Chapter 7

Arrays

Adapted from notes by Pat Baker based on Java Foundations by Lewis, Chase, & DePasquale

Chapter Topics

- Array declaration and use
- Bounds checking
- Arrays as objects
- Arrays of objects
- Command-line arguments
- Variable-length parameter lists
- Multidimensional arrays

Arrays

An **array** is an object that holds a list of values

- The array has a name that represents the *entire* array
- Each value in the array is stored at a specific location (cell)
- Each cell has a numeric index
  - Note that array indices start at 0
  - So an array with 5 cells has indices 0, 1, 2, 3, & 4

Arrays

An **array** is an object that holds a list of values.

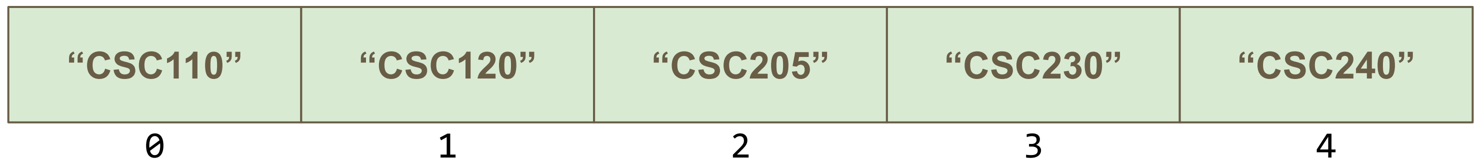
Array of ints:

3	98	45	68	129	21	9	42	57	35	77
0	1	2	3	4	5	6	7	8	9	10

Arrays

An **array** is an object that holds a list of values.

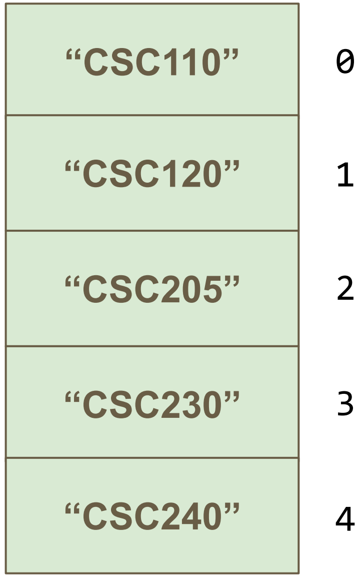
Array of Strings:



Arrays

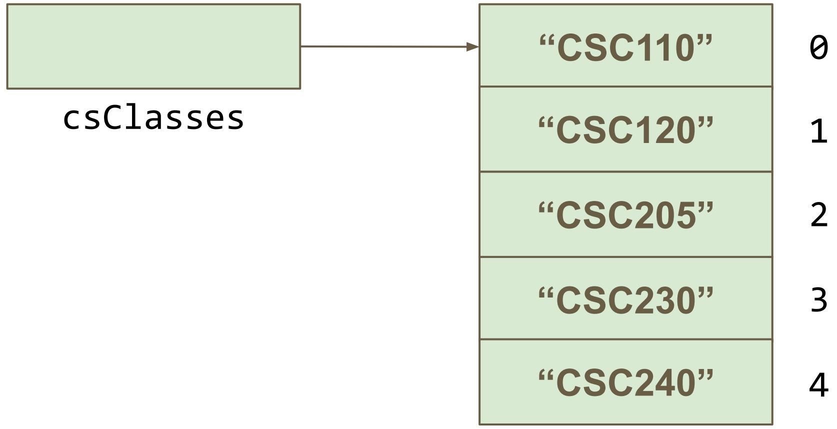
An **array** is an object that holds a list of values.

Array of Strings (this time displayed with a horizontal orientation):



Arrays

- An array is an object and an array can hold objects as elements
- The array name is an *object reference variable*
- Say that our String array was called “csClasses”
- Then this is another way to visually depict an array:



Arrays

3	98	45	68	129	21	9	42	57	35	77
0	1	2	3	4	5	6	7	8	9	10

We reference a particular value stored in an array using the array's name followed by the index of the value in brackets.
If the array above was named "**`nums`**", then

- **`nums[0]`** = 3
- **`nums[1]`** = 98
- **`nums[3]`** = 3
- **`nums[7]`** = 3
- **`nums[10]`** = 3

Arrays

- We can also use expressions as the index of the array so

  ```
  nums[x+4]
  ```

  would return the index 4 cells beyond cell x

- We can use an array expression in place of any variable (as long as the array stores the correct type

  ```
  int result = nums[7] + 3
  ```

  would return the value **`45`** (42 + 3)

