2d arrays
recursion

CIS 36a, Lecture 11.2

April 6, 2015

Nathaniel Titterton

Questions?

Multidimensional Arrays

- It is sometimes useful to have an array with more than one index
- Multidimensional arrays are declared and created in basically the same way as one-dimensional arrays
  - You simply use as many square brackets as there are indices
  - Each index must be enclosed in its own brackets

```
double[][]table = new double[100][10];
int[][][] figure = new int[10][20][30];
Person[][] = new Person[10][100];
```

Multidimensional Arrays

- Multidimensional arrays may have any number of indices, but perhaps the most common number is two
  - Two-dimensional array can be visualized as a two-dimensional display with the first index giving the row, and the second index giving the column

```
char[][] a = new char[5][12];
```

  - Note that, like a one-dimensional array, each element of a multidimensional array is just a variable of the base type (in this case, **char**)

Multidimensional Arrays

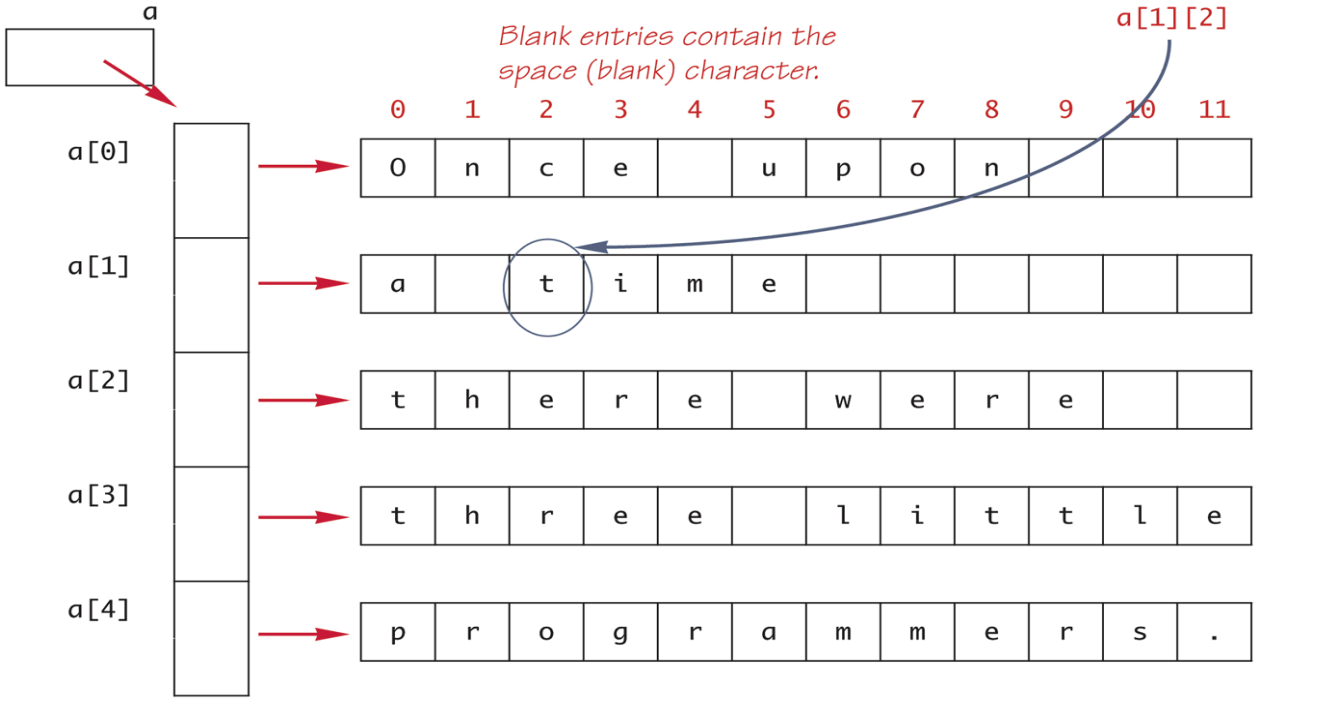
- In Java, a two-dimensional array, such as **a**, is actually an array of arrays
  - The array **a** contains a reference to a one-dimensional array of size 5 with a base type of **`char[]`**
  - Each indexed variable (**`a[0]`**, **`a[1]`**, etc.) contains a reference to a one-dimensional array of size 12, also with a base type of **`char[]`**
- A three-dimensional array is an array of arrays of arrays, and so forth for higher dimensions

Two-Dimensional Array as an Array of Arrays

Display 6.17 Two-Dimensional Array as an Array of Arrays

```
char[][] a = new char[5][12];
```

Code that fills the array is not shown.



