Chapter 7: Arrays

CS 121

Department of Computer Science
College of Engineering
Boise State University

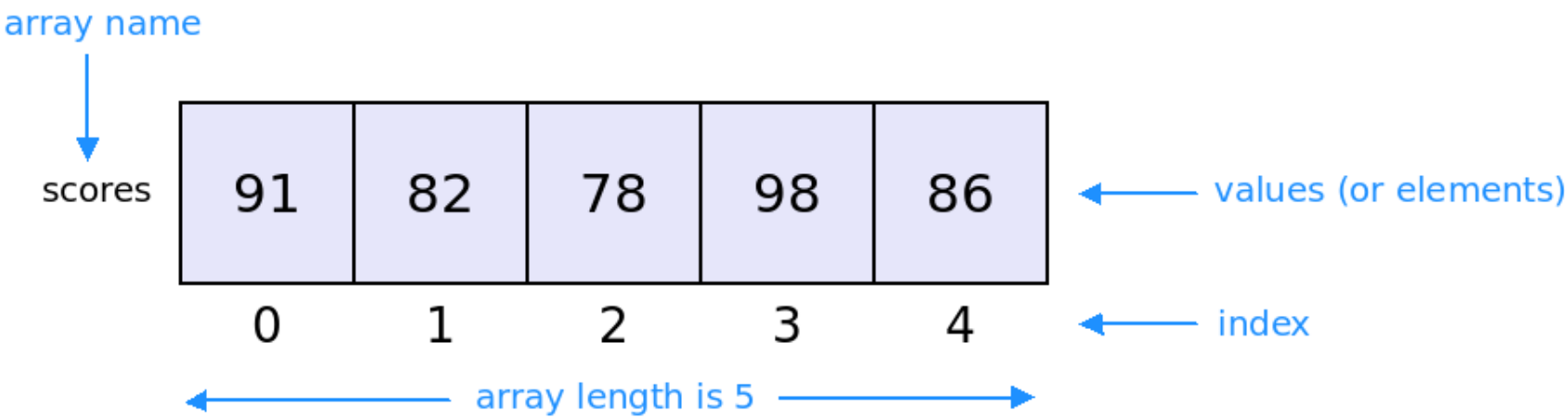
August 29, 2016

Topics

- Array declaration and use Go to part 0
- Bounds checking Go to part 1
- Arrays as objects Go to part 2
- Arrays of objects Go to part 3
- Arrays as Method Parameters Go to part 4
- Command-line arguments Go to part 5
- Multi-dimensional arrays Go to part 6

Arrays

- An array is an ordered list of values.

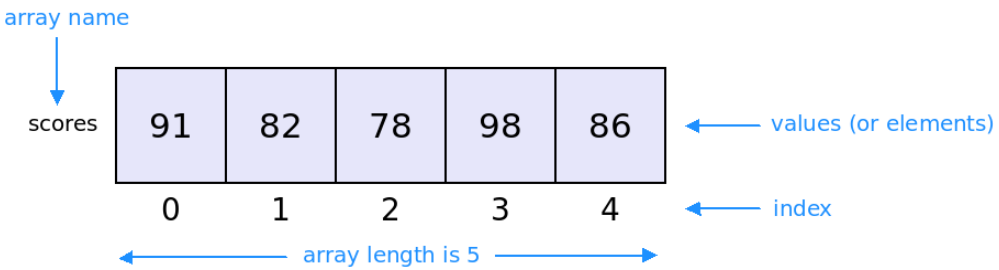


- Each array has a name by which it can be referenced.
- Each value (or element), of an array has a numeric index.

