
- Examples:
  - `System.out.println("There can be " + catalog.length + " courses offered.");`
- length is an instance variable, not a method
  - It's accessed through the dot operator, but there are no parentheses.
- length returns how many elements the array *can* hold, not how many elements are in the array
  - Capacity, not actual size!

The `length` Constant

- Caution: `length` holds the number of elements, NOT the largest index!
  - The largest index is `length-1`
- This is a common source of off-by-one errors.

Accessing Array Elements

- Array elements are accessed by putting the index in square brackets after the array name.

```
arrayName[index]
```

- The type of arrayName is an object- all arrays are objects!
- The type of arrayName[index] is whatever type the array holds (int, String, etc.).

Assigning Values to an Array

- To assign a value to a space in the array, the array[index] goes on the left of the equals sign.
  - The left side is array[index].
  - The right side is a value.
- As always, evaluate the right side first then assign to the variable on the left.
  - It just happens that the variable doesn’t have its own name- it’s just the name of the array at a specific index.

Assigning Values to an Array

- Examples:
  - `names[0] = "Alice";`
  - `names[1] = "Bob";`

  - `scores[5] = 99;`
  - `scores[3] = 100 / 2;`

  - `catalog[87] = new Course("Intro to CS", 1, true);`

Using Values in an Array

- To use a value already stored in the array, the array[index] would go somewhere on the right of the equals sign.
- The elements inside of an array can be used *anywhere* you would expect a variables of that type.
  - As a parameter
  - As an invoking object

Using Values in an Array

- Examples:
  - `System.out.println(names[4]);`
  - `totalScore += scores[3];`
  - `audioItems[1].playSample();`

Practice

- Write code to fill your arrays:
  - An array that stores the first 4 prime numbers
  - An array that stores three names
  - An array that stores the possible letter grades A, B, C, D, and F
- Write code to print the first and last element in each of these arrays.

Initializer Lists

- There is one other way to both declare and fill an array in one step.
- To do this, you follow the declaration by curly brackets that contain the values separated by commas.
- When using an initializer list, you do not use the new operator or specify a size.
  - The size is the number of elements you list.
- An initializer list can only be used when you first declare an array.

Initializer Lists

- Example:
  - `int[] counts = {4, 6, 7};`
  - `String[] directions = {"North", "South", "East", "West"};`

- What is `counts.length`?
- What is `directions.length`?

Practice

- Comment out and rewrite your creation statements using initializer lists.

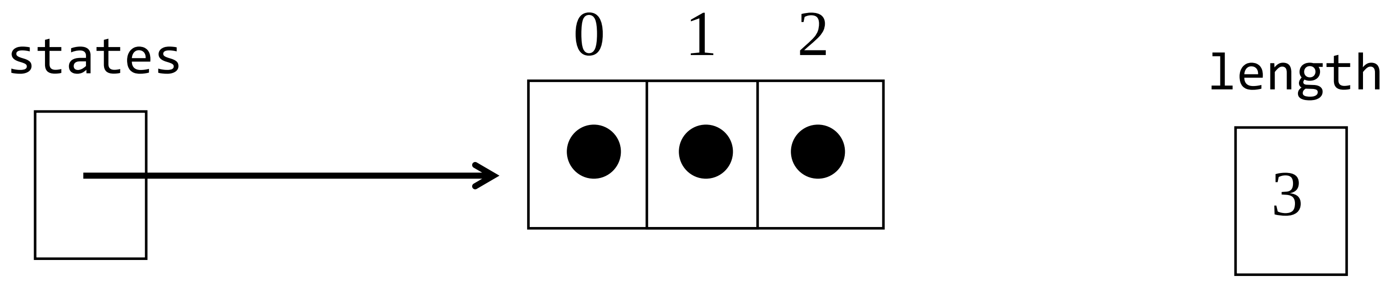
Array Example

- Trace what happens in memory and what is printed with each line.

```
String[] states = new String[3];
System.out.println(states [1]);
System.out.println(states [3]);
states = {“California", “New York", “Iowa"};
```