(continued)

Examples

Using the `length` Instance Variable

```
char[][] page = new char[30][100];
```

- The instance variable **`length`** does not give the total number of indexed variables in a two-dimensional array
  - Because a two-dimensional array is actually an array of arrays, the instance variable **`length`** gives the number of first indices (or "rows") in the array
    - **`page.length`** is equal to 30
  - For the same reason, the number of second indices (or "columns") for a given "row" is given by referencing **`length`** for that *"row" variable*
    - **`page[0].length`** is equal to 100

Ragged Arrays

- Each row in a two-dimensional array need not have the same number of elements
  - Different rows can have different numbers of columns
- An array that has a different number of elements per row it is called a *ragged array*

Ragged Arrays

```
double[][] a = new double[3][5];
```

- The above line is equivalent to the following:

```
double [][] a;
a = new double[3][]; //Note below
a[0] = new double[5];
a[1] = new double[5];
a[2] = new double[5];
```

  - Note that the second line makes **a** the name of an array with room for 3 entries, each of which can be an array of **doubles** *that can be of any length*
  - The next 3 lines each create an array of doubles of size 5

Ragged Arrays

```
double [][] a;
a = new double[3][];
```

- Since the above line does not specify the size of `a[0]`, `a[1]`, or `a[2]`, each could be made a different size instead:

```
a[0] = new double[5];
a[1] = new double[10];
a[2] = new double[4];
```

Multidimensional Array Parameters and Returned Values

- Methods may have multidimensional array parameters
  - They are specified in a way similar to one-dimensional arrays
  - They use the same number of sets of square brackets as they have dimensions

```
public void myMethod(int[][] a)
{ . . . }
```

  - The parameter **a** is a two-dimensional array

Multidimensional Array Parameters and Returned Values

- Methods may have a multidimensional array type as their return type
  - They use the same kind of type specification as for a multidimensional array parameter

```
public double[][] aMethod()
{ . . . }
```