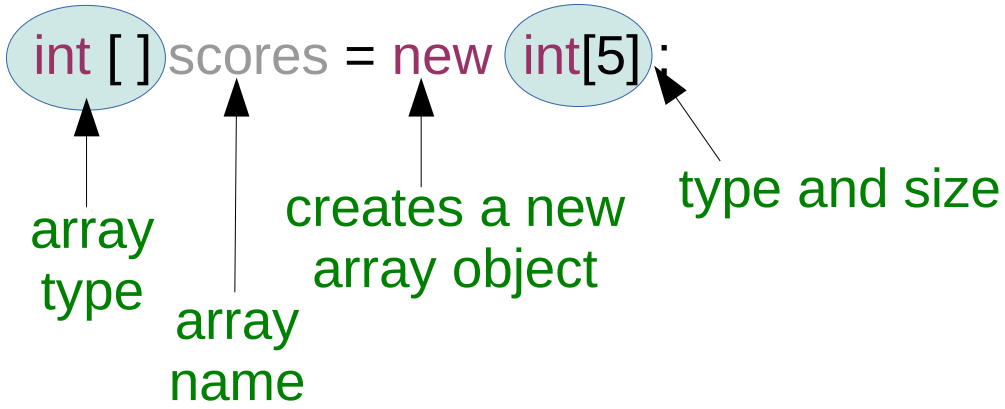
Arrays

- In Java, arrays are indexed from 0 to $n-1$, where n is the number of elements in the array.
  - For example, our `scores` array has 5 elements that are indexed from $0-4$.
- Values stored in the same array must be of the same type – the element type.
- The element type can be a primitive type (e.g. `int`, `double`, `boolean` etc.) or an object reference (e.g. `String`, `Song`, `Card`, etc.)
- In Java, the array itself is an object that must be instantiated using the `new` operator.

Declaring Arrays



- The `scores` array could be declared as follows.



- LHS – Declares the type of the `scores` variable as `int[]` (meaning, an array of `int` values).
- RHS – Instantiates a new `int[]` (integer array) object of size 5.

Declaring Arrays

- An array of letters

```
char[] letters;
letters = new char[26];
```

- An array of `String` objects

```
String[] dictionary = new String[480000];
```

- An array of `Song` objects

```
Song[] playlist = new Song[3];
```

- An array of `Card` objects

```
Card[] deckOfCards = new Card[52];
```

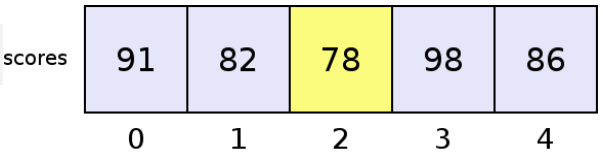
- An array of `boolean` objects

```
boolean[] lightSwitches = new boolean[100];
```

Accessing Array Elements

- A particular value in an array can be referenced using its index in the array.
- For example, to access the second element of our `scores` array, we would use the expression

```
scores[2]
```



- The value returned by the expression `scores[i]` is just an `int`. So, we can have expressions like,

```
totalScore += scores[2];
scores[2] = 89;  // Updates the value in the array
scores[count] = scores[count] + 2;
System.out.println("High score: " + scores[3]);
```

Using Arrays

- Typically, array elements are accessed using a for loop:

```
// every array has a public constant called length
// that stores the size of the array
int totalScore = 0;
for (int i = 0; i < scores.length; i++)
{
    totalScore += scores[i];
}
```

- Or a for-each loop:

```
int totalScore = 0;
for (int score: scores)
{
    totalScore += score;
}
```

Using Arrays: Example

```
/**
 *  BasicArray.java - Demonstrates basic array declaration and use.
 *  @author Java Foundations
 */
public class BasicArray
{
   /**
    * Creates an array, fills it with various integer values,
    * modifies one value, then prints them out.
    */
   public static void main(String[] args)
   {
      final int LIMIT = 15, MULTIPLE = 10;

      int[] list = new int[LIMIT];

      //  Initialize the array values
      for (int index = 0; index < LIMIT; index++)
         list[index] = index * MULTIPLE;

      list[5] = 999;  // change one array value

      //  Print the array values
      for (int value: list)
         System.out.print(value + "  ");
   }
}
```