Array Example

```
String[] states = new String[3];
```

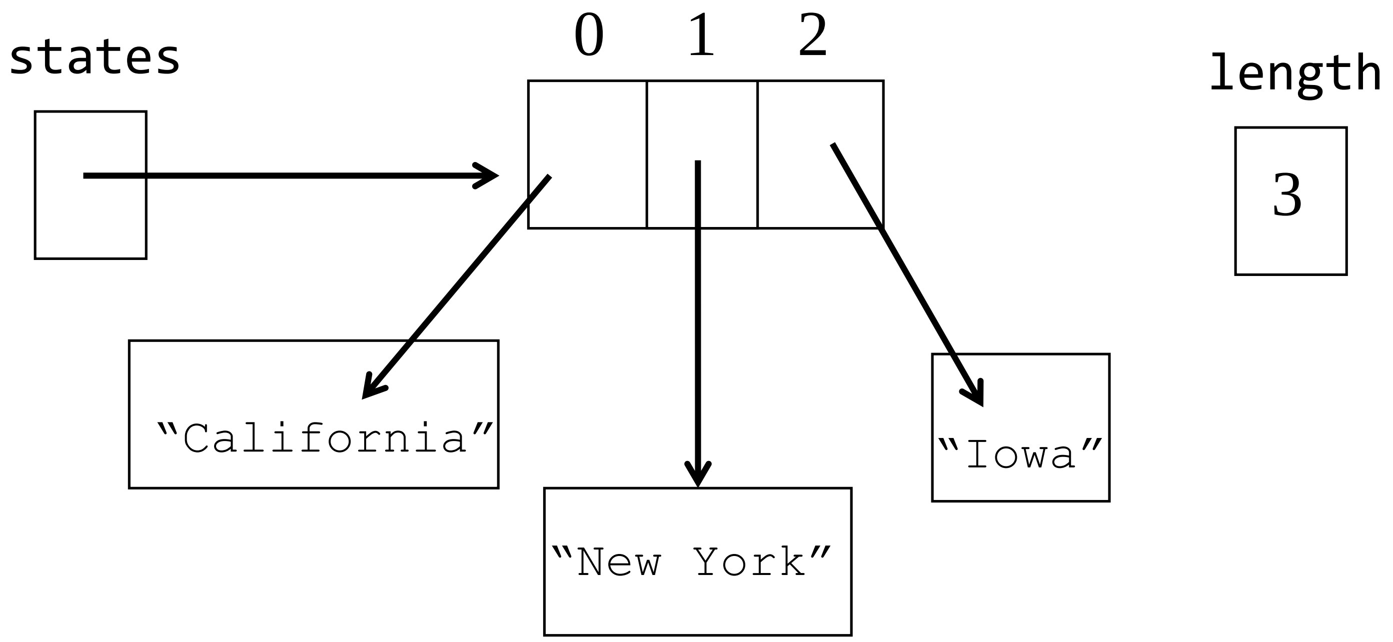


Array Example

```
states[0] = “California”;
states[1] = “New York”;
states[2] = “Iowa”;
states[3] = “Alaska”;
```

Array Example

```
states[0] = “California”;
states[1] = “New York”;
states[2] = “Iowa”;
```



Array Example

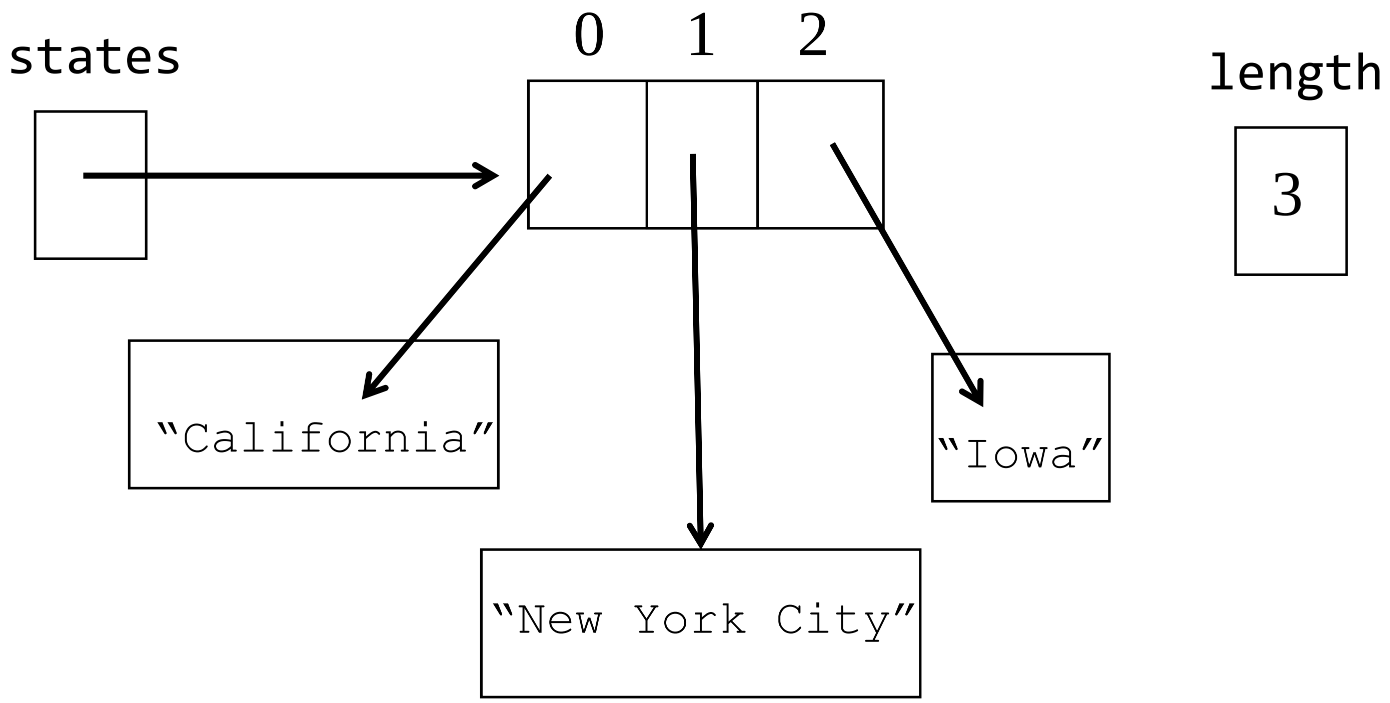
```
states[0].toUpperCase();
System.out.println(states[0].toUpperCase());

states[1] = states[1] + “City”;
System.out.println(states[1]);
```

Array Example

```
states[0].toUpperCase();
System.out.println(states[0].toUpperCase());

states[1] = states[1] + "City";
System.out.println(states[1]);
```



Accessing Arrays: `for` Loops

- You can access each element of an array with a for-loop.
- for-loops can be used to initialize, retrieve, or modify elements.

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

for(int i=0; i<nums.length; i++)
    nums[i] = i*5;
```

initialize

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

retrieve

```
for(int i=0; i<nums.length; i++)
    if(i%2==0)
          nums[i] *=2;
```

modify

Bounds Checking

- An index of an array is in the range 0 to n-1.
- If an array index is out of bounds, Java throws an `ArrayIndexOutOfBoundsException`
  - This is a nasty run-time error… avoid it like the plague!

- It is easy to make off-by-one errors.

