YEAHtzee

Tuesday Nov 6 2018

Aamnah Khalid and
Peter Maldonado

ArrayLists

- Why ArrayLists?
  - Your may not know how much data you need
- ArrayLists are dynamically sized (grow as needed)
- Can check if something is in it with `.contains()`
- Can only store objects (no primitives!)
  - Must use wrapper classes (e.g. `Integer` for `int`) as substitute for primitives

```
ArrayList<Type> myList = new
ArrayList<Type>();
```

`ArrayList` Methods

`boolean add(<T> element)` Adds a new element to the end of the **`ArrayList`**; the return value is always **`true`**.
`void add(int index, <T> element)` Inserts a new element into the **`ArrayList`** before the position specified by **`index`**.
`<T> remove(int index)` Removes the element at the specified position and returns that value.
`boolean remove(<T> element)` Removes the first instance of **`element`**, if it appears; returns **`true`** if a match is found.
`void clear()` Removes all elements from the **`ArrayList`**.
`int size()` Returns the number of elements in the **`ArrayList`**.
`<T> get(int index)` Returns the object at the specified index.
`<T> set(int index, <T> value)` Sets the element at the specified index to the new value and returns the old value.
`int indexOf(<T> value)` Returns the index of the first occurrence of the specified value, or −1 if it does not appear.
`boolean contains(<T> value)` Returns **`true`** if the **`ArrayList`** contains the specified value.
`boolean isEmpty()` Returns **`true`** if the **`ArrayList`** contains no elements.

Arrays

- Why Arrays?
  - Great for representing a **fixed-size list**
  - We want to use the most efficient data structure possible
- Store data at different indices in the array, and then look up by index
- Can only store both objects and primitives!

```
Type[] myArray = new Type[SIZE];
```

ArrayLists

- Variable length
- Store only objects
- Class with methods like `.contains`

Both

- Store sequences of data
- Homogeneous (single type)

Arrays

- Fixed length (specify when created)
- Store objects and primitives
- Only attribute is `.length`

Array Operations

- To create a new array we to specify `Type` and `SIZE` in a call to `new`

```
Type[] myArray = new Type[SIZE];
```

- To access an element in the array, use the square brackets to choose the index

```
myArray[index]
```

  - This evaluates to a **reference** to that position in the array
- While arrays don't have methods, they have a single field for their length

```
myArray.length
```

2D Arrays (Grids)

What if `Type` is an array? (e.g. `int[]`)

```
Type[] myGrid = new
Type[numElems];
```

```
Type[] myGrid = new
Type[numElems];
```

Let's try setting `Type` to `int[]`

```
int[][] myGrid = new
int[numArrs][numElems];
```

```
int[][] myGrid = new int[arrays][elems];
```

- This works since each `int[]` is an object
- We make an array of integer arrays!
  - This is why we say our array spans two dimensions
- Note: we must specify size of each array (`numArrs` and `numElems`)

```
Type[][] myGrid = new
Type[rows][cols];
```

- Each row is an array
- Each column represents an index that exists in each row array
- Each element is at a specific column in a specific row!

Interpreting Multidimensional Arrays

- As 2D Grid
  - Looking up `arr[row][col]` selects the element in the array at position `(row, col)`
- As an array of arrays
  - Looking up `arr[n]` gives back a one-dimensional array representing the `n+1`–th row
    - Remember we get a reference to that array!
  - First dimension indexes different arrays, second dimension indexes elements in a particular array

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

What does this evaluate to? (What is returned?)

`multiArr[1]`

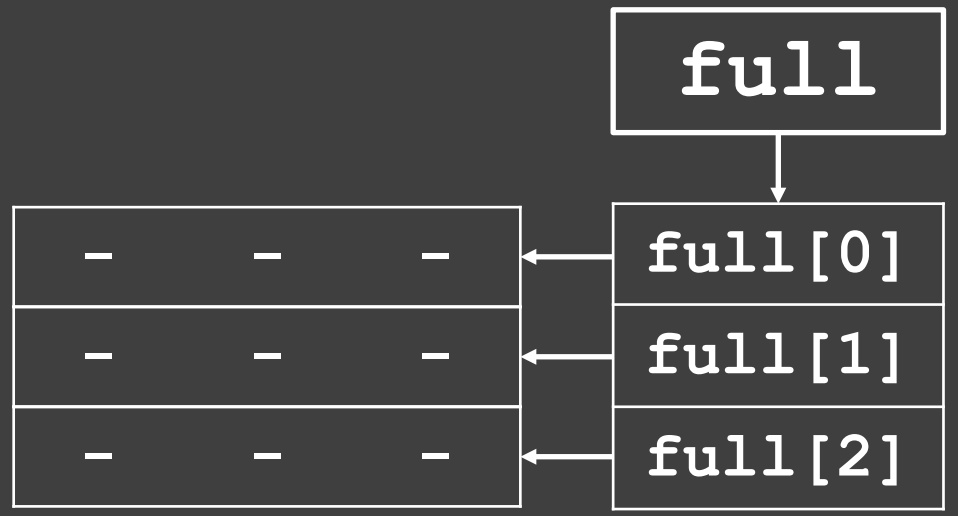
`int[5]` -> a reference to an array of five integers