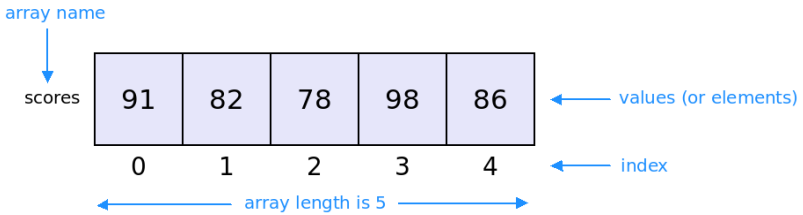
Using Arrays: Example

Bounds Checking

- When an array is created, it has a fixed size. The size of the array is provided by a public constant named `length`.
- When accessing an element of an array, we must use a valid index. For example, for an array `scores`, the range of valid indexes is `0` to `scores.length - 1`.
- What happens when we try to access something out of bounds? The Java interpreter throws an `ArrayIndexOutOfBoundsException`.
- This is called automatic bounds checking.

Bounds Checking

- Recall our `scores` array. The valid index range is 0 to 4.



- Now, we want to print all values in our array using this loop:

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

- Will this work? *NO. The last iteration of our loop is trying to access the element at index* 5. *But it doesn't exist!*
- *We will get an exception...*

```
java ScoresArray
10 20 30 40 50 Exception in thread "main" java.
    lang.ArrayIndexOutOfBoundsException: 5
        at ScoresArray.main(ScoresArray.java:10)
```

Bounds Checking

- Off-by-one errors are common when using arrays.
- Remember, the `length` constant stores the *size* of the array, not the largest index.
- The correct loop condition is

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

Examples

- Example: ReverseOrder.java
  - Reads a list of numbers from a user and prints it in the opposite order.
- Example: LetterCount.java
  - Reads a sentence and prints the counts of lowercase and uppercase letters.

In-class Exercise

- Write an array declaration for the ages of 100 children.
- Write a for loop to print the ages of the children
- Write a for-each loop to print the ages of the children
- Write a for loop to find the average age of these children, assuming that the array has been initialized.

In-class Exercise

- What does the following code do?

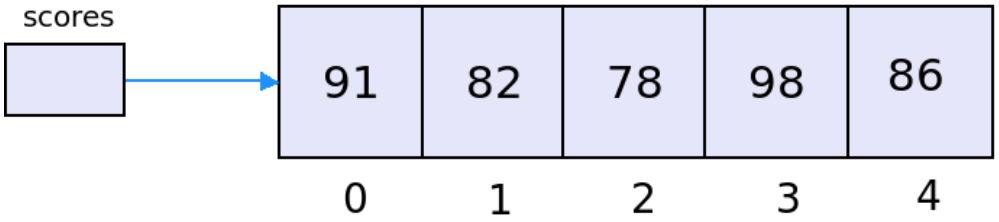
```
int[] array = new int[100];
for (int i = 0; i < array.length; i++)
    array[i] = 1;

int[] temp = new int[200];
for (int i = 0; i < array.length; i++)
    temp[i] = array[i];
```

- What happens if we now assign `temp` to `array`?

```
array = temp;
```

Arrays of Objects (1)

- The name of an array is an object reference variable:



- An array of objects really just holds object *references*. For example, the following declaration reserves space to store 5 references to `String` objects.

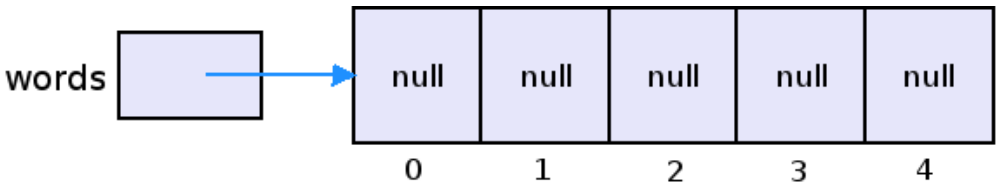
```
String[] words = new String[5];
```

- It does **not** create the `String` objects themselves.
- Initially, the array holds `null` references. We need to create the `String` objects.

