Lecture 12 - Ref. Type Examples and Multi-Dim Arrays

Bentley James Oakes

February 18, 2018

This Lecture

1 Primitive versus Reference Types

2 Null

3 Swapping Values

4 Reference Type Examples

5 Revisiting Strings

6 Two-Dimensional Arrays

7 2D Array Examples

Section 1

Primitive versus Reference Types

Primitive vs Reference Types

- Primitive types:
  - `int, double, boolean, char`
- Reference types:
  - `String`
  - Arrays
  - Objects

What's different about reference types?

- Can't use == for comparisons
- Variables store **addresses** instead of values
- We can call methods and access members of variables
  - `.equals()` for Strings, `.length` for arrays

Reference Type Variable

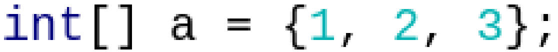
```
int[] a = {1, 2, 3};

System.out.println("Array a: " + a);
```

What prints?

```
Array a: [I@7347c5f3
```

This is the **address** of the data within a

Addresses Figure



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

Address	Variable Type	ID	Value
1171...	`int[]`	`a`	@7347...
...			
7347...	`int`	`a[0]`	1
7347...	`int`	`a[1]`	2
7347...	`int`	`a[2]`	3

- The array variable stores the address where the data starts
- When we index, we are looking up the address in the computer's memory

Consequences

- **Reference type variables store addresses**
- This means that printing out their value might not work
- Also means that we have to compare them a different way
    - This is why we can't use == to compare Strings
- Another consequence: we can have two variables storing the same address

Aliasing

```
//create a
int[] a = {1, 2, 3};
System.out.println(a);
System.out.println(Arrays.toString(a));

//copy the address from a into b
int[] b = a;
System.out.println(b);
System.out.println(Arrays.toString(b));

//change the first element in b
b[0] = 5;

//print out a again
System.out.println(Arrays.toString(a));
```

What prints?

```
[I@72510787
[1, 2, 3]
[I@72510787
[1, 2, 3]
```

- Here we have two array variables pointing to the **same address**
- A change in one affects the other
- This is called **aliasing**
- Analogy: They both contain the same website address, so changes are seen for both

Aliasing

```
//create a
int[] a = {1, 2, 3};
System.out.println(a);
System.out.println(Arrays.toString(a));

//copy the address from a into b
int[] b = a;
System.out.println(b);
System.out.println(Arrays.toString(b));

//change the first element in b
b[0] = 5;

//print out a again
System.out.println(Arrays.toString(a));
```

What prints?

```
[I@72510787
[1, 2, 3]
[I@72510787
[1, 2, 3]
[5, 2, 3]
```

- Here we have two array variables pointing to the **same address**
- A change in one affects the other
- This is called **aliasing**
- Analogy: They both contain the same website address, so changes are seen for both

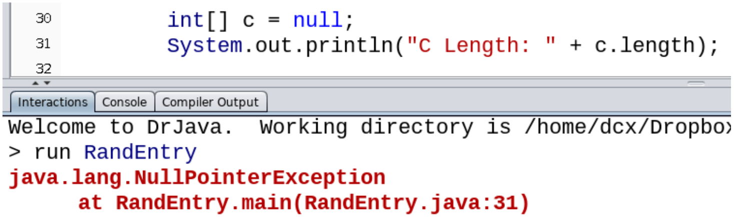
Section 2

Null

null

- Reference type variables can also store the **null** value.
- `null` means *no address*
  - Analogy: The website address is a big red X
- Null is useful to check if something has not been initialized yet
- We'll see examples of using `null` later

NullPointerException



```
30          int[] c = null;
31          System.out.println("C Length: " + c.length);
32
```

```
Welcome to DrJava.  Working directory is /home/dcx/Dropbo
> run RandEntry
java.lang.NullPointerException
     at RandEntry.main(RandEntry.java:31)
```

- This is a common run-time error
- Occurs when a reference variable has the value *null* and you try to access it
  - Example: Trying to access `c.length` if c is `null`
  - Or trying to print out the first element in c