`multiArr[2][3]`

`int` -> a single integer value

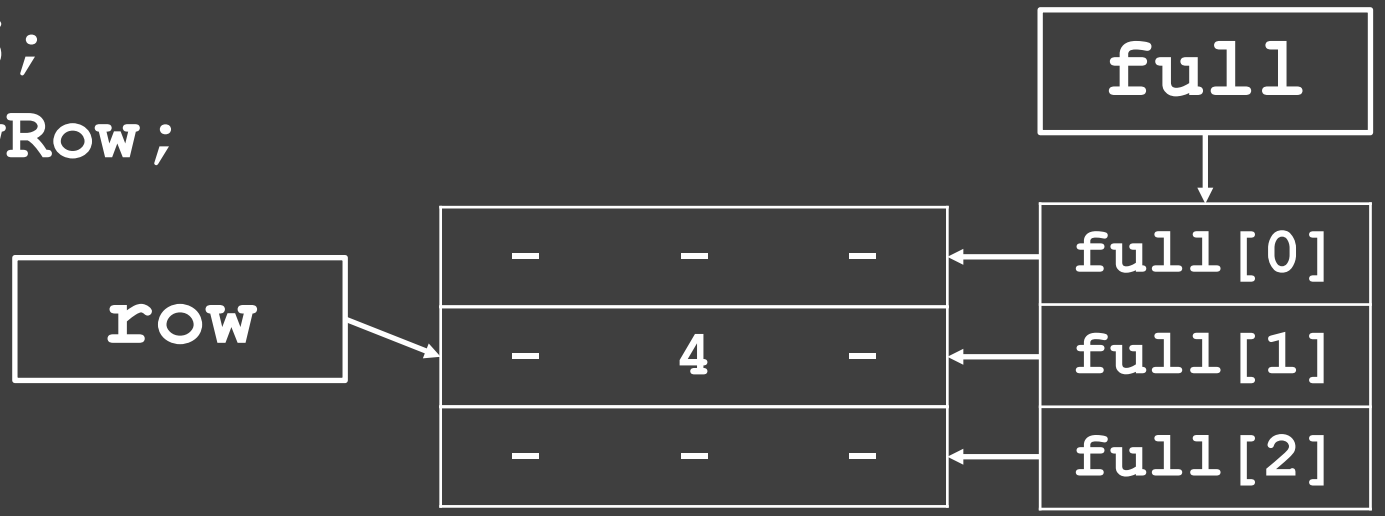
Accessing Arrays Uses References!

```
public class HelloWorld {
    public static void main(String []args) {
        int[][] full = new int[3][3];
        int[] row = full[1];
        row[1] = 4;
        int[] newRow = new int[3];
        newRow[0] = 6;
        full[2] = newRow;
    }
}
```



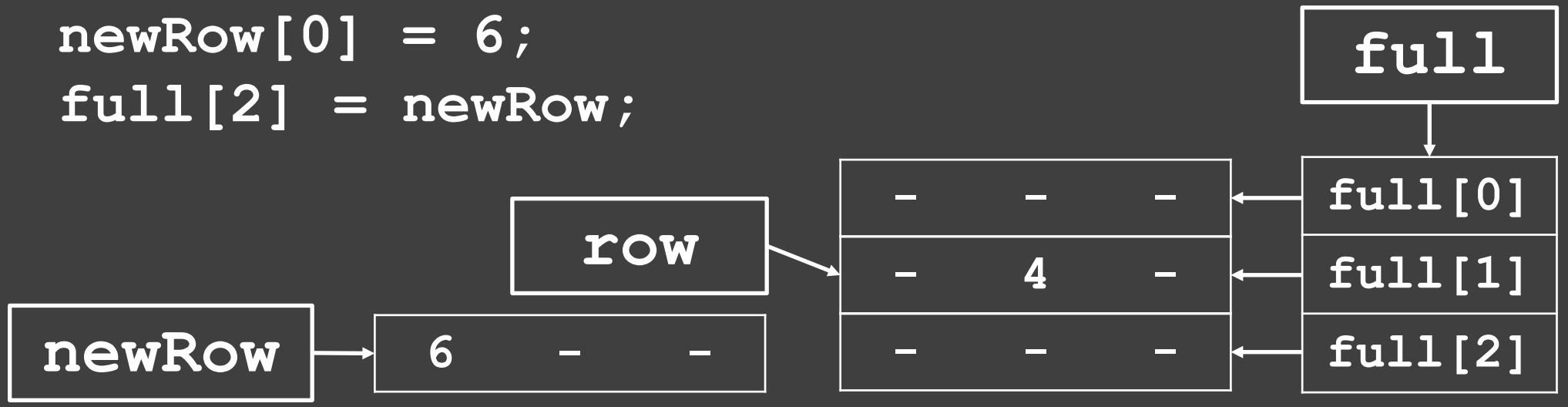
Accessing Arrays Uses References!

```
public class HelloWorld {
    public static void main(String []args) {
        int[][] full = new int[3][3];
        int[] row = full[1];
        row[1] = 4;
        int[] newRow = new int[3];
        newRow[0] = 6;
        full[2] = newRow;
    }
}
```



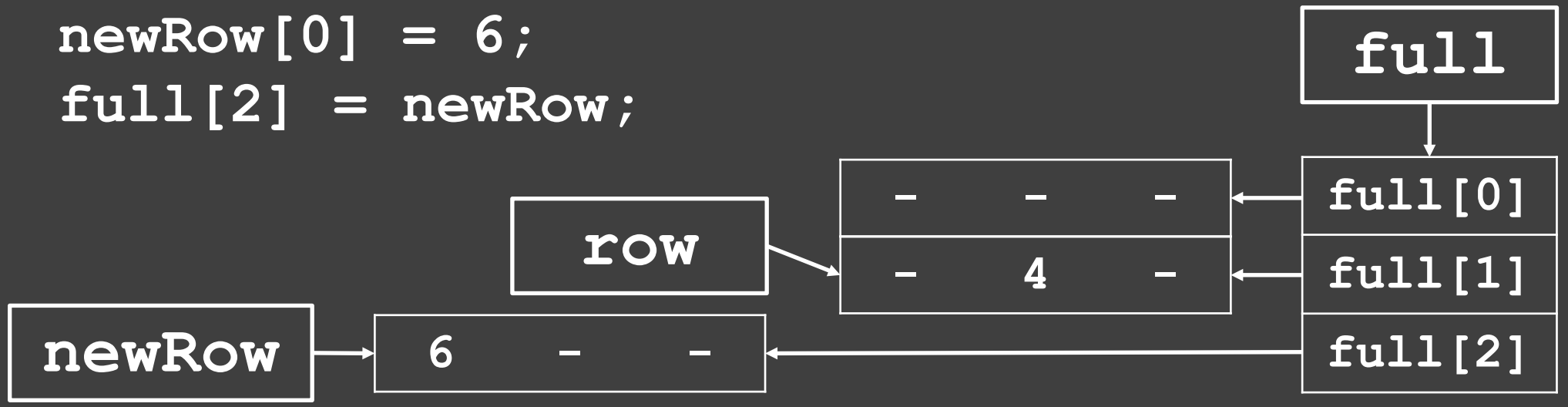
Accessing Arrays Uses References!

```
public class HelloWorld {
    public static void main(String []args) {
        int[][] full = new int[3][3];
        int[] row = full[1];
        row[1] = 4;
        int[] newRow = new int[3];
        newRow[0] = 6;
        full[2] = newRow;
    }
}
```