Arrays of Objects (2)

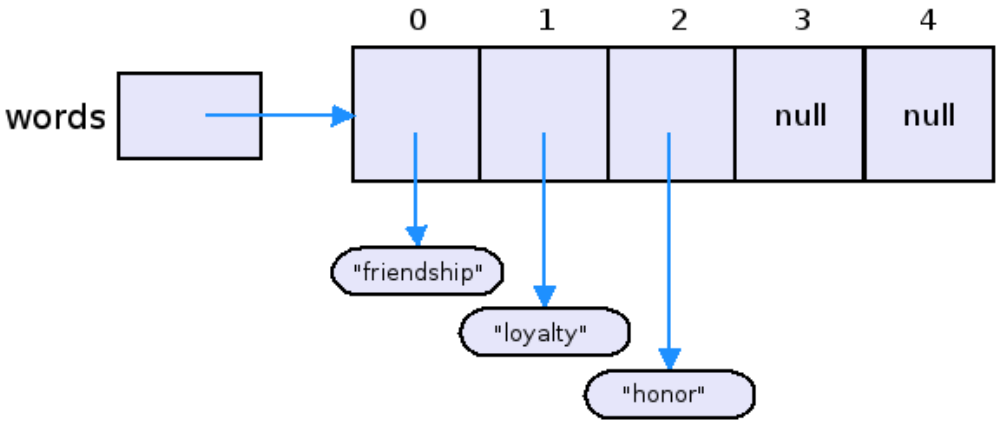
- After declaration.

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



- After adding 3 strings.

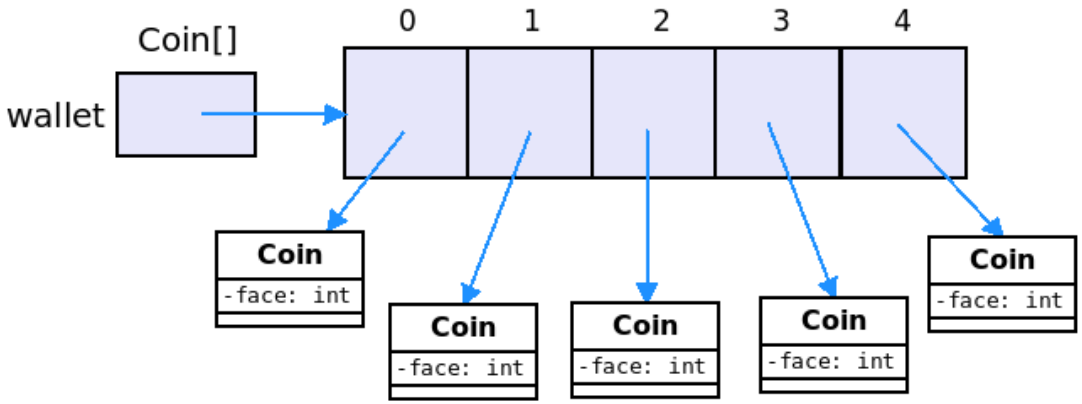
```
words[0] = "friendship";
words[1] = "loyalty";
words[2] = "honor";
```



Arrays of Objects (3)

- An array of coins.

```
Coin[] wallet = new Coin[5];
for (int i = 0; i< wallet.length; i++)
    wallet[i] = new Coin();
```