Arrays

- An array element can be assigned a value, printed, or used in a calculation

```
scores[2] = 89;

scores[first] = scores[first] + 2;

mean = (scores[0] + scores[1])/2;

System.out.println("Top = " + scores[5]);
```

Arrays

- The values held in an array are called **array elements**
- An array stores multiple values of the same type – the **element type**
- The element type can be a primitive type or an object reference
- Therefore, we can create an array of ints, an array of chars, an array of `String` objects, an array of `Account` objects, etc.
- In Java, the array itself is an object that must be instantiated

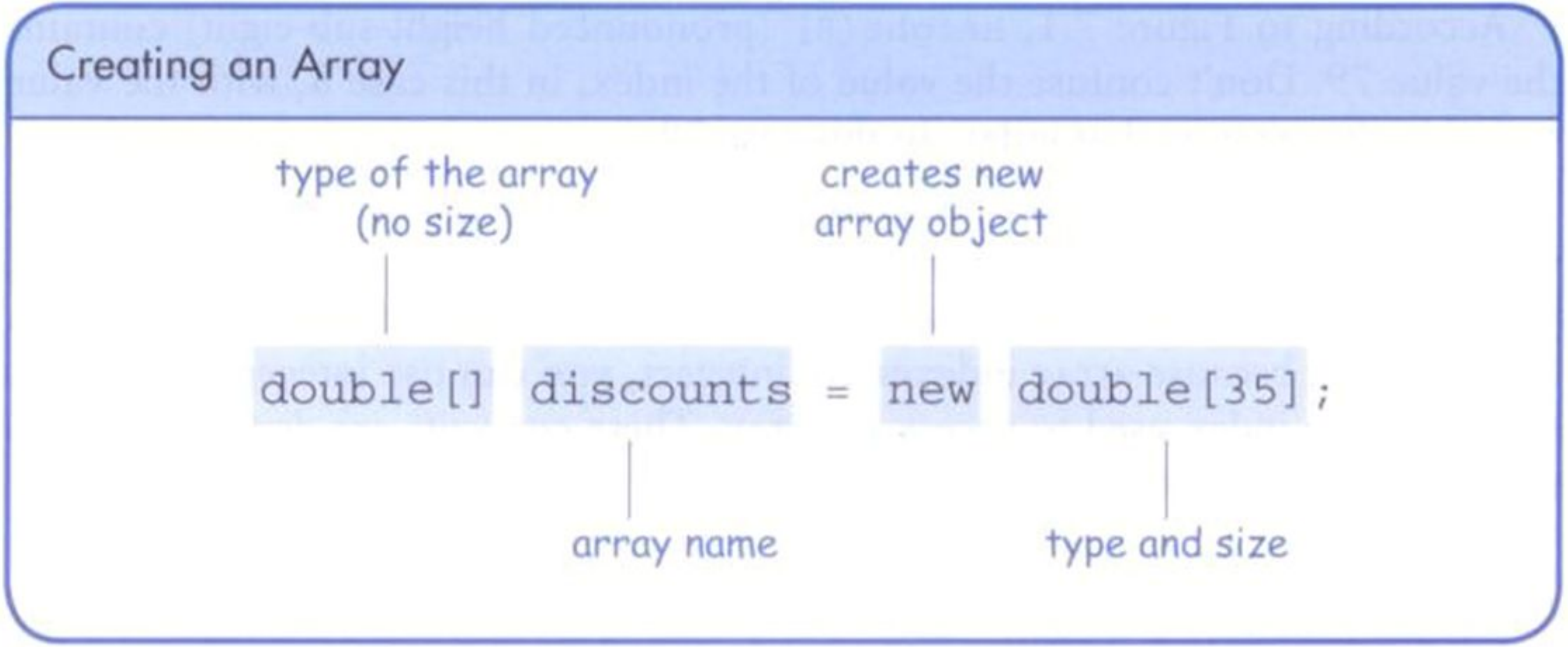
Declaring Arrays

- The `nums` array could be declared as follows

```
int[] nums = new int[11];
```

- The type of the variable `nums` is `int[]` (an array of integers)
- Note that the array type does not specify its size, but each object of that type has a specific size
- The reference variable `nums` is set to a new array object that can hold 11 integers
-