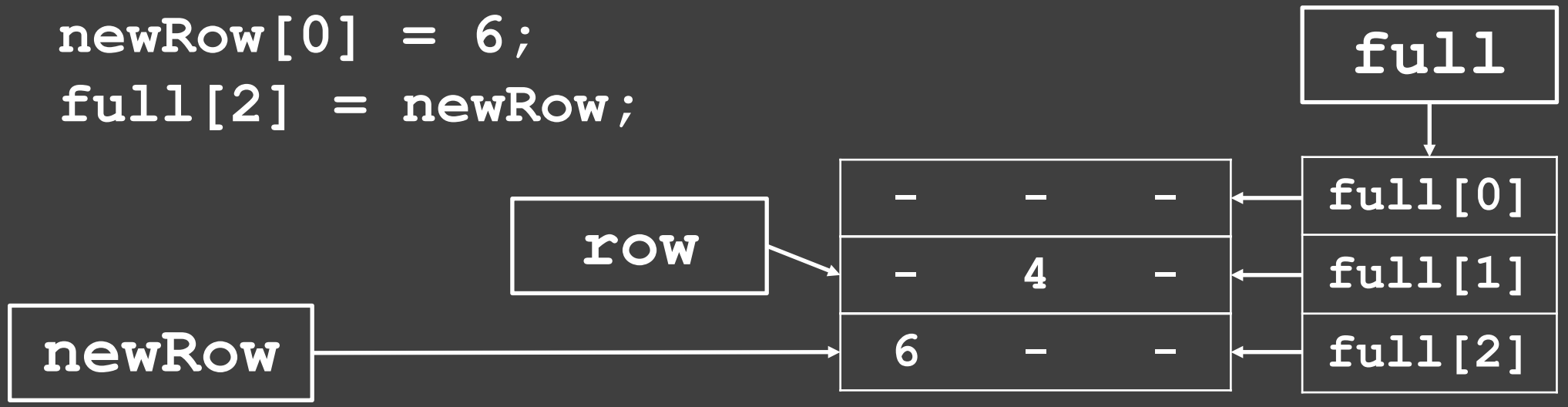
Accessing Arrays Uses References!

```
public class HelloWorld {
    public static void main(String []args) {
        int[][] full = new int[3][3];
        int[] row = full[1];
        row[1] = 4;
        int[] newRow = new int[3];
        newRow[0] = 6;
        full[2] = newRow;
    }
}
```

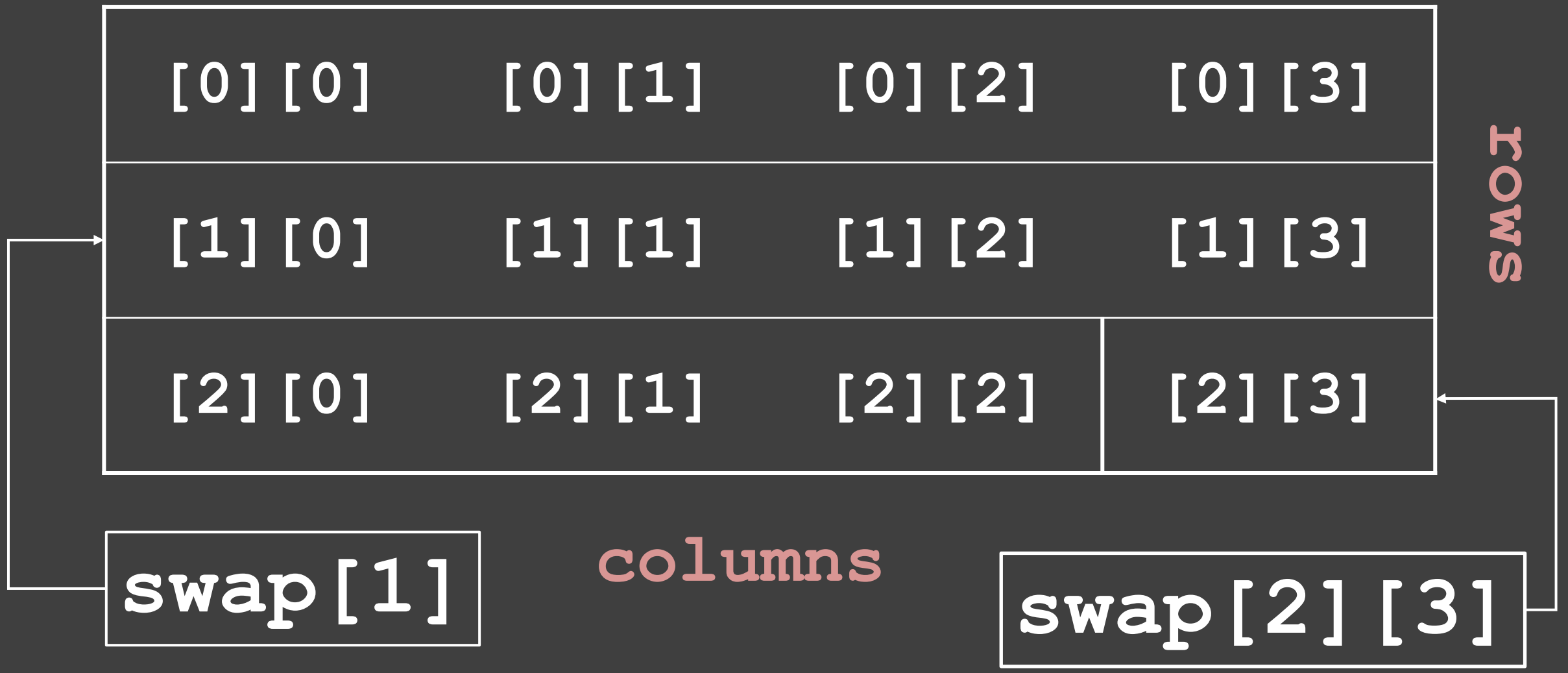


Accessing Arrays Uses References!