  - The method **`aMethod`** returns an array of **`double`**

Image processing

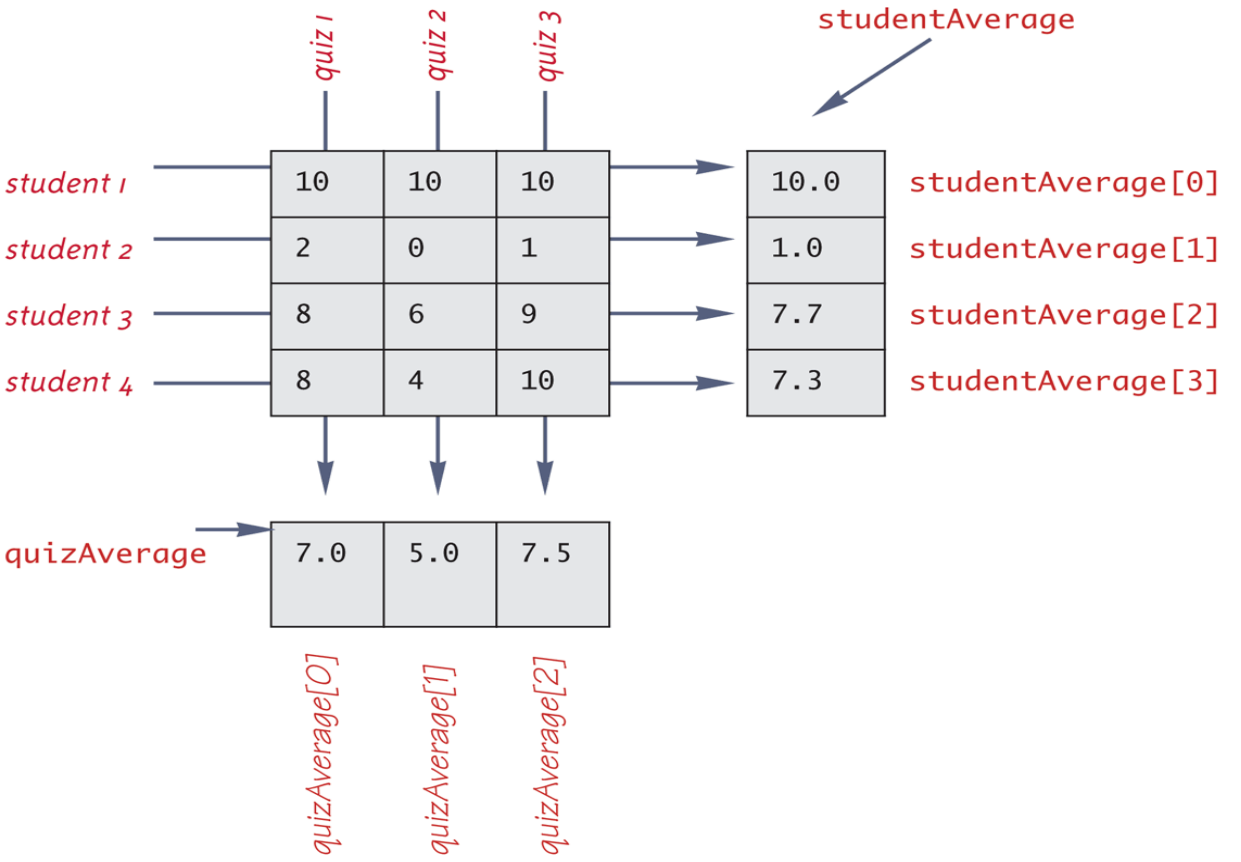
- Digital images can be considered arrays of pixels
- In our example, a image is stored as 3 separate arrays of the same size
  - one each for red, green, and blue

A Grade Book Class

- As an example of using arrays in a program, a class **`GradeBook`** is used to process quiz scores
- Objects of this class have three instance variables
  - **`grade`**: a two-dimensional array that records the grade of each student on each quiz
  - **`studentAverage`**: an array used to record the average quiz score for each student
  - **`quizAverage`**: an array used to record the average score for each quiz

The Two-Dimensional Array grade

Display 6.19 The Two-Dimensional Array grade



Recursion

Recursion

An algorithmic technique where a function, in order to accomplish a task, calls itself with some part of the task.

Using recursive procedures

- Everyone thinks it's hard!
  - (well, it is… aha!-hard, not complicated-hard)

- Using repetition and loops to find answers

All recursion procedures need...

1. Base Case (s)
    - Where the problem is simple enough to be solved directly
2. Recursive Cases (s)
    1. Divide the Problem
        - into one or more smaller problems
    2. Invoke the function
        - Have it call itself recursively on each smaller part
    3. Combine the solutions
        - Combine each subpart into a solution for the whole