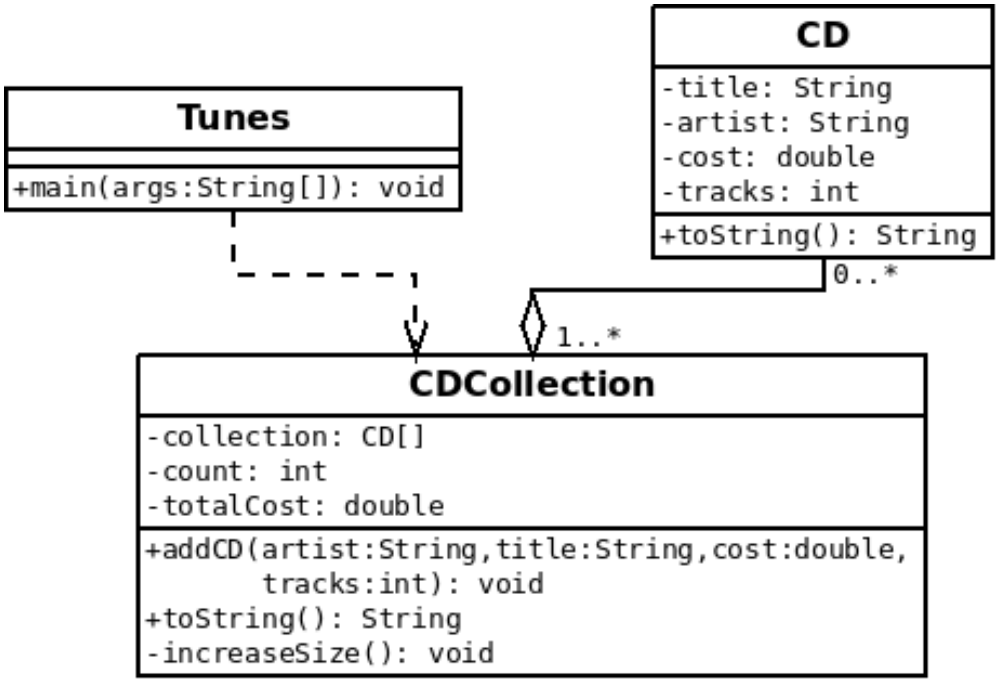


- A collection of a hundred random die.

```
Random rand = new Random();
Die[] diceCollection = new Die[100];
for (int i = 0; i< diceCollection.length; i++) {
    int numFaces = rand.nextInt(20) + 1;
    diceCollection[i] = new Die(numFaces);
}
```

Arrays of Objects (4)

- Example: CD.java, CDCollection.java, Tunes.java



Growing Arrays: A Space–Time Tradeoff

- The size of an array is fixed at the time of creation. What if the array fills up and we want to add more elements?
- We can create a new array and copy the existing elements to the new array. In effect, we have *grown the array*.
- How much bigger should the new array be?
  - **Minimum space**: We could grow the array by one element so it can store the new element.
  - **Minimum time**: Grow the array to the maximum size we will ever need. However, in many cases we don't know ahead of time how large the array needs to grow....
  - Heuristic: A good heuristic is to double the size so we don't have to do the copying again and again.
- The `ArrayList` class grows an array internally.
- Example: GrowingArrays.java

In-class Exercise

- Declare and instantiate an array of hundred `Color` objects.

```
Color[] myColors = new Color[100];
```

- Now fill the array with random colors using a for loop.

```
Random rand = new Random();
for (int i = 0; i < myColors.length; i++) {
    myColors[i] = new Color(rand.nextInt(256),
                            rand.nextInt(256),
                            rand.nextInt(256));
}
```

- Write an array declaration and any necessary supporting classes to represent credit card transactions that contain a transaction number, a merchant name, and a charge.

Initializing Arrays

- An initializer list can be used to instantiate and fill an array in one step.
- For example,

```
int[] scores = {91, 82, 78, 98, 86};
String[] fruit = {"apple", "orange", "banana"};
```

- The `new` operator is not needed (it is implied).
- The *size* of the new array is determined by the number of items in the initializer list.
- Initializer lists can only be used in the array declaration.
- Initializer lists can contain expressions or objects (including calls to `new` to create objects). For example:

```
Die[] myCollection = {new Die(10), new Die(20),
                      new Die(20)};
```

Arrays as Method 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 outside of the method.
- 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.
- Example: ArrayPassing.java

Command-Line Arguments (1)

- A program can accept any number of arguments from the command line (known as command-line arguments).
- Allows the user to specify configuration information *when the program is launched*, instead of asking for it at run-time.
- For example, suppose a Java application called `Sort` sorts lines in a file. To sort the data in a file named `friends.txt`, a user would enter:

```
java Sort friends.txt
```

Command-Line Arguments (2)

- Recall: The `main` method takes an array of `String` objects as a parameter.

```
public static void main(String[] args) { ... }
```

- When an application is launched, the runtime system passes the command-line arguments to the application's `main` method via this array of `String` objects.
- In our previous example, the `String` array passed to the `main` method of the `Sort` application contains a single `String`: `"friends.txt"`.