Section 3

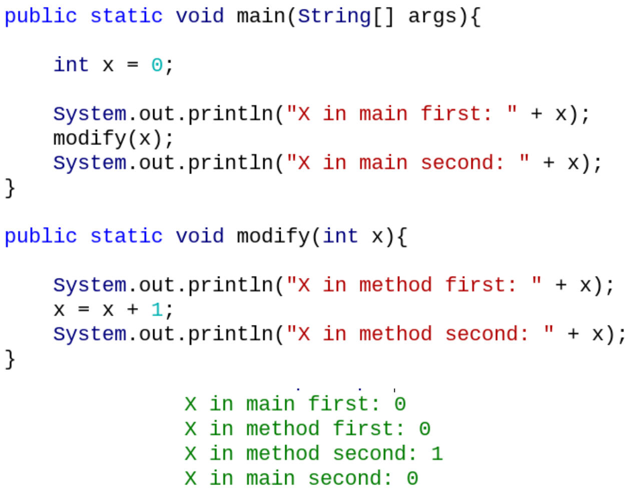
Swapping Values

Code for Swapping

```
int x = 5;
int y = 7;

System.out.println("X: " + x + " Y: " + y);
//X: 5 Y: 7

int temp = x;
x = y;
y = temp;

System.out.println("X: " + x + " Y: " + y);
//X: 7 Y: 5
```

Recall

- If you remember when we were talking about passing parameters to methods, parameters are **copied** to the method
- Here, the *x* variable is not changed in the main method



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

    int x = 0;

    System.out.println("X in main first: " + x);
    modify(x);
    System.out.println("X in main second: " + x);
}

public static void modify(int x){

    System.out.println("X in method first: " + x);
    x = x + 1;
    System.out.println("X in method second: " + x);
}
```

```
X in main first: 0
X in method first: 0
X in method second: 1
X in main second: 0
```

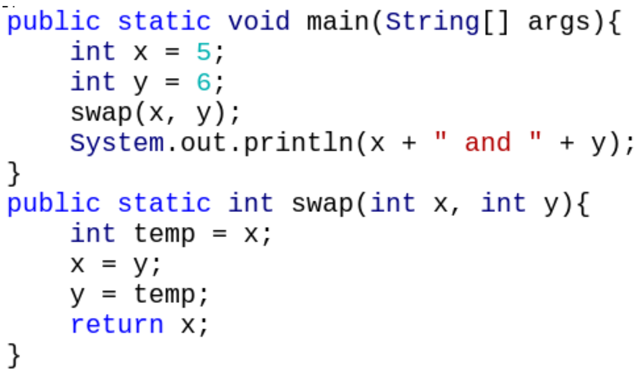
Swapping Primitive Types

```java
public static void main(String[] args){
    int x = 5;
    int y = 6;
    swap(x, y);
    System.out.println(x + " and " + y);
}
public static void swap(int x, int y){
    int temp = x;
    x = y;
    y = temp;
}
```

What prints? 5 and 6 Why?

- Value of x is copied to the first parameter of the method
- Value of y is copied to the second parameter of the method
- Values in the method's variables are changed, not in main's variables

Swap with Return



```java
public static void main(String[] args){
    int x = 5;
    int y = 6;
    swap(x, y);
    System.out.println(x + " and " + y);
}
public static int swap(int x, int y){
    int temp = x;
    x = y;
    y = temp;
    return x;
}
```

There's a *return* statement now

What prints? 5 and 6 Why?

- We are returning a value from swap, but it is not assigned to x or y
- Because we **copy** primitive variables when passing them, main's variables are different than swap's variables

Incorrect Swapping Values in Arrays

```
public static void main(String[] args){
    int[] a = {1,2,3,4,5};
    System.out.println(a[1]+ " and " + a[3]);

    swap(a[1], a[3]);
    System.out.println(a[1]+ " and " + a[3]);
}
public static void swap(int x, int y){
    int temp = x;
    x = y;
    y = temp;
}
```

What prints? 2 and 4   2 and 4   Why?

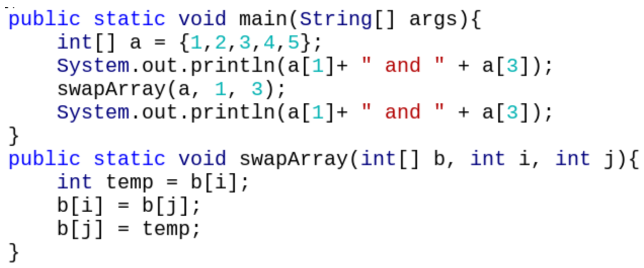
- We are passing two int values, which are copied to the parameters of swap
- Only the variables in `swap` are being swapped

Modifying Array Values

- Let's modify an element in an array
- We pass the value of a, which is an **address**
- Then we can change around the data in the array
- The `modifyArray` method is modifying data (in the computer's memory)

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

    int[] a = {1, 2, 3, 4};

    System.out.println("a[0] first: " + a[0]); //a[0] first: 1
    modifyArray(a);
    System.out.println("a[0] second: " + a[0]); //a[0] second: 2
}

public static void modifyArray(int[] a){
    a[0] = a[0] + 1;
}
```

Properly Swapping Array Values



```
public static void main(String[] args){
    int[] a = {1,2,3,4,5};
    System.out.println(a[1]+ " and " + a[3]);
    swapArray(a, 1, 3);
    System.out.println(a[1]+ " and " + a[3]);
}
public static void swapArray(int[] b, int i, int j){
    int temp = b[i];
    b[i] = b[j];
    b[j] = temp;
}
```

What prints? 2 and 4   4 and 2   Why?