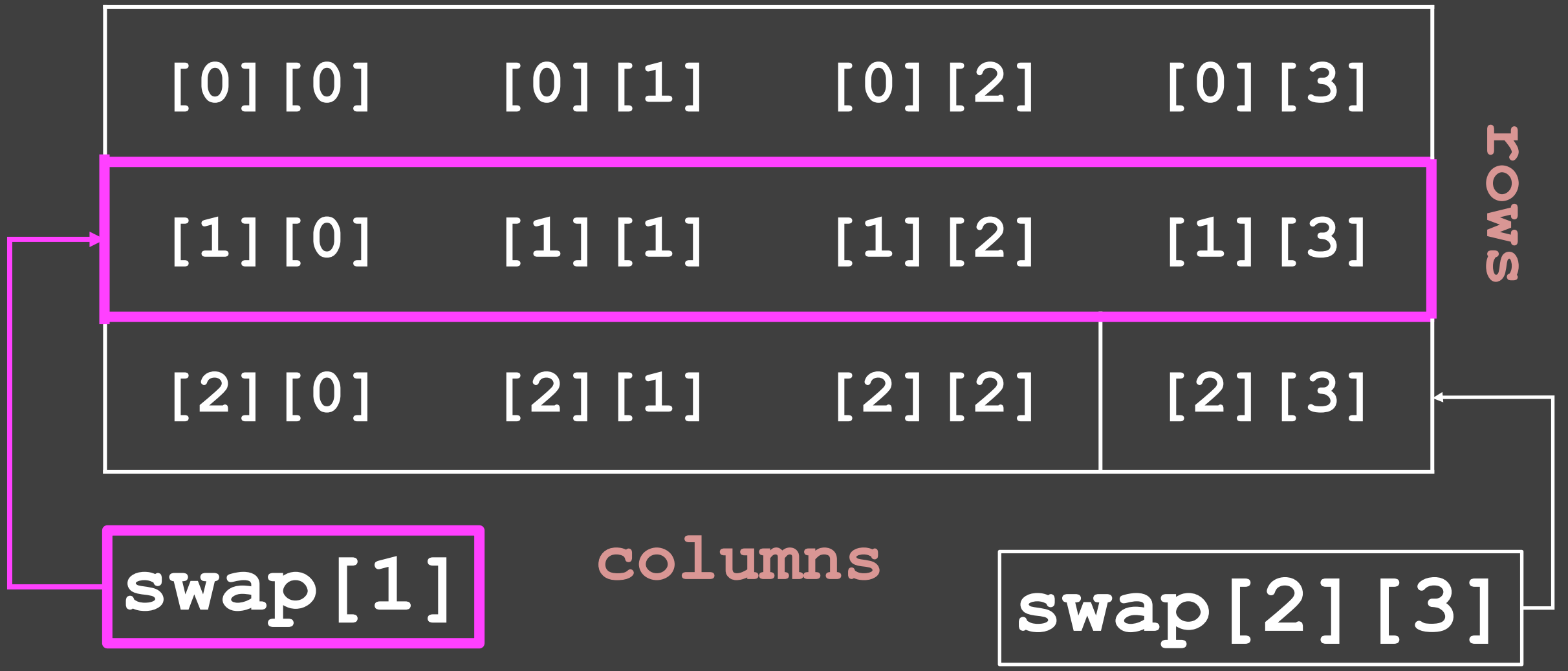
```
public class HelloWorld {
    public static void main(String []args) {
        int[][] full = new int[3][3];
        int[] row = full[1];
        row[1] = 4;
        int[] newRow = new int[3];
        newRow[0] = 6;
        full[2] = newRow;
    }
}
```