```
int codes = new int[100];
for(int index = 0; index <=100; index++)
        codes[index] = index*5;
```

problem

Accessing Arrays: `for` Loops

- Best Practice: use `< arrayName.length` as the upper bound in your for-loop.
  - Avoid hard-coding numbers into your code!

Practice

- Write code to create:
  - An array that stores the numbers from 1 to 100
  - An array that stores the first 100 multiples of 3.
  - An array that stores a String version of the first 50 odd numbers (e.g., “1”, “3”, …).
  - An array to store 100 random true/falses.

Accessing Arrays: `for-each` Loops

- There is another type of loop that is very helpful for arrays.
- The “enhanced for-loop” or “for-each” loop allows you to:
  - **Access** each element of the array
- The for-each loop does ***not*** allow you to:
  - Access only *some* elements of the array
  - Access the *index* of the element
  - Assign new or updated values to an array

Accessing Arrays: `for-each` Loops

```
for(type varName : arrayName)  {
   statement;
}
```

- Read as "for each type in the arrayName, called, varName, perform a statement"
  - Example: for each int in scores (called score), double the score
  - Example: for each Course in the catalog (called c), print the title

Accessing Arrays: `for-each` Loops

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

equivalent!

```
for(int eachNum : nums)
    System.out.println(eachNum);
```

Accessing Arrays: `for-each` Loops

The for-each is the same as if you had a for-loop but declared a local variable inside the loop: `int eachNum = nums[i]`

```
for(int i=0; i<nums.length; i++) {
    int eachNum = nums[i];
    // do something with eachNum
}
```

equivalent!

```
for(int eachNum : nums) {
    // do something with eachNum
}
```

When to use which?

- Both are acceptable.
  - Although the for-each is gaining in popularity because many people find it easier to read and it eliminates the need for the (often meaningless) index variable.
  - If you don't care about the index value, use a for-each!
  - Oracle also now recommends its use: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/for.html
- Anything you can do with a for-each can also be done with a regular for.
- But there are things you *can* do with a for-loop that you *cannot* do with a for-each.

What can't a for-each do?

- Access only *some* elements of the array
  - Example: `for(int i=0; i<array.length; i=i+2)`
    - Allows you to process every-other element in an array
    - Can't be done directly with a for-each
- Access the *index* of the element
  - Example:
    ```
    for(int i=0; i<array.length; i++)
            System.out.println("The item at pos. " +
            (i+1) + "=" + array[i]");
    ```
  - Allows you to fill an array based on the index
  - Since there is no index variable in a for-each, you can't do this directly
- *Update* the value stored in an array
  - Example:
    ```
    for(int i=0; i<array.length; i++)
            array[i] = newValue;
    ```
  - Allows you to update the contents of an array at a certain position
  - Since there is no index, there is no way to update at that position.

Caution!

- You **cannot** update array contents with a for-each.

```
for(int eachNum : nums) {
      eachNum = newValue;
   }
```

mistake!!

```
for(int i=0; i<nums.length; i++) {
      int eachNum = nums[i];
      eachNum = newValue;
   }
```

that would be the equivalent of this- which does NOT change the array!

Practice

- Use a for-loop to print out the contents of your arrays.
- Use a for-each loop to do the same thing.

Arrays are Objects!

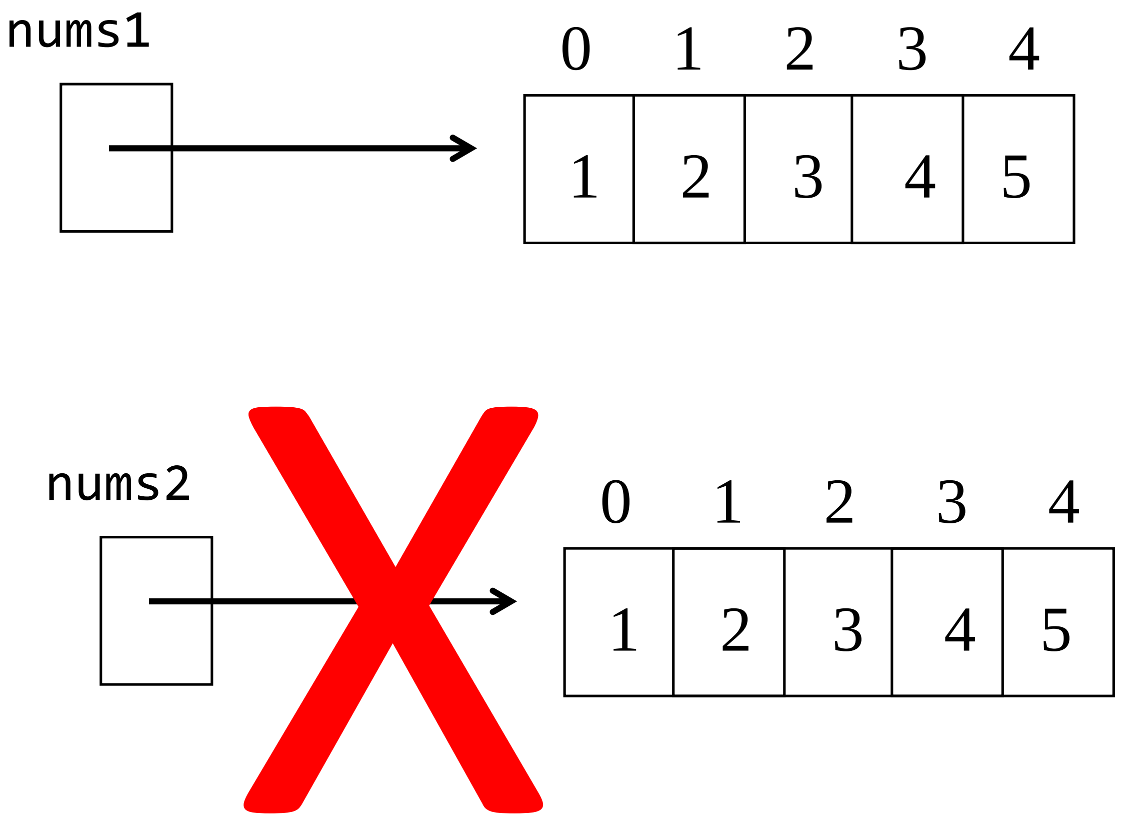
- Recall that arrays are objects.
- So what is stored in memory is a memory location or reference to somewhere else where the full data is stored.
- Recall what this means about using direct assignment!

Caution!

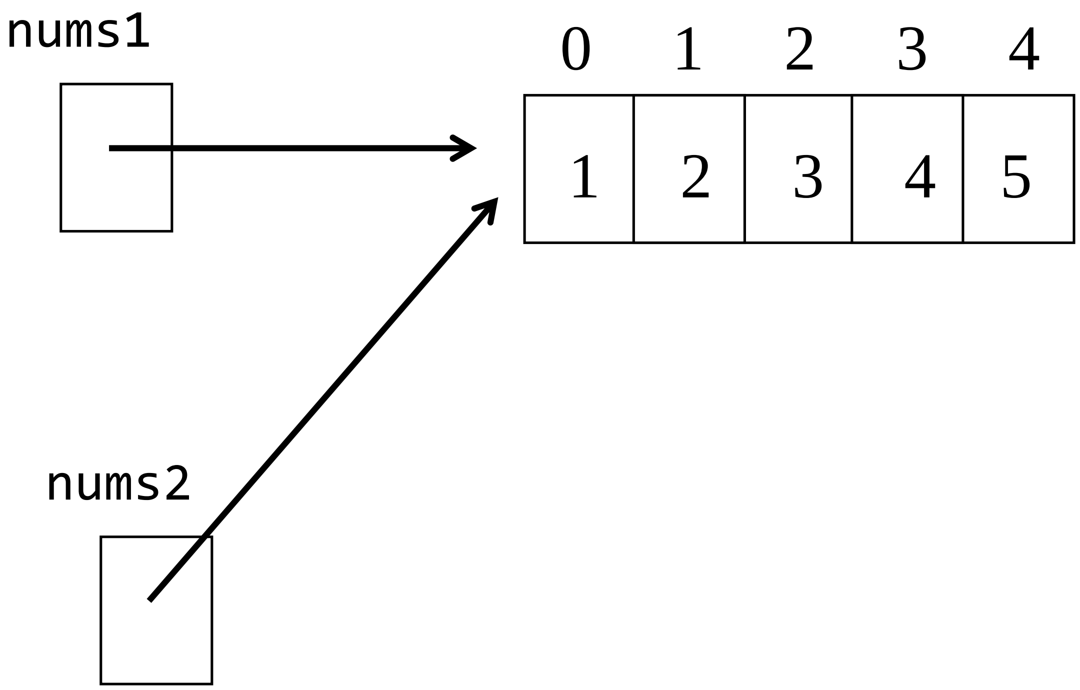
- What is stored in `nums1[0]` after this code?