- We pass the value of a, which is an **address**
- Then we can change around the data in the array
- The `swapArrays` method is moving around data (in the computer's memory)

Conclusion

Bottom line:

- Values are **copied** when they are passed as parameters
- Arrays store addresses
- You must pass an array to a method if you want to change values within the array inside the method

Creating a New Array

```
public static void changeContent(int[] arr) {
    // If we change the content of arr.
    arr[0] = 10; // The address of arr is passed, and we change the data inside
    //This changes main's array
}
public static void changeRef(int[] arr) {
    // If we change the reference
    arr = new int[2]; // this changes the address stored in the method's variable
    arr[0] = 15; //this refers to different data
}
public static void main(String[] args) {
    int [] arr = {4,5};
    System.out.println(arr[0]); //Will print 4.
    changeContent(arr); //pass address, data inside changes
    System.out.println(arr[0]); // Will print 10.
    changeRef(arr); //pass address, but data inside does not change
    System.out.println(arr[0]); // Will still print 10.
}
```

Section 4

Reference Type Examples

Copying Arrays

- How can we properly copy the values in one array into another?
- We saw that `int[] b = a;` does not work, it just copies the address
- We'll create a method to copy an array

Copying Arrays

```
public static void main(String[] args){
    int[] a = {7, 4, 5, 2, 6};
    int[] b = copyArray(a);

    System.out.println(Arrays.toString(b));
    //prints [7, 4, 5, 2, 6]

    b[0] = 100;
    System.out.println(Arrays.toString(b));
    //prints [100, 4, 5, 2, 6]
}

public static int[] copyArray(int[] arr){
    int[] result = new int[arr.length];
    for (int i=0; i < arr.length; i++){
        result[i] = arr[i];
    }
    return result;
}
```

Double the Value of Elements in an Array

- Let's create a method to double the values of elements in an array
  - Example:

$$1, 4, 7$$

  becomes

$$2, 8, 14$$

- We could use a method similar to copying arrays
  - Where we create a new array which has the doubled values
- We can also access the array values directly

Double the Value of Elements in an Array

```
public static void main(String[] args){
    int[] a = {7, 4, 5, 2, 6};
    doubleArray(a);

    System.out.println(Arrays.toString(a));
    //prints [14, 8, 10, 4, 12]
}

public static void doubleArray(int[] arr){
    for (int i=0; i < arr.length; i++){
        arr[i] = arr[i] * 2;
    }
}
```

- Method to double each element in the array
- Note that this method is `void`, but the elements are still modified

Section 5

Revisiting Strings

- Let's go back to `Strings`
- `Strings` are a reference type, but they are special
- The data pointed to by a `String` variable can't be changed after it has been created
  - `Strings` are **immutable**
  - Immutable means the data can't be changed

Changing Strings

- Haven't we changed Strings before?

```
String s = "apples";
s = s + " and bananas";

System.out.println(s);
```

Prints “apples and bananas”

Immutable Strings

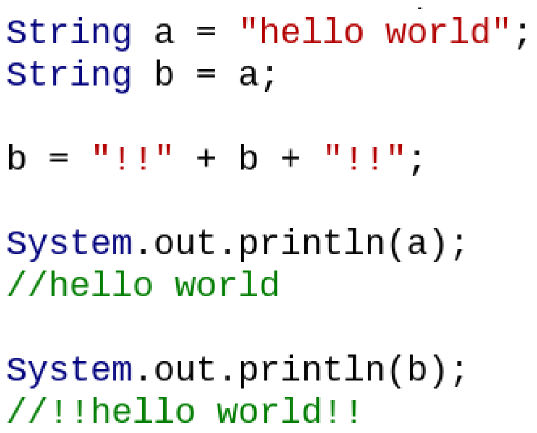
Behind the scenes, it looks more like this:

```
String s = "apples";
String temp = s + " and bananas";
s = temp;
```

- New `String` data is created with every concatenation
- The original variable then points to the new data

- Once the `String` “apple” has been created, there is no way to modify those characters

String Aliasing



```
String a = "hello world";
String b = a;

b = "!!" + b + "!!";

System.out.println(a);
//hello world

System.out.println(b);
//!!hello world!!
```

- Both a and b stored the same address
- But the address stored in b was automatically changed when the concatenation occurred

Immutable Strings

```
public static void main(String[] args){
    String s = "cats";
    changeString(s);
    System.out.println(s);
}

public static void changeString(String s){
    s = s + " and dogs";
}
```

What prints? *cats* Why?