Declaring Arrays


Creating an Array

```
double[] discounts = new double[35];
```

Declaring Arrays

Some other examples of array declarations

```
float[] prices = new float[500];

boolean[] flags;

flags = new boolean[20];

char[] codes = new char[1750];
```

-

Using Arrays

- The for-each loop can be used when processing array elements:

```
for (int score : scores)
    System.out.println(score);
```

- Note that using a for-each loop is only appropriate when you want to process every one of the array elements from the lowest index to the highest index

Bounds Checking

- Once an array is created, it has a fixed size
- An index used in an array reference must specify a *valid* element
- That is, the index value must be in range 0 to N-1
- The Java interpreter throws an `ArrayIndexOutOfBoundsException` if an array index is out of bounds
- This is called **automatic bounds checking**

Bounds Checking

- For example, if the array codes can hold 100 values, it can be indexed using only the numbers 0 to 99
- If the value of count is 100, then the following reference will cause an exception to be thrown

```
System.out.println(codes[count]);
```

- It's common to introduce off-by-one errors when using arrays

```
for (int index=0; index <= 100; index++)
    codes[index] = index*50 + epsilon;
```

Bounds Checking

- Each array object has a public constant called length that stores the size of the array
- It is referenced using the array name

```
nums.length
```

- Note that length holds the number of elements, not the largest index

Alternate Array Syntax

- The brackets of the array type can be associated with *the element* type or with the name of the array
- Therefore the following two declarations are equivalent

```
float[] prices;

float prices[];
```

- The first format generally is more readable and should be used

Array Initialization

- An **initializer list** can be used to instantiate and fill an array in one step
- The values are delimited by braces and separated by commas
- Examples:

```
int[] units = {147, 323, 89, 933, 540,
               269, 97, 114, 298, 476};

char[] letterGrades = {'A', 'B', 'C', 'D', ’F'};
```

Array Initialization

- Note that when an initializer list is used
  - the `new` operator is not used
  - no size value is specified
- The size of the array is determined by the number of items in the initializer list
- An initializer list can be used only in the array declaration

Arrays as Parameters

- An entire array can be passed as a parameter to a method
- Like any other object, the reference to the array is passed, making the formal and actual parameters aliases of each other
- Therefore, changing an array element within the method changes the original
- An individual array element can be passed to a method as well, in which case the type of the formal parameter is the same as the element type

Arrays of Objects

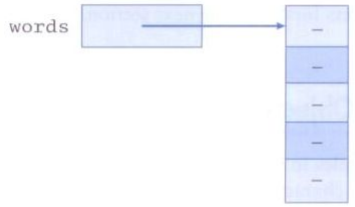
- An array of objects really holds object references
- The following declaration reserves space to store 5 references to `String` objects

```
    String[] words = new String[5];
```

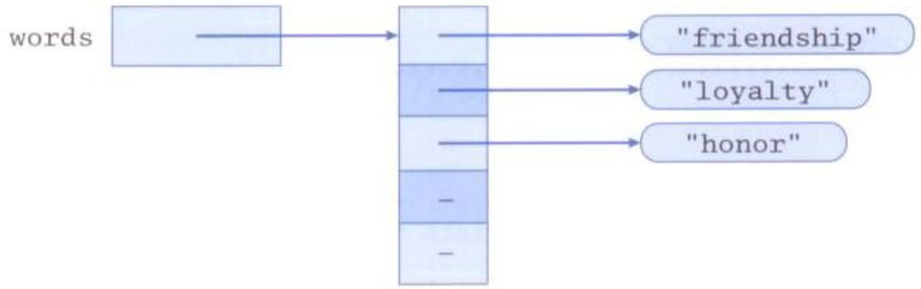
- It does *not* create the String objects themselves
- Initially an array of objects holds null references
- Each object stored in an array must be instantiated separately

Arrays of Objects

- After initial creation, an array holds null references:



- Each element is a reference to an object:



Arrays of Objects

- Keep in mind that String objects can be created using literals
- The following declaration creates an array object called verbs and fills it with four String objects created using string literals

```
String[] verbs = {"play", "work", "eat", "sleep"};
```

Arrays of Objects

The following example creates an array of Grade objects, each with a string representation and a numeric lower bound

```
Grade[] grades =
{
   new Grade("A", 95), new Grade("A-", 90),
   new Grade("B+", 87), new Grade("B", 85), new Grade("B-", 80),
   new Grade("C+", 77), new Grade("C", 75), new Grade("C-", 70),
   new Grade("D+", 67), new Grade("D", 65), new Grade("D-", 60),
   new Grade("F", 0)
};
```