```
int[] nums1 = {1, 2, 3, 4, 5};
int[] nums2 = nums1;
nums2[0] = 99;
```

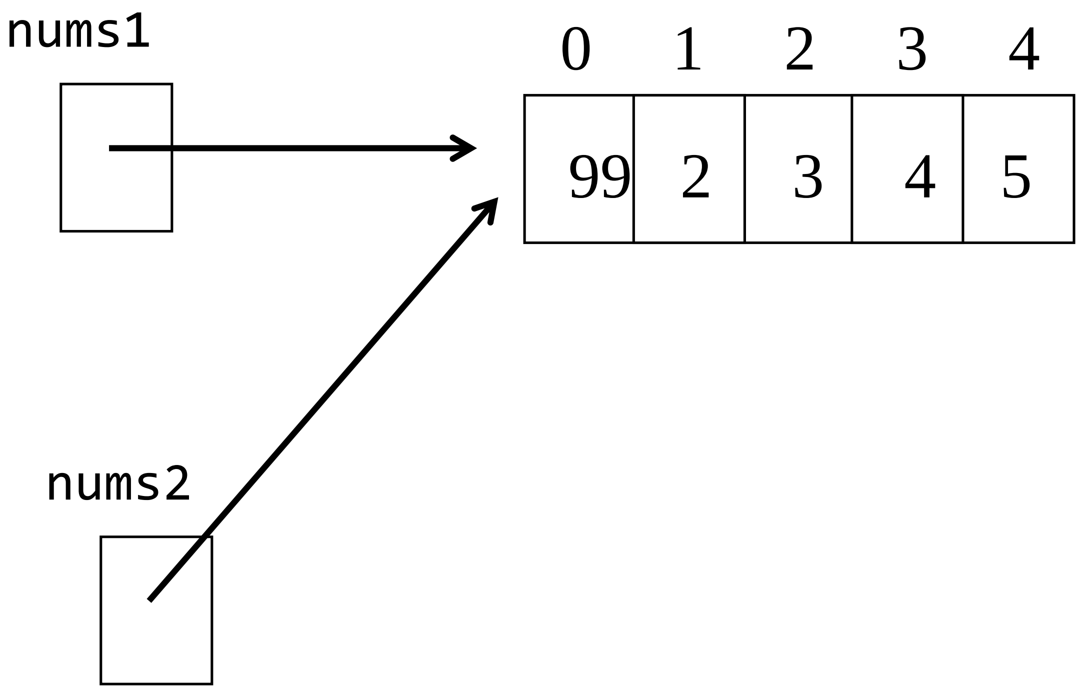
Arrays as Objects



Arrays as Objects



Arrays as Objects



Copying Arrays

- What is stored in `nums1[0]` after this code?