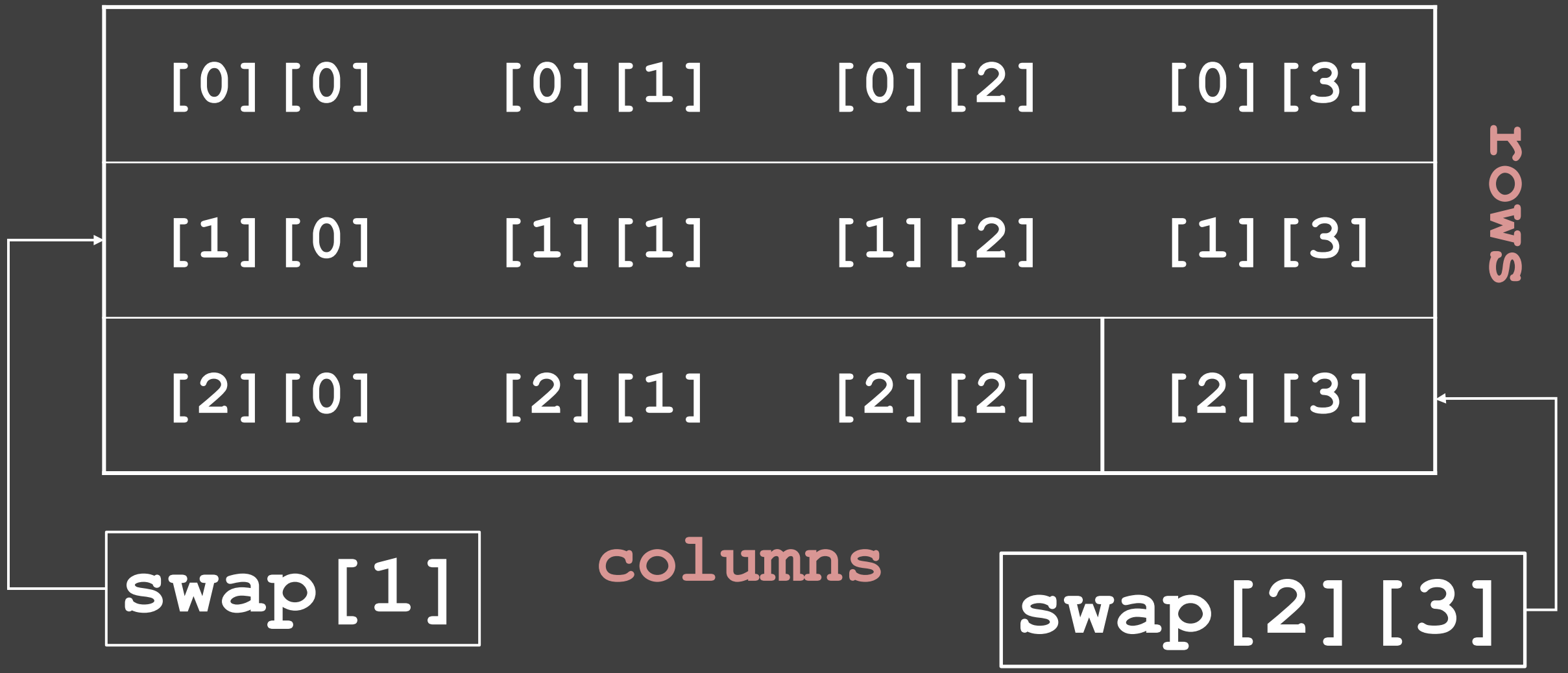
```
int[][] swap = new int[3][4];
```



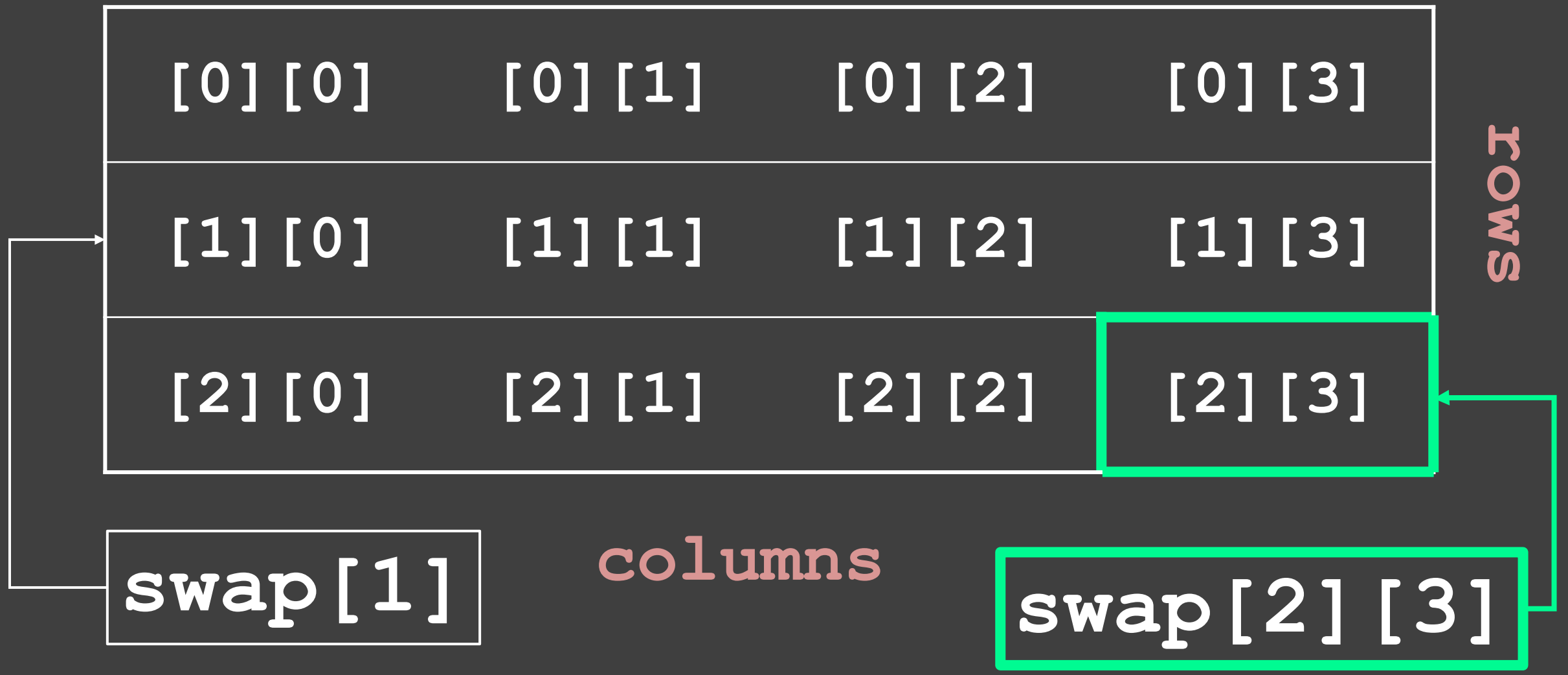
```
int[][] swap = new int[3][4];
```



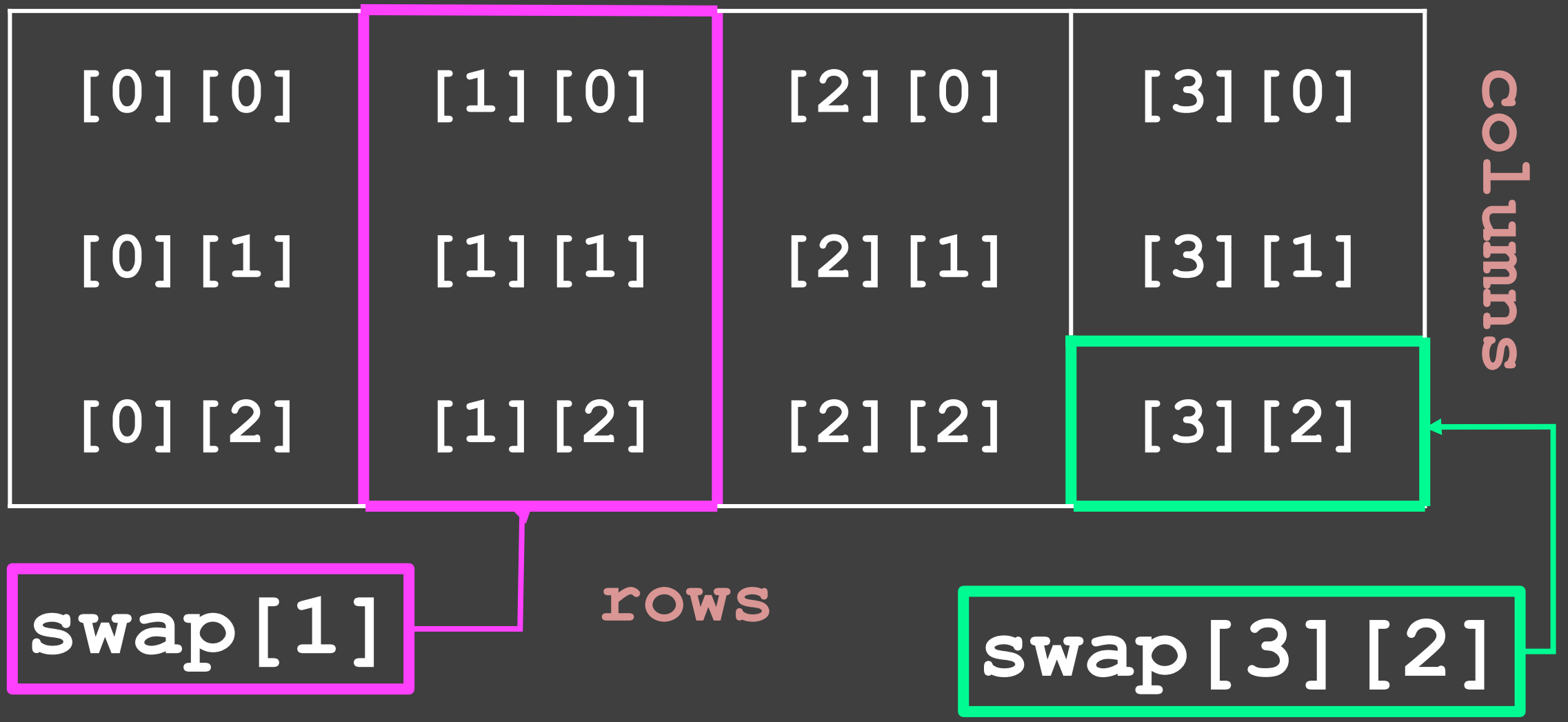
```
int[][] swap = new int[3][4];
```



```
int[][] swap = new int[3][4];
```



```
int[][] swap = new int[4][3];
```



Dimension Swapping

- Swapping dimension sizes switches which dimension of a grid is represented as 1D arrays
  - Remember 2D arrays always follow

```
Type[][] myGrid = new
Type[numArrs][numElems];
```

- Doesn’t change total elements in grid, flips rows and columns

```
Type[][] myGrid = new Type[rows][cols];
```

- Number of rows is number of arrays, number of cols is number of elements in each array!

Iterating Through a 2D Array

```
Type[][] arr = /* ... */