The for-each Loop and Arrays

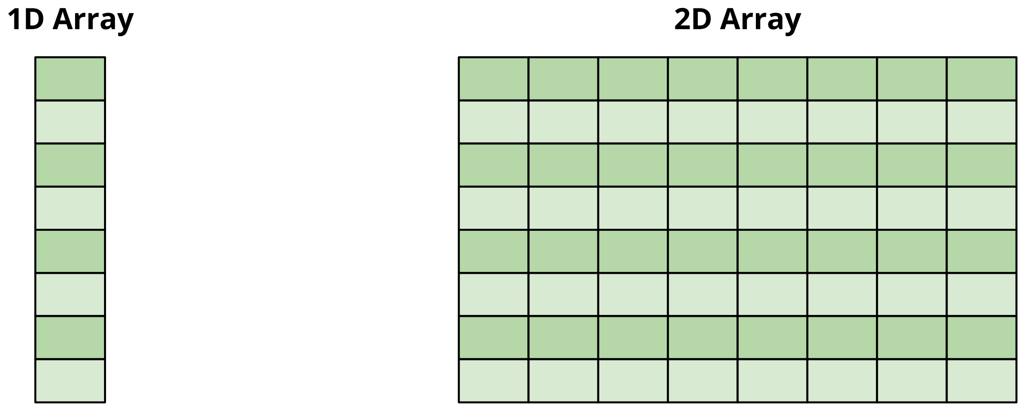
- We can use a variant of the for loop called the **for-each** loop to s
- For example, if **`GradeList`** is an array that manages **`int`** values, the following loop will print each number in the array:

```
    for (int num : GradeList) {
        System.out.print(num);
```

- Notice we don’t have to use array indexes when using a for-each loop - that is all handled under the hood by Java
- You also can only iterate *forward* using a for-each loop

Two-Dimensional Arrays

- The arrays we have seen before store lists of elements
- You can also have a 2 dimensional array, which you can think of as a table with row and columns



Two-Dimensional Arrays

- To be precise, in Java a two-dimensional array is an array of arrays
- A two-dimensional array is declared by specifying the size of each dimension separately

```
int[][] scores = new int[12][50];
```

Two-Dimensional Arrays

We can also initialize a 2D array when we declare it:

```
int[][] scores = { {89,  73, 83, 94,  95},
                   {98, 100, 94, 92, 100},
                   {88,  94, 88, 79,  81},
                   {100, 89, 91, 98,  94} };
```

Result:

89	73	83	94	95
98	100	94	92	100
88	94	88	79	81
100	89	91	98	94

Two-Dimensional Arrays

We reference individual elements using two index values

```
        value = scores[2][4];
```

Some examples of using indices to access elements:

```
scores[0][0] = 89
scores[1][3] = 92
scores[2][1] = 94
scores[2][2] = 88
scores[2][4] = 81
scores[3][4] = 94
```

Scores

89	73	83	94	95
98	100	94	92	100
88	94	88	79	81
100	89	91	98	94

Two-Dimensional Arrays

We can reference individual rows using a single index:

```
        value = scores[2];
```

Some examples of using indices to access individual rows:

Scores[0]	89	73	83	94	95
Scores[1]	98	100	94	92	100
Scores[2]	88	94	88	79	81
Scores[3]	100	89	91	98	94

Two-Dimensional Arrays

Suppose we have a 2D array declared as follows:

```
int[][] table = new int[5][10];
```