```
int[] nums1 = {1, 2, 3, 4, 5};
int[] nums2 = nums1;
nums2[0] = 99;
```

- 99 is stored there!
  - `nums1` and `nums2` aliases of each other.
  - Changes to one will affect changes to the other.
- To only affect one of these arrays, we need to make a *copy*:
  - Create a whole new array
  - Copy over *each* element in the array

Array Size Revisited

- You cannot make an array bigger.
- You *can* create a *new, bigger* array and make your array reference point to it.
  – You need to manually copy over the data.

Copying an array

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

original array

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

bigger, empty array

```
for(int i=0; i<nums.length; i++) {
  moreNums[i] = nums[i];
}
```

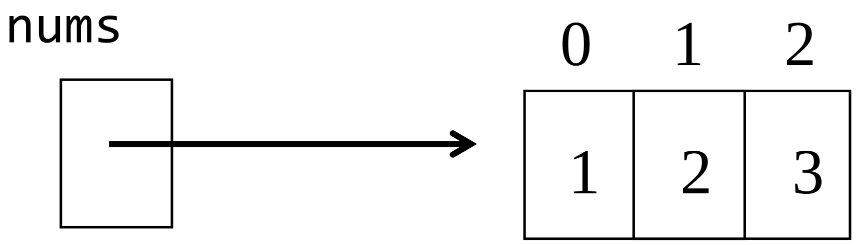
copy the old array contents into the new array

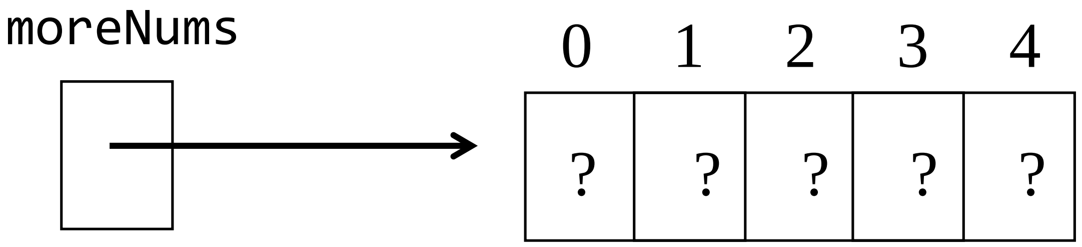
```
nums = moreNums;
```

point the reference to the new array

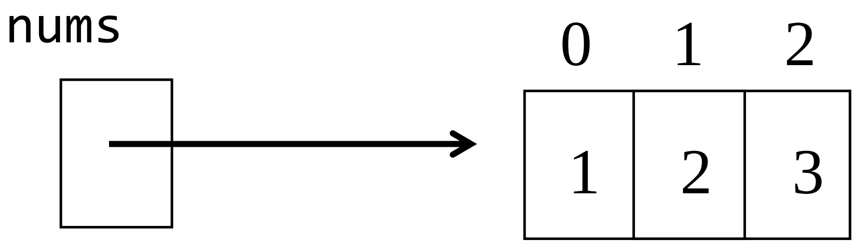
the old “nums” memory location is lost and those contents are garbage collected

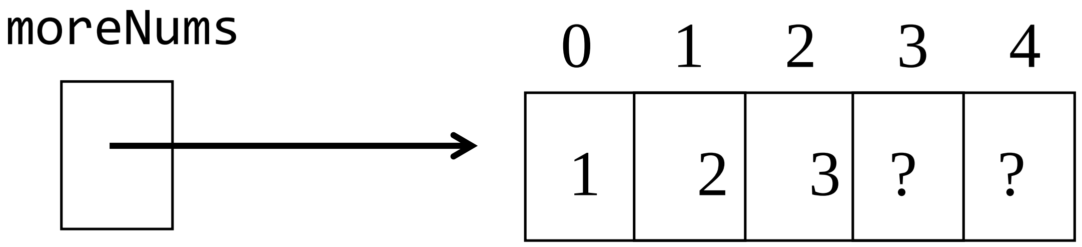
Arrays as Objects



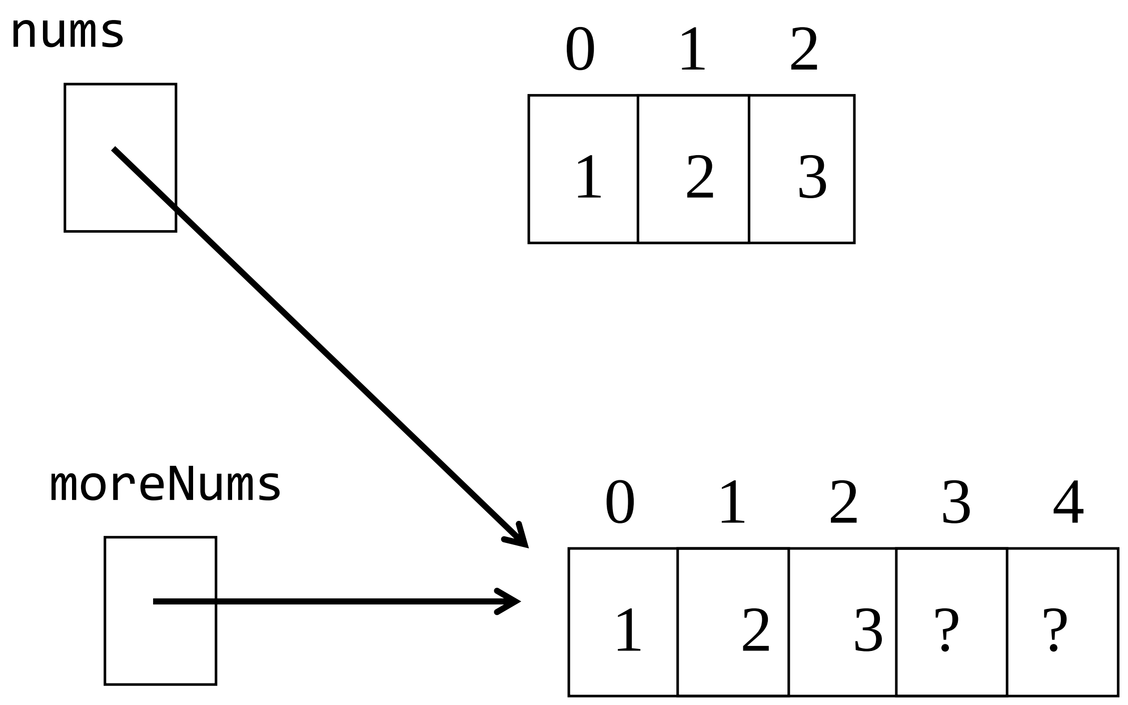


Arrays as Objects





Arrays as Objects



Practice

- Copy your array that stored the numbers from 1 to 100 into an array of size 200. Finish filling it up.

The Arrays Class

- Note that Java provides a class called Arrays (note the “s”) which has lots of useful array utilities, including a copy method that does the work for you!
- Google “java arrays api”

Arrays as Parameters

- An entire array can be passed as a parameter to a method.
- Like any other object, the value is passed.
  - The value is the *reference* to the data in the array.
  - So, the formal and actual parameters (the two array references) become *aliases* of each other.
- This means that changing an array element within the method will also change that element in the original copy of the array.

Practice

- Fill an array with 100 random integers between 0 and 499. Then write a method to:
  - find the maximum value in the array.
  - display the elements in the array in reverse order. (Do not change the array.)
  - print all elements in the array that are greater than a passed parameter.
  - return the sum of all elements greater than a passed parameter.
  - return the number of even elements in the array.

More Practice

- Write a program to read integers from the user in the range of 0-50 (inclusive). The user enters a number *outside* of the range to quit. Use an array to count how many occurrences there are of each number. When the user is done, print the frequency of each number entered by the user.
  - Hint: how can you use the array index to help you?

Even More Practice!

- Write code to determine if an array contains any duplicate values. The method header is:
  `public static boolean hasDuplicates(int[] numbers)`
  - Hint: think about a nested loop!
- Write a method to double the elements in an array.
  - First double the values in the actual array.
  - Second create a new array with doubled values.
  - This demonstrates the pass-by-value issue related to passing an array as a parameter!

THE ARRAYLIST CLASS

The `ArrayList` Class

- The `java.util` package contains the `ArrayList` class
- Like an array, an `ArrayList` can store a list of values and reference each one with a numeric index
- Unlike an array, an `ArrayList` object grows and shrinks as needed, adjusting its capacity as necessary

The `ArrayList` Class

- Elements can be inserted, removed, and retrieved by invoking methods.
- Some useful methods are below. See all methods at the API page (google “java arraylist api”)