for (int row = 0; row < arr.length; row++) {
    for (int col = 0; col < arr[row].length; col++) {
        /* access arr[row][col] ... */
    }
}
```

2D Array Example

```
double[][] arr = new double[2][3]
for (int row = 0; row < arr.length; row++) {
    for (int col = 0; col < arr[row].length; col++) {
        arr[row][col] = col / (double)(row + 1);
    }
}
```

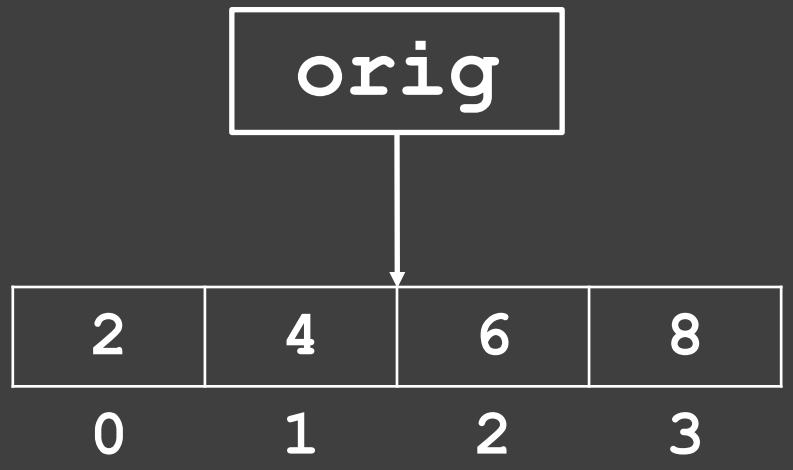
0.0	1.0	2.0
0.0	0.5	1.0

Another 2D Array Example

```
int[][] arr = new int[2][3]
for (int row = 0; row < arr.length; row++) {
    for (int col = 0; col < arr[row].length; col++) {
        if (row == col) {
            arr[row][col] = 1;
        } else {
            arr[row][col] = 0;
        }
    }
}
```

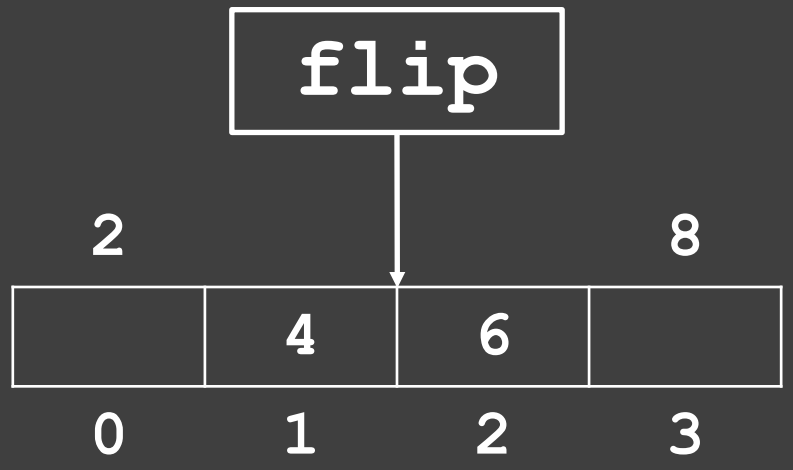
1	0	0
0	1	0

Flipping an Array!