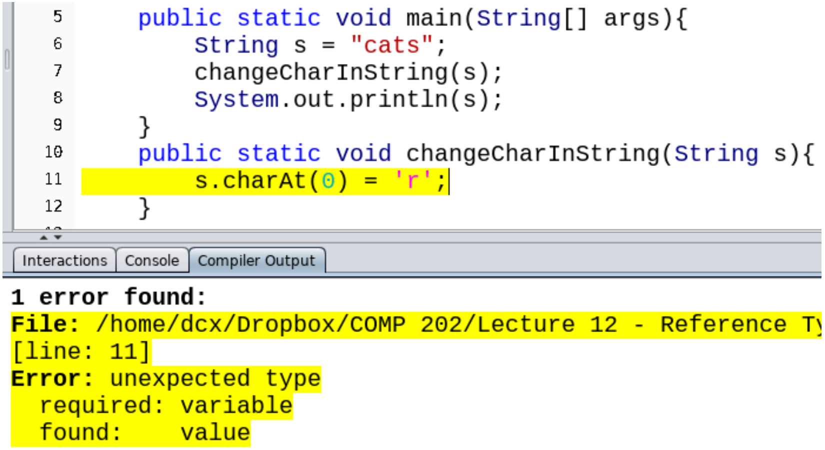
- The address of the String is copied to the method's parameters
- The method's variable is then updated with a new address to point to the new String

Immutable Strings

- Is it possible to change the characters within a `String`?
- No, we can't

- What about using the `charAt` method?

Immutable Strings


```
    public static void main(String[] args){
        String s = "cats";
        changeCharInString(s);
        System.out.println(s);
    }
    public static void changeCharInString(String s){
        s.charAt(0) = 'r';
    }
```

```
1 error found:
File: /home/dcx/Dropbox/COMP 202/Lecture 12 - Reference Ty
[line: 11]
Error: unexpected type
  required: variable
  found:    value
```

What prints? *There is a compile error* Why?

- `s.charAt(0)` returns a char 'c'
- Java can't assign a new value to 'c'

Recap

- **Primitive types store values**
  - `int, double, boolean, char`
- **Reference types store addresses**
  - `Strings`, arrays, objects
- But `Strings` are immutable
  - We can't change the data within a `String`
  - This means they act more like a primitive type

- This can be confusing, so make sure you understand the differences!
- Very common test question

Section 6

Two-Dimensional Arrays

Storing Sales

Let's look at a program for calculating and comparing sales.

```
String[] names = {"Bentley", "Melanie", "Giulia"};

double[] janSales = {560.34, 110.01, 200.34};
double[] febSales = {100, 150, 400};
double[] marchSales = {100.12, 150, 400};


for (int i=0; i < names.length; i++){
    System.out.println(names[i] +":");
    System.out.println(janSales[i] + "\t\t"
              + febSales[i]+ "\t\t" + marchSales[i]);
}
```

Storing Sales

Let's get the total sales for a month

```
double janSum = 0;
for (int i=0; i < janSales.length; i++){
    janSum += janSales[i];
}

System.out.println("January: " + janSum);
```

- To do this for every month is tedious
- We would have to copy and paste a lot
- Repeated code is bad, because it's easy to make mistakes

2D Arrays

Let's store the sales numbers for the first three months in the same variable
This will be an array of arrays - a *two-dimensional array*

- The inner arrays will store the sales by each person
- The outer array stores the inner arrays

Month Sales

```
String[] names = {"Bentley", "Melanie", "Giulia"};

double[][] firstQuarterSales = {
    {560.34, 110.01, 200.34}, //jan sales
    {100, 150, 400},          //feb sales
    {100.12, 150, 400}        //mar sales
};

for (int monthIndex=0; monthIndex < firstQuarterSales.length; monthIndex++){

    //get the month array
    double[] monthSales = firstQuarterSales[monthIndex];

    double monthSum = 0;
    for (int personIndex = 0; personIndex < monthSales.length; personIndex++){
        monthSum += monthSales[personIndex];
    }

    System.out.println("Month " + monthIndex + ": " + monthSum);
}
```