```
add(Object obj)  // adds to the end of the list
add(int index, Object obj)   // adds at index
remove(int index) // removes the item at index
size() // returns the number of items in the list
get(int index) // returns the item at that position
contains(Object obj) // returns true/false
```

The `ArrayList` Class

- When an element is inserted in the middle, the other elements “move aside” or “shift down” to make room
- When an element is removed, the list “collapses” to close the gap
- All indexes of elements are automatically updated.
- Indexes start at 0!

`ArrayList` Efficiency

- The `ArrayList` class is implemented behind-the-scenes using an array.
- The array is manipulated so that indexes remain continuous as elements are added or removed.
- If elements are added to and removed from the end of the list, this processing is fairly efficient.
- If elements are inserted and removed from the front or middle of the list, all remaining elements need to be shifted, which can be inefficient.

`ArrayList` and Generics

- An `ArrayList` stores references to the `Object` class, which means it can store any object reference.
  - It can store anything except primitives.
- We can also define an `ArrayList` to accept *only* a particular type of object.
- Example:

  -
    ```
    ArrayList<String>  names=
            new ArrayList<String>()
    ```
  - This `ArrayList names` will now only accept `String` objects
- This is an example of *generics*!

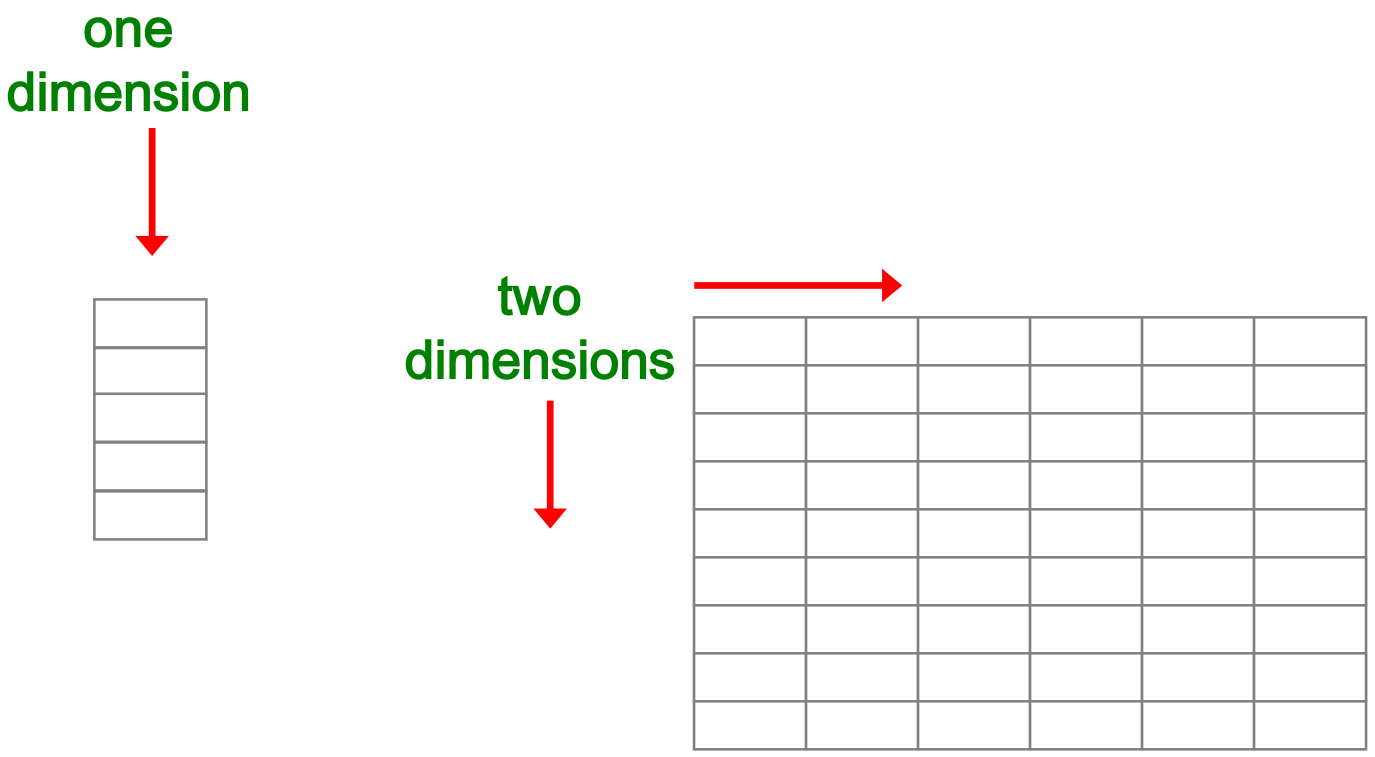
Practice

- Write a method to read numbers from the user. The user enters 0 to quit.
  - Keep track of the numbers in an ArrayList.