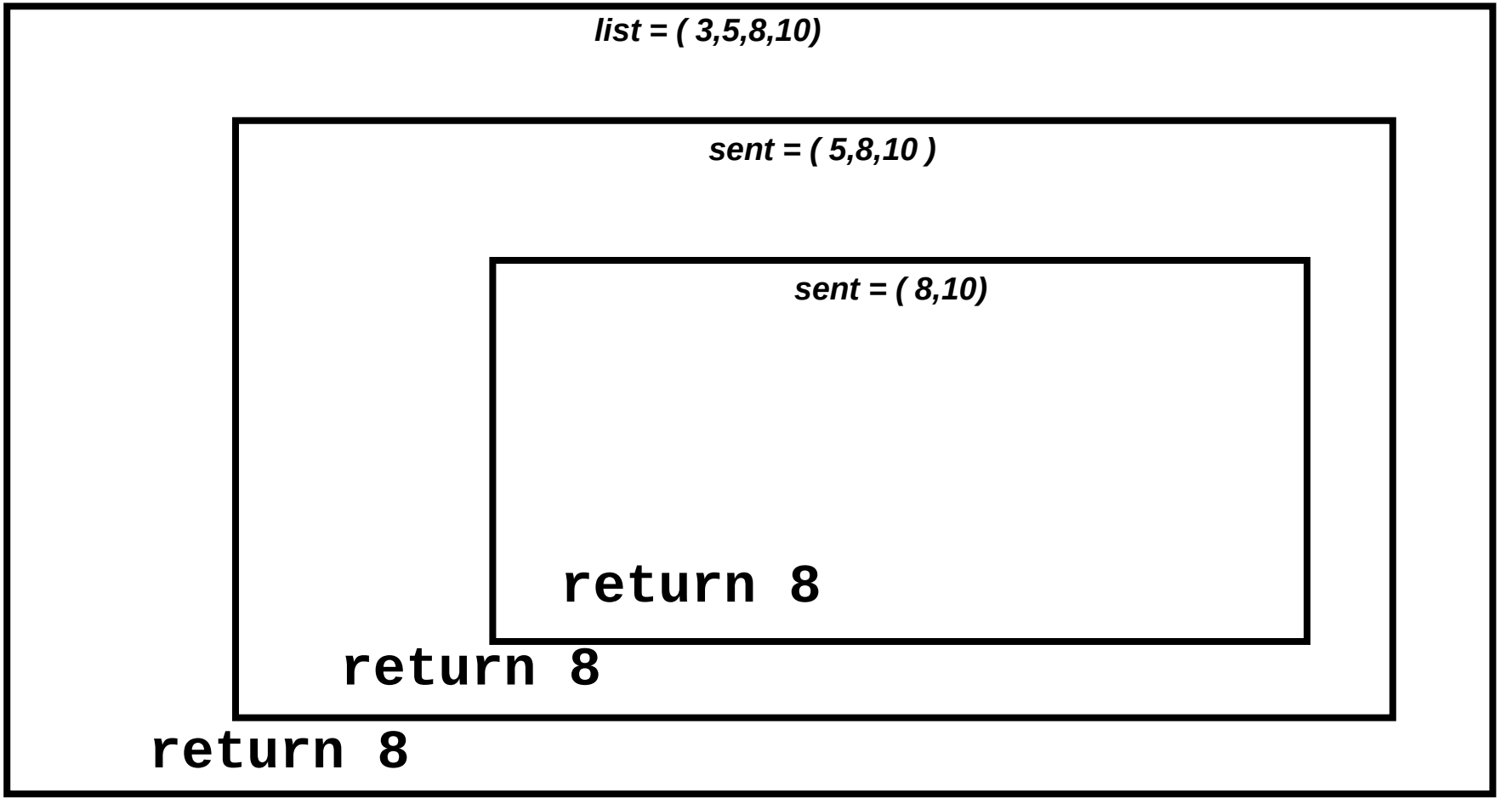
An example

```
// Find the number of characters in a nonnull
// (though possibly empty) string.

public static int charCount (String s) {
   if (s.equals("")) {
      return 0;
   } else {
      return 1 + charCount (s.substring(1));
   }
}
```

- A recursive call inside a method essentially creates a "clone" of the method.
  - The clone has copies of the arguments from its original caller.
  - When the clone is executed, it may set up another clone with a copy of its arguments.
    This is recursion!

findFirstEven {3, 5, 8, 10}



➔ 8

Understanding recursion 1

1. Examine the base cases. These should correspond to arguments for which the method can easily handle or return an answer for without any extra computation.

Understanding recursion 2

- Try arguments that are one recursive call away from a base case. Does the recursive call progress toward the base case? Does the method correctly use the result of the recursive call to compute its own answer?

Understanding recursion 3

- Try arguments that are two recursive calls away from a base case. Ask yourself the same questions: Is there progress toward a base case? Is the return value used correctly?

Understanding recursion 4

- By now, you should be seeing a pattern to the calls and returns. If there's a comment for the method that describes its arguments and return values, you ought by now to be able to rely on it without having to trace through multiple calls. If there's not a comment, you ought now to be able to invent one.