- We have one for-loop to go through the months
- One for-loop to count up the sales in the months

Arrays of Arrays

Three main ways to create arrays of arrays:

1. `int[][] a = {{1, 2, 3}, {5, 6}}`
2. `int[][] b = new int[4][5];`
3. `int[][] c = new int[3][];`

Hard-coding Values

```
int[][] a = {{1, 2, 3}, {5, 6}}
```

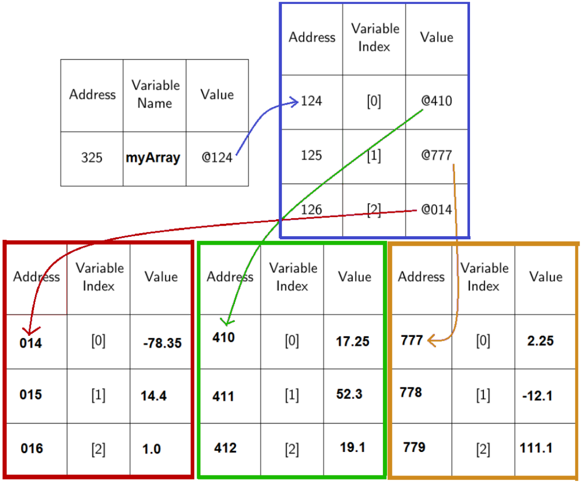
- Creates an array of length 2
- The first element in the array is another array of length 3 (with values 1,2,3).
- The second element is an array of length 2 (with values 5,6).

Arrays as Reference Types

```
double[][] myArray = { {17.25, 52.3, 19.1},
                {2.25, -12.1, 111.1},
                {-78.35, 14.4, 1.0} };
```

Let's see what this looks like in terms of reference types.
That is, how many addresses do we have here?

Address Diagram



Address	Variable Name	Value
325	**myArray**	@124

Address	Variable Index	Value
124	[0]	@410
125	[1]	@777
126	[2]	@014

Address	Variable Index	Value
014	[0]	**-78.35**
015	[1]	**14.4**
016	[2]	**1.0**

Address	Variable Index	Value
410	[0]	**17.25**
411	[1]	**52.3**
412	[2]	**19.1**

Address	Variable Index	Value
777	[0]	**2.25**
778	[1]	**-12.1**
779	[2]	**111.1**

```
double[][] myArray = { {17.25, 52.3, 19.1},
                       {2.25, -12.1, 111.1},
                       {-78.35, 14.4, 1.0} };
```

Creating An Empty Array

Recall the default value placed in arrays when created:

- int/double: 0
- boolean: false
- char: ASCII value 0
- Reference types: null

```
int[] arr = new int[4];
```

Creates an array of length 4, where all elements are initialized to 0.

Creating An Empty Array of Arrays

```
int[][] grid = new int[4][5];
```

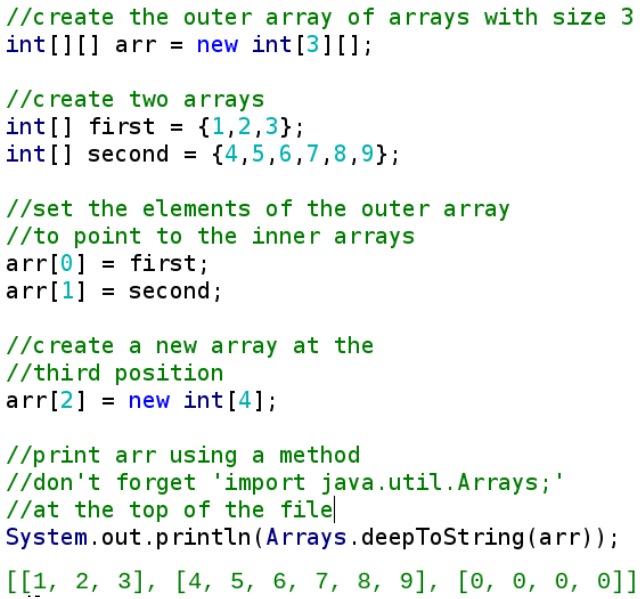
- Creates an array of length 4, where each element in that array is another array of length 5
- This creates a grid (or matrix), which might represent rows and columns of something
- Because the array store ints, every entry will start off with a value of 0