  - Print the number of numbers entered, the sum of the numbers, and how many numbers were even.
- Write a method to have a user add Strings to an ArrayList. If the word is not in the list, add it to the list. When the user enters “quit,” end the method and print the contents of the list.

TWO-DIMENSIONAL ARRAYS

Two-Dimensional Arrays

- A *one-dimensional array* stores a list of elements.
- A *two-dimensional array* can be thought of as a table of elements, with rows and columns.



Two-Dimensional Arrays (cont.)

- A two-dimensional array is declared by specifying the size of each dimension separately.
  - The first size is the number of rows.
  - The second size is the number of columns.
- An element is referenced with two index values (the row and column of the element)

Two-Dimensional Arrays (cont.)

- You often used a nested for-loop to access two-dimensional arrays.
  - The outer loop iterates the number of rows.
  - The inner loop iterates the number of columns.

Two-Dimensional Array Example

```
String[][] table = new String[3][5];
for(int i=0; i<3; i++)
  for(int j=0; j<5; j++)
     table[i][j] = "(" + i + "," + j +")";
```

(0,0)	**(0,1)**	**(0,2)**	**(0,3)**	**(0,4)**
(1,0)	**(1,1)**	**(1,2)**	**(1,3)**	**(1,4)**
(2,0)	**(2,1)**	**(2,2)**	**(2,3)**	**(2,4)**

Arrays of Arrays

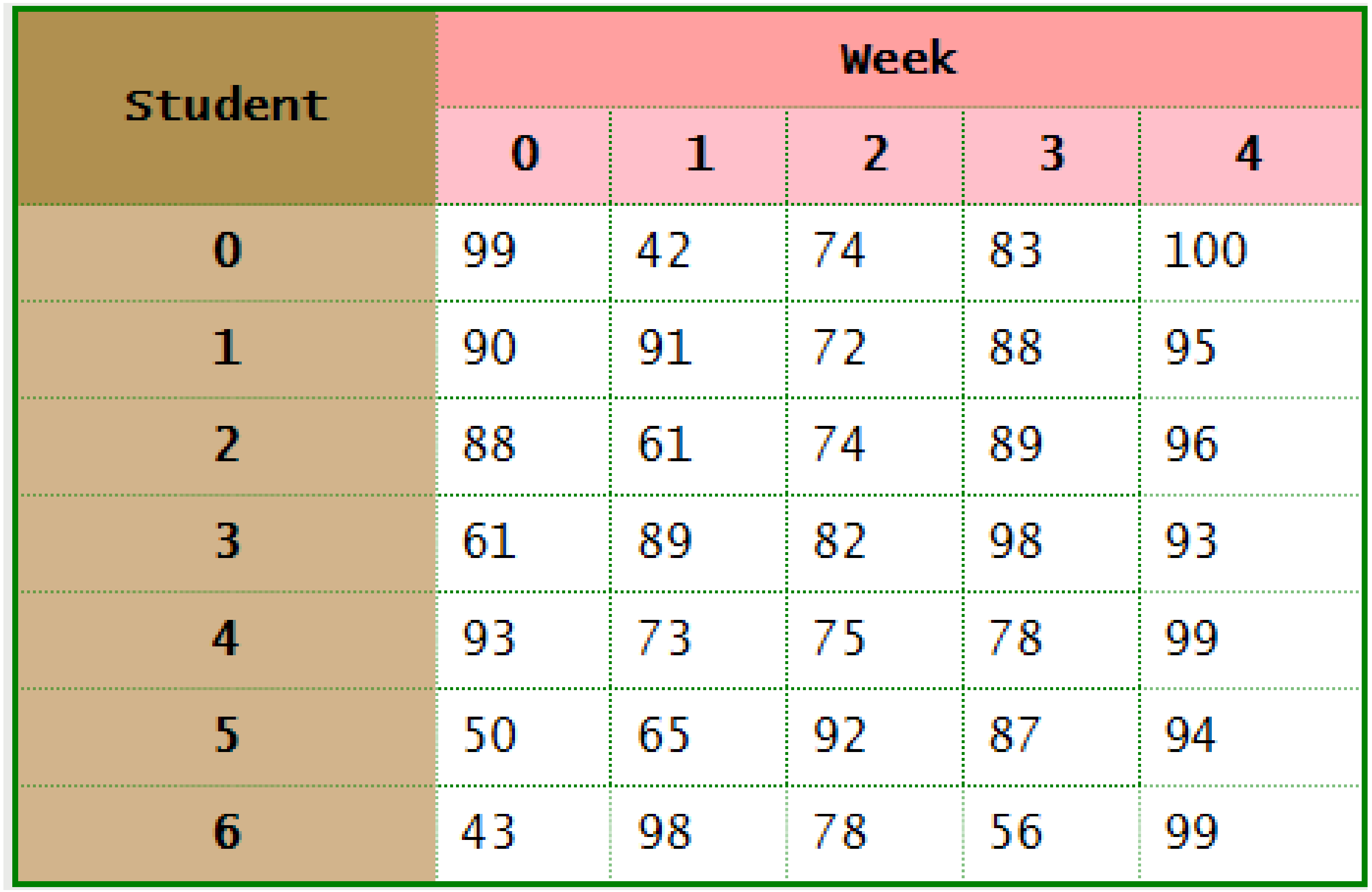
- A two-dimensional array is actually an array of arrays.
  - Another way to think about `table` is as an array with 3 elements.
  - Each element is an array with 5 elements.
- You can reference the array stored in one row by using one index.
  - `String[] secondTableRow = table[1];`