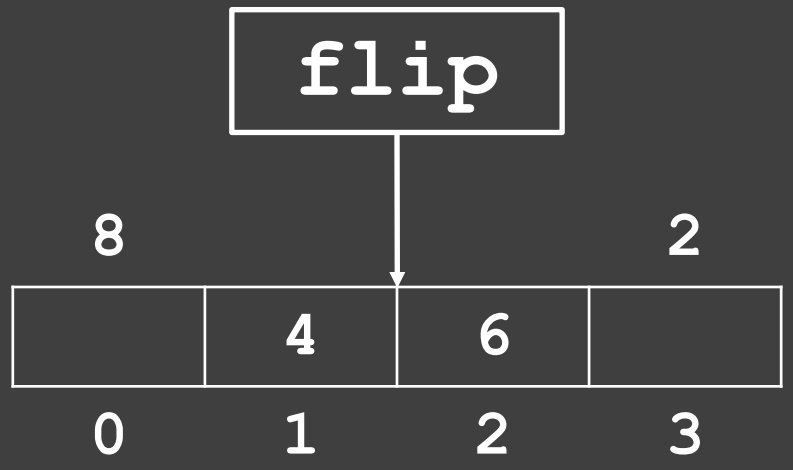
- How many times do we flip elements? What is this number relative to the size of the array?

Flipping an Array!



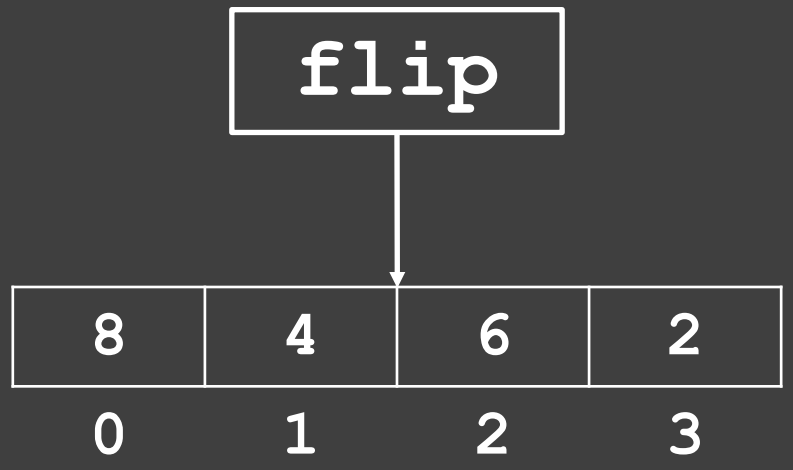
- How many times do we flip elements? What is this number relative to the size of the array?

Flipping an Array!



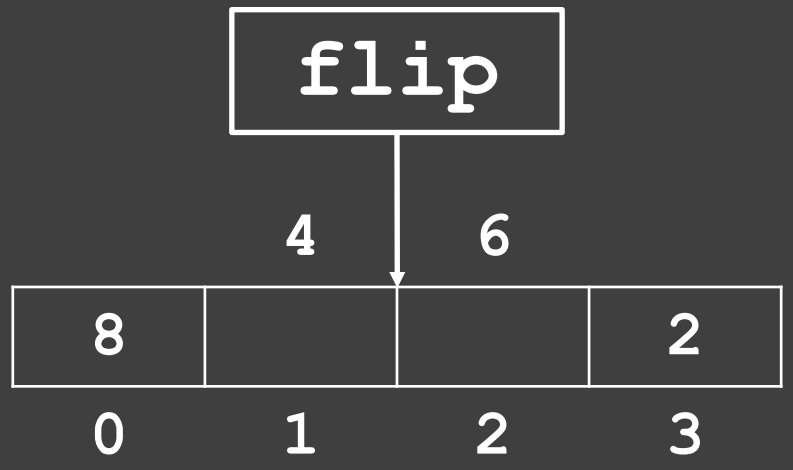
- How many times do we flip elements? What is this number relative to the size of the array?

Flipping an Array!



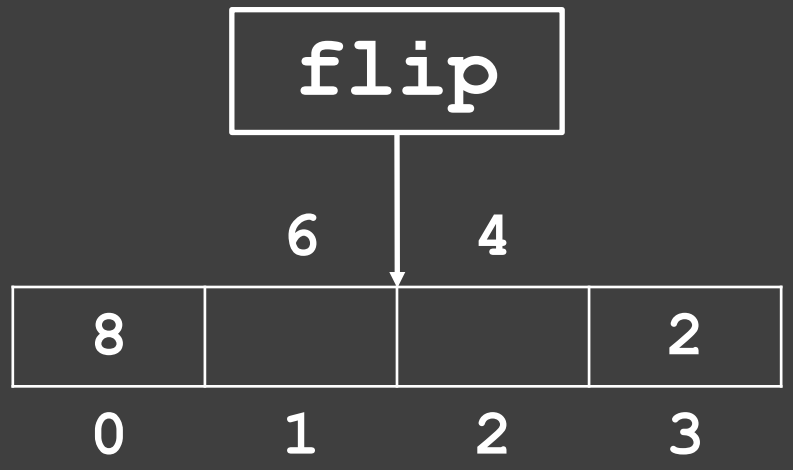
- How many times do we flip elements? What is this number relative to the size of the array?

Flipping an Array!