Creating an Empty Array of Arrays

```
int[][] arr = new int[3][];
```

- This creates an array of length 3.
- Each element in the array will be an array.
- However, what is currently stored is the value *null*
  - Represents an uncreated array
- We need to create three integer arrays and place them in `arr`

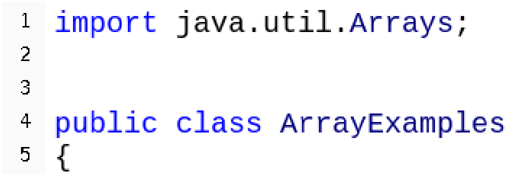
Inputting Elements in the Array



```
//create the outer array of arrays with size 3
int[][] arr = new int[3][];

//create two arrays
int[] first = {1,2,3};
int[] second = {4,5,6,7,8,9};

//set the elements of the outer array
//to point to the inner arrays
arr[0] = first;
arr[1] = second;

//create a new array at the
//third position
arr[2] = new int[4];

//print arr using a method
//don't forget 'import java.util.Arrays;'
//at the top of the file
System.out.println(Arrays.deepToString(arr));
```

```
[[1, 2, 3], [4, 5, 6, 7, 8, 9], [0, 0, 0, 0]]
```

Import Statements



```
import java.util.Arrays;


public class ArrayExamples
{
```

- The top line is an example of an **import statement**
- It's required so that we can use methods in the Arrays class
- We'll also need this later for reading/writing from files
- All your import statements go at the top of your .java file, before your `public class`

Built-in Methods

- `Arrays.deepToString(arr)` $\rightarrow$ prints all of the elements of `arr`, where `arr` is a multi-dimensional array
- `Arrays.deepEquals(arr1, arr2)` $\rightarrow$ Two array references are considered deeply equal if both are null, or if they refer to arrays that contain the same number of elements and all corresponding pairs of elements in the two arrays are deeply equal

As before, you may be asked to write the code for these methods on an exam

Jagged Arrays

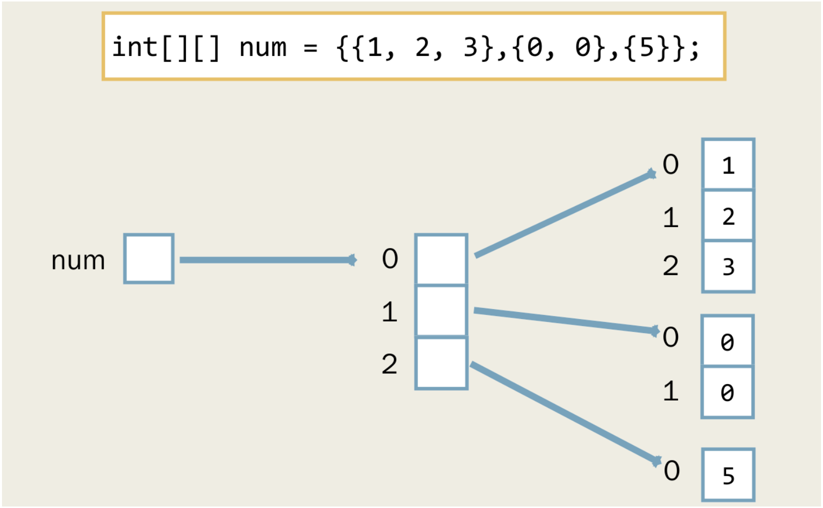
- Sometimes, we may have arrays of arrays of different length

For example:

```
int[][] arr = {{1, 4, 5 }, {52}, {} }
```

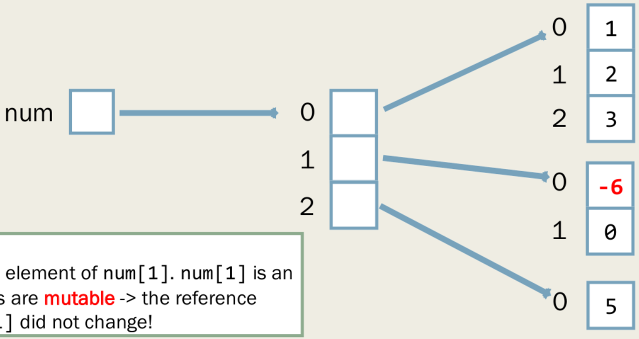
The array contains three arrays, of size 3, 1, and 0.