(1,0)	**(1,1)**	**(1,2)**	**(1,3)**	**(1,4)**

Arrays of Arrays (cont.)

- You can access the number of rows by using the `length` field:
  - `table.length; // this is 3`
- You can access the number of columns by using the `length` field on any row:
  - `table[0].length; // this is 5`
  - `table[1].length; // this is 5`
  - `table[2].length; // this is 5`

Example: GradeBook



Student	Week				
	0	1	2	3	4
0	99	42	74	83	100
1	90	91	72	88	95
2	88	61	74	89	96
3	61	89	82	98	93
4	93	73	75	78	99
5	50	65	92	87	94
6	43	98	78	56	99

Example: GradeBook

```
int[][] gradeBook = new int[7][5];

int[] student0 = { 99, 42, 74, 83, 100 };
gradeBook[0] = student0;

int[] student1 = { 90, 91, 72, 88, 95 };
gradeBook[1] = student1;

int[] student2 = { 88, 61, 74, 89, 96 };
gradeBook[2] = student2;

int[] student3 = { 61, 89, 82, 98, 93 };
gradeBook[3] = student3;

int[] student4 = { 93, 73, 75, 78, 99 };
gradeBook[4] = student4;

int[] student5 = { 50, 65, 92, 87, 94 };
gradeBook[5] = student5;
```

Example: GradeBook

```
for (int row = 0; row < gradeBook.length; row++) {
   for (int col = 0; col < gradeBook[row].length; col++)
       System.out.print(gradeBook[row][col] + "\t");
   System.out.println();
}
```

Multidimensional Arrays

- An array can have any number of dimensions
- Each dimension subdivides the previous one into the specified number of elements
- Each dimension has its own `length` constant
- Because each dimension is an array of array references, the arrays within one dimension can be of different lengths (often called *ragged arrays*).

- Using multidimensional arrays can make your program logic hard to follow.
- If you can’t picture it, perhaps stay away from it…

MORE ARRAY FUN

Command-Line Arguments

- The `main` method takes a `String[]` as a parameter
- This array represents *command-line arguments* that can be provided when a program is run
- (You don’t see this too much anymore.)

Command-Line Arguments (cont.)

- Example: `java NameReader jane jim john`
  - This command will run the `NameReader` program and send the main method a `String[]` containing three elements:
    - `args[0] = jane`
    - `args[1] = jim`
    - `args[2] = john`

Command-Line Arguments (cont.)

- If using a development environment, you might have to use a special setting to access command-line arguments.
  - In Eclipse, it’s under Run > Run Configurations > Arguments

Variable Length Parameter Lists

- It’s often helpful to create a method that processes different amounts of data from one invocation to the next.
- Example: an `average` method that returns the average of a set of integer parameters

```
// one call to average three values
mean1 = average (42, 69, 37);

// another call to average seven values
mean2 = average (35, 43, 93, 23, 40, 21, 75);
```

Variable Length Parameter Lists (cont.)

- One approach would be to overload the `average` method.
  - One method would take three parameters and the other would take seven.
  - But what if we want to average four numbers? Or 100?
  - We don’t want to have to overload that many methods!

Variable Length Parameter Lists (cont.)

- Another approach would be to accept an array of integers as the parameter.
  - But what if we don’t have the numbers in an array already when we call the method?
  - We would have to declare and initialize the array before invoking the method.
  - That could be a pain…

Variable Length Parameter Lists (cont.)

- A third approach is to use a *variable length parameter list* which allows for any number of parameters with the same type.
- In the method header, we use a special syntax to indicate there will be any number of parameters.
- Inside the method, the parameter is treated as an *array.*

Variable Length Parameter Lists (cont.)

Indicates a variable length parameter list

```
public double average (int ... numList)
{
   // whatever
}
```

element type

array name

- Using this syntax in the formal parameter, the `average` method can accept any number of integers as parameters.
- Behind the scenes, the parameters are automatically put into an array
  - The parameter `numList` is an `int[ ]`

Variable Length Parameter Lists (cont.)

```
public double average (int ... list) {
   double result = 0.0;

   if (list.length != 0) {
      int sum = 0;
      for (int num : list)
         sum += num;
      result = (double) sum / list.length;
   }

   return result;
}
```

Important Check!

Variable Length Parameter Lists (cont.)

- The type of the parameter can be primitive or object

```
public void printGrades (Grade ... grades) {
   for (Grade letterGrade : grades)
      System.out.println (letterGrade);
}
```

grades is a Grade[]

Variable Length Parameter Lists (cont.)

- A method with a variable number of parameters can also accept other parameters.
  - The varying number of parameters must come *last* in the formal arguments.
  - A single method cannot accept two sets of varying parameters.

```
public void test (int count, String name,
                  double ... nums) {
   // whatever
}
```

- Constructors can use variable parameters.

Practice

- Write a method to concatenate Strings together. Use a variable length parameter list.

SUMMING UP

Key Points

- Arrays are objects!
  - Java is pass by value. The value of an object is the memory location. This applies to arrays.
- Arrays are indexed from position 0 to length-1 (inclusive)
- ArrayLists are great!