Iterating Over Command-Line Arguments (1)

- The following program (CommandLineEcho.java) prints each element of the `args` array to the console.

```
public class CommandLineEcho
{
    public static void main(String[] args)
    {
        for (String arg: args)
            System.out.println(arg);
    }
}
```

- If we execute the program as follows

```
java CommandLineEcho monkey peanut banana
```

- We would get

```
monkey
peanut
banana
```

Iterating Over Command-Line Arguments (2)

- Note that the space character separates command-line arguments.
- To have all words interpreted as a single argument, we can enclose them in quotation marks.

```
java CommandLineEcho "monkey peanut banana"
```

- Would give us

```
monkey peanut banana
```

Parsing Command-Line Arguments

- We always want to validate our command-line arguments and print an appropriate usage message to the user if they entered invalid arguements.
- Typically, we want to validate
  - the number of arguments
  - the type of arguments
  - the values are within a specific range
- Let's say we have a program that accepts a filename (`String`) followed by the number of characters per line (`int`). The number of characters per line must be between 1 and 80.
- Example: CommandLineValidation.java

Parsing Numeric Command-Line Arguments

- In many cases, our command-line arguments will need to support numeric arguments.
- To handle this, we need to convert a `String` argument to a numeric value.

```
int firstArg;
if (args.length > 0) {
    try {
        firstArg = Integer.parseInt(args[0]);
    } catch (NumberFormatException e) {
        System.err.println("Argument" + args[0]
                           + " must be an integer.");
        System.exit(1);
    }
}
```

- `parseInt` throws a `NumberFormatException` if the format of `args[0]` isn't valid.
- All of the wrapper classes for primitive types have `parseX` methods that convert a `String` representing a number to an object of their type `X`.

2-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.

1-dimensional

91	82	78	98	86
0	1	2	3	4

2-dimensional

	0	1	2	3	4
0	91	82	78	98	92
1	80	80	83	98	90
2	83	98	86	100	86

2-Dimensional Arrays (1)

- In Java, a 2-D array is an array of arrays.
- A 2-D array is declared by specifying the size of each dimension separately.

```
int[][] table = new int[3][5];
```

- An array element is referenced using two index values

```
int value = table[1][3];
```

	0	1	2	3	4
0	91	82	78	98	92
1	80	80	83	98	90
2	83	98	86	100	86

- Note that `table.length` is the number of rows in the table.
- Note that `table[i].length` is the length of the *i*th row in the table

2-Dimensional Arrays (2)

- **In-class Exercise**. What does the following 2-d array contain after the code executes?

```
int numRows = 3, numCols = 5;
int[][] table = new int[numRows][numCols];

for (int row = 0; row < numRows; row++)
    for (int col = 0; col < numCols ; col++)
        table[row][col] = row;
```

- **In-class Exercise**. What if we change the initialization?

```
for (int row = 0; row < numRows; row++)
    for (int col = 0; col < numCols ; col++)
        table[row][col] = row * numCols + col;
```

2-Dimensional Arrays (3)

- **In-class Exercise**. What does the following method do?