Changing Elements


```
int[][] num = {{1, 2, 3},{0, 0},{5}};
```

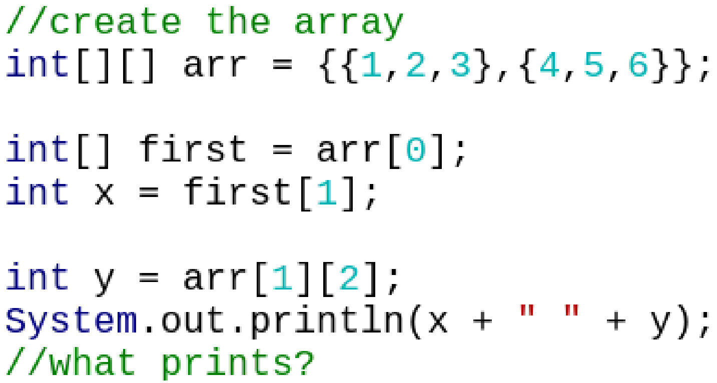
Changing Elements

```
int[][] num = {{1, 2, 3},{0, 0},{5}};
num[1][0] = -6;
```



Note:
- We changed an element of `num[1]`. `num[1]` is an array and arrays are **mutable** -> the reference stored in `num[1]` did not change!

2D Arrays



```java
//create the array
int[][] arr = {{1,2,3},{4,5,6}};

int[] first = arr[0];
int x = first[1];

int y = arr[1][2];
System.out.println(x + " " + y);
//what prints?
```

2 6

The second element of the first array, and the third element of the second array

Higher Dimensional Arrays

You can create higher dimensional arrays:

```
int[][][][][][][][][][] x = new
int[2][2][2][2][2][2][2][2][2][2];
```

Anything larger than 3D is very rare

Higher Dimensional Arrays

If we had a method that takes as input an `int[][][]` and returns the largest value in the array:
How many loops do you need to do this?

Answer: Three nested loops
One for the outer array, one for the middle array, and one for the inner array

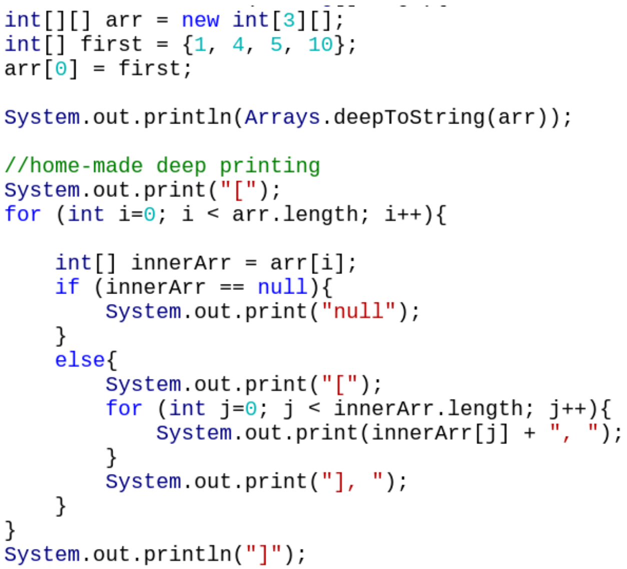
Multi-Dimensional Arrays

```
int[][][] arr = {
    {
            {1,2,3},{4,5,6},{5,5,5},{1,9,0}
    },
    {
            {3,4,8},{9,1,2},{9,5,1},{3,4,3}
    }
};
System.out.println(arr.length);          //2
System.out.println(arr[0].length);       //4
System.out.println(arr[0][0].length);    //3
System.out.println(arr[1][2][0]);        //9
```

Section 7

2D Array Examples

Deep Printing



```java
int[][] arr = new int[3][];
int[] first = {1, 4, 5, 10};
arr[0] = first;

System.out.println(Arrays.deepToString(arr));

//home-made deep printing
System.out.print("[");
for (int i=0; i < arr.length; i++){

    int[] innerArr = arr[i];
    if (innerArr == null){
        System.out.print("null");
    }
    else{
        System.out.print("[");
        for (int j=0; j < innerArr.length; j++){
            System.out.print(innerArr[j] + ", ");
        }
        System.out.print("], ");
    }
}
System.out.println("]");
```

Sum All Numbers

Write a method that takes as input a 2D array of doubles and returns the sum of all of the numbers in the array.