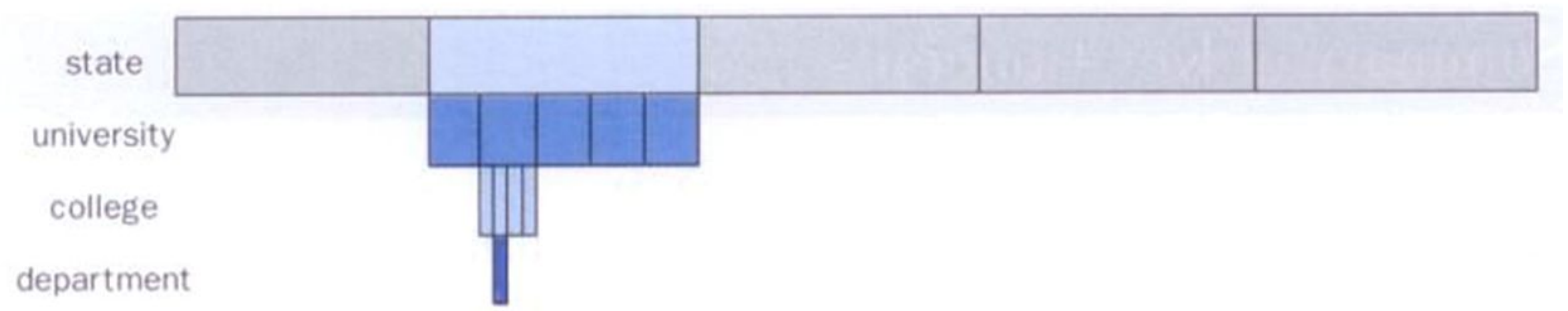
Expression	Type	Description
`table`	`int[][]`	2D array of integer, or array of integer arrays
`table[5]`	`int[]`	Array of integers
`table[5][2]`	`int`	Integer

Multi-dimensional Arrays

- Any array with more than one dimension is a **multidimensional array**
- 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
- these are sometimes called ragged arrays

Arrays

- One way to visualize a four-dimensional array:



- Two-dimensional arrays are common, but beyond that usually an array has other objects involved

Command-Line Arguments

- The signature of the `main` method indicates that it takes an array of `String` objects as a parameter
- These values come from **command-line arguments** that are provided when the interpreter is invoked
- For example, the following invocation of the interpreter passes three `String` objects into main

```
> java StateEval pennsylvania texas arizona
```

- These strings are stored at indexes 0-2 of the array parameter of the main method

Variable Length Parameter Lists

- Suppose we wanted to create a method that processed a different amount of data from one invocation to the next
- For example, let's define a method called average that returns the average of a set of integer parameters

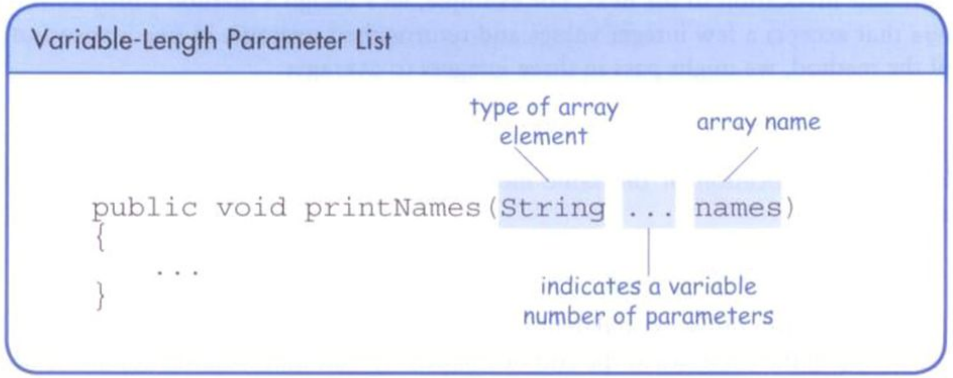
```
// one call to average three values
mean1 = average (42, 69, 37);

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

Variable Length Parameter Lists

- We could define overloaded versions of the `average` method
- Downside: we'd need a separate version of the method for each parameter count
- We could define the method to accept an array of integers
    - Downside: we'd have to create the array and store the integers prior to calling the method each time
- Instead, Java provides a convenient way to create **variable length parameter lists**

Variable Length Parameter Lists

- Using special syntax in the formal parameter list, we can define a method to accept any number of parameters of the same type
- For each call, the parameters are automatically put into an array for easy processing in the method


Variable-Length Parameter List

```
public void printNames(String ... names)
{
    . . .
}
```

Variable Length Parameter Lists

```
public double average(int ... list)
{
    double result = 0.0;
    if (list.length != 0)
    {
    int sum = 0;
    for (int num : list)
        sum += num;
    result = (double)num / list.length;
    }
    return result;
}
```

Variable Length Parameter Lists

The type of the parameter can be any primitive or object type

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

Variable Length Parameter Lists

- A method that accepts a variable number of parameters can also accept other parameters
- The following method accepts an int, a String object, and a variable number of double values into an array called nums

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

Variable Length Parameter Lists

- The varying number of parameters must come last in the formal arguments
- A single method cannot accept two sets of varying parameters
- Constructors can also be set up to accept a variable number of parameters
-

Now go write some code!