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

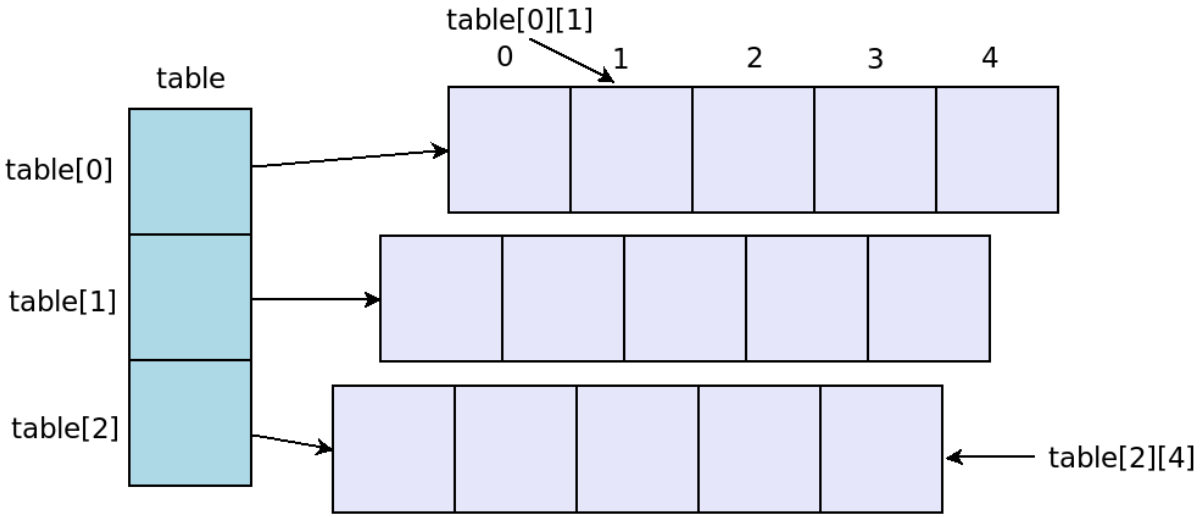
- Example: TwoDimArrays.java

2-Dimensional Arrays (3)

- Since a 2-dimensional array is an *array of arrays*, we can declare it in two parts:

```
int[][] table = new table[3][]; //2nd dim blank
for (int i = 0; i < table.length; i++)
    table[i] = new int[5];
```

- Layout of a 2-dim array in memory:



2-Dimensional Arrays (4)

- Two-dimensional arrays don't have to be square or rectangular in shape!
- Example: FunkyArrays.java
- **In-class Exercise** What does the following code do?

```
Color[][] board = new Color[8][8];
for (int row = 0; row < board.length; row++)
{
    for (int col = 0; col < board[row].length; col++)
    {
        if (row % 2 == col % 2)
            board[row][col] = Color.white;
        else
            board[row][col] = Color.red;
     }
}
```

Multi-Dimensional Arrays (1)

- Any array with more than one dimension is a multi-dimensional 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.

Multi-Dimensional Arrays (2)

- Arrays can have more than two-dimensions. Here is a declaration for a 3-dimensional array.

```
double[][][] data = new double[4][1000][100];
```

- Can you think of when a 3-D array might be useful?
  - A spreadsheet is a 2-dimensional array. The tabs would make it 3-dimensional.
  - Simulations of liquids, solids, space etc.
  - Modeling in science and engineering.
- A 4-D array? (not very common...)
- Instead of building larger dimensional arrays, it is a better design to have arrays of objects such that the objects contain arrays inside them as needed to get the dimensional depth.

Multi-Dimensional Arrays (3)

- Consider a 3-dim array to represent a universe that has a 100 galaxies. Suppose that each galaxy has a 1000 star clusters. Each cluster has 10 stars.

```
Star[][][] myUniverse = new Star[100][1000][10];

public class Star {
...
}
```

- Here is a different design that avoids the multidimensional array.

```
Galaxy[] myUniverse = new Galaxy[100];

public class Galaxy {
    private Cluster[] myClusters = new Cluster[1000];
    // other related instance variables
}

public class Cluster {
    private Star[] myStars = new Star[10];
    // other related instance variables
}

public class Star {
...
}
```

In-class Exercise

- How would we implement an `ArrayList<String>`? How would we implement the following operations?
  - `add(String element)`: adds an element to the end of the array list
  - `add(String element, int index)`: adds an element at the *index*th position
  - `remove(int index)`: removes an element at the *index*th position
  - `contains(String s)`: returns true if the array list contains the string `s`

Exercises

- Read Chapter 7 (skip Section 7.5).
- **Recommended Homework**:
  - Exercises: EX 7.1, 7.4 (e), 7.5, 7.8.
  - Projects: PP 7.1, 7.2, 7.5.
- Browse: Sections 6.1.