Sum All Numbers

```
public static void main(String[] args){
    double[][] arr = {
        {134.6, 1235.2, 1314.5},
        {1934, 134.1, 13923.4324, 434},
        {1323, 1},
        {},
        {34343,234,24,2426,47,47}
    };

    double sum = getSum(arr);
    System.out.println("Sum: " + sum);
}

public static double getSum(double[][] arr){
    double sum = 0;
    for (int outer = 0; outer < arr.length; outer++){
        for (int inner = 0; inner < arr[outer].length; inner++){
            sum += arr[outer][inner];
        }
    }
    return sum;
}
```

Swapping Arrays

```
public static void main(String[] args){
    int[][] arr = {{1,2},{3,4}};
    //swap the two inner arrays
    swap(arr, 0, 1);
    //print out the inner arrays
    System.out.println(Arrays.toString(arr[0]) + " " +
                       Arrays.toString(arr[1]));
}
//pass the array of arrays, and the two indices
public static void swap(int[][] b, int i, int j){
    //switch the arrays
    int[] temp = b[i];
    b[i] = b[j];
    b[j] = temp;
}
```

What prints?

```
[3, 4] [1, 2]
```

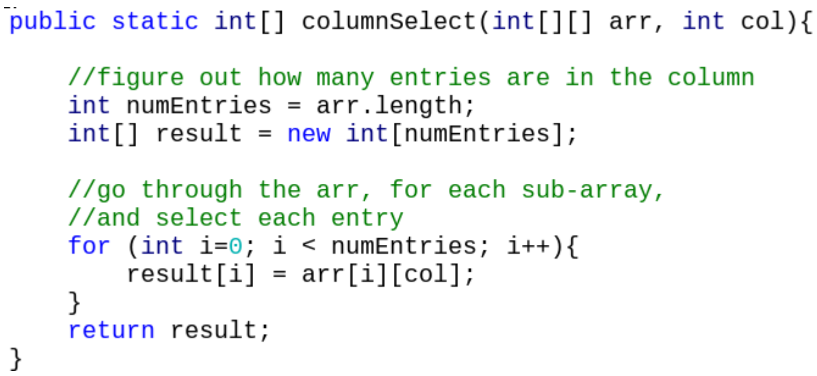
Selecting a Column

Write a method that takes as input a 2D integer array and an integer number representing a column index. It should return the column at that index.
For example, if the input array is:

```
int[] arr = {
{1,2,3},
{5,6,7},
{9,2,1},
{0,3,0}
};
```

And the index is 1, it should return the array. `{2,6,2,3}`

columnSelect



```
public static int[] columnSelect(int[][] arr, int col){

    //figure out how many entries are in the column
    int numEntries = arr.length;
    int[] result = new int[numEntries];

    //go through the arr, for each sub-array,
    //and select each entry
    for (int i=0; i < numEntries; i++){
        result[i] = arr[i][col];
    }
    return result;
}
```

Multiply by a Value

Write a method `multiplyMatrix` that takes as input one 2D array representing a matrix, as well as a double value. Multiply each element in the array by that value and do not return anything.

multiplyMatrix

```
public static void main(String[] args){
    double[][] matrix = {
        {1,4,5},
        {4,9,5},
        {3.4,5.4,34}
    };

    System.out.println(Arrays.deepToString(matrix));
    multiplyMatrix(matrix, 3.4);
    System.out.println(Arrays.deepToString(matrix));
}

public static void multiplyMatrix(double[][] mx, double val){

    for (int i = 0; i < mx.length; i++){

        double[] innerMx = mx[i];
        for (int j = 0; j < innerMx.length; j++){
            innerMx[j] *= val;
        }
    }
}
```

Practice

Write `multiplyMatrix` again, but make a copy of the input, multiply each element of the copy, and return that instead. Do not modify the original.

Be sure to copy at the level of primitive types! (Don't copy any references)
Throw an exception if the input is not rectangular, or if any of the 'sub-arrays' are null.

sumMatrix

Write a method sumMatrix that takes two 2D integer arrays as input with the same dimensions. The method should return a new 2D integer array corresponding to their sum.

Example: consider the following 2D arrays

```
int[][] matrix1 = {{2,3}, {5,1}};
int[][] matrix2 = {{-1,5}, {2,-4}};
```

Then sumMatrix(matrix1, matrix2) should return the 2D array

```
{{1, 8}, {7, -3}}
```

sumMatrices

```
public static int[][] sumMatrices(int[][] mx1, int[][] mx2){

    int size = mx1.length;

    int[][] result = new int[size][size];

    for (int i = 0; i < size; i++){
        for (int j = 0; j < size; j++){
            result[i][j] = mx1[i][j] + mx2[i][j];
        }
    }
    return result;
}
```

Practice

Write `sumMatrices` again, but make a copy of the first array and add each element of the second array to the copy, and return that instead. Do not modify the original.

Be sure to copy at the level of primitive types! (Don't copy any references)
Throw an exception if any of the 'sub-arrays' are null.