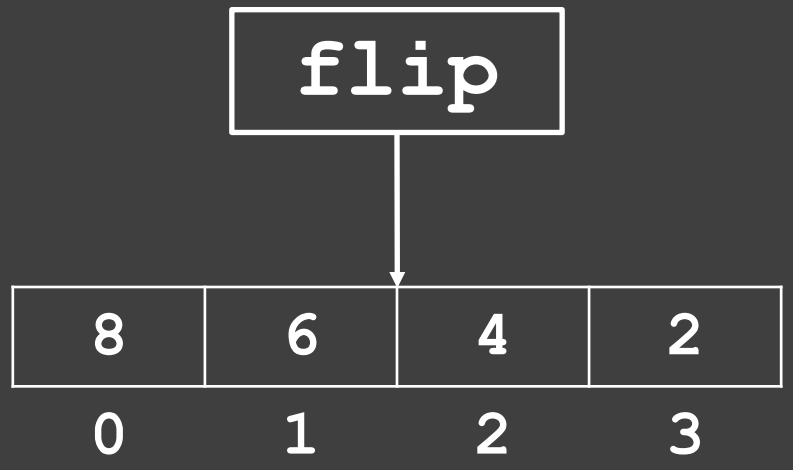
- How many times do we flip elements? What is this number relative to the size of the array?

Flipping an Array!



- How many times do we flip elements? What is this number relative to the size of the array?

Flipping an Array!



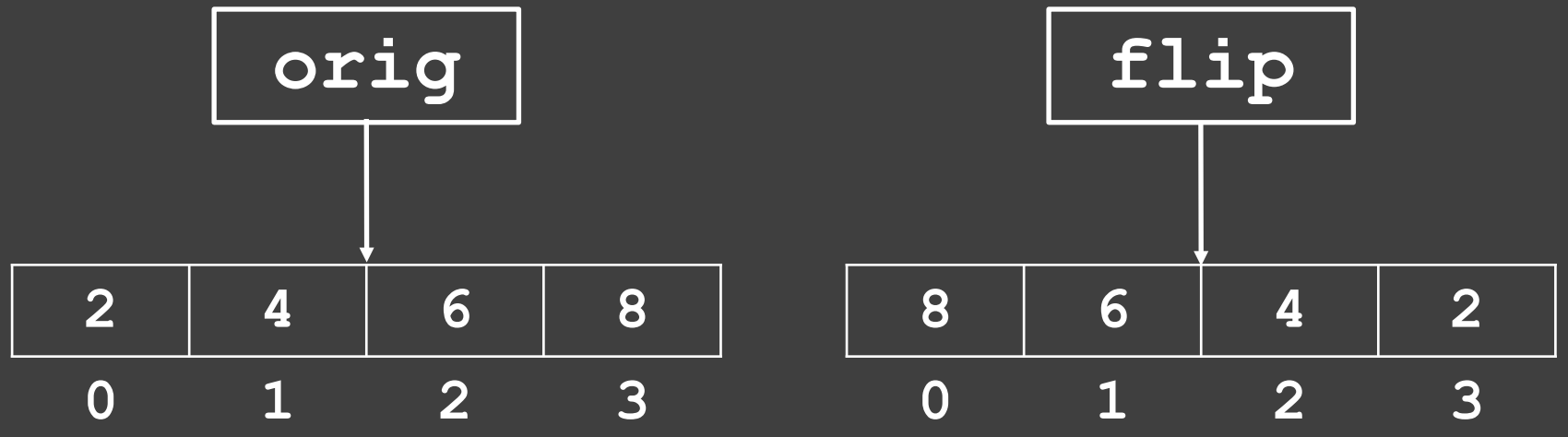
- How many times do we flip elements? What is this number relative to the size of the array?

Flipping an Array!

- How many times do we flip elements? What is this number relative to the size of the array?
  - Only need to perform `orig.length / 2` swaps!
  - Each element in the front half of the array swapped with corresponding element in back half

Flipping an Array!

```
private void flip(int[] toFlip) {
    for (int i = 0; i < toFlip.length / 2; i++) {
        int tmp = toFlip[i];
        toFlip[i] = toFlip[toFlip.length - i - 1];
        toFlip[toFlip.length - i - 1] = tmp;
    }
}
```



Flipping a 2D Array!

```
private void flip(int[][] toFlip) {
    for (int i = 0; i < toFlip.length / 2; i++) {
        int[] tmp = toFlip[i]; /* we swap integer
                                  arrays instead! */
        toFlip[i] = toFlip[toFlip.length - i - 1];
        toFlip[toFlip.length - i - 1] = tmp;
    }
}
```

Yahtzee Assignment

DUE at 1:30PM on Wednesday, Nov. 14

Graphics already implemented for you

Practice with arrays

Demo

What is provided in the starter project?

- **`Yahtzee.java:`** Initialization code provided, expand to play the game.
- **`YahtzeeConstants.java:`** defines several constants used in the game. Note the **category** constants at the end of the file
- **`YahtzeeDisplay:`** manages all the graphics and event handling (Check out the JavaDoc on website)
- **`YahtzeeMagicStub:`** exports a method **`checkCategory`** that will allow you to get your program working a little sooner. **Eventually you need to write this method yourself!** (Check out the JavaDoc on website)

```
/* The constants that specify categories on the
scoresheet */
  public static final int ONES = 1;
  public static final int TWOS = 2;
  public static final int THREES = 3;
  public static final int FOURS = 4;
  public static final int FIVES = 5;
  public static final int SIXES = 6;
  public static final int UPPER_SCORE = 7;
  public static final int UPPER_BONUS = 8;
  public static final int THREE_OF_A_KIND = 9;
  public static final int FOUR_OF_A_KIND = 10;
  public static final int FULL_HOUSE = 11;
  public static final int SMALL_STRAIGHT = 12;
  public static final int LARGE_STRAIGHT = 13;
  public static final int YAHTZEE = 14;
  public static final int CHANCE = 15;
  public static final int LOWER_SCORE = 16;
  public static final int TOTAL_= 17;
```

Yahtzee Display Class Methods

`public YahtzeeDisplay(GCanvas gc, String[] playerNames)`
`public void waitForPlayerToClickRoll(int player)`
`public void displayDice(int[] dice)`
`public void waitForPlayerToSelectDice()`
`public boolean isDieSelected(int index)`
`public int waitForPlayerToSelectCategory()`
`public void updateScorecard(int category, int player, int score)`
`public void printMessage(String message)`

Game Setup

- User enters the names of the players, one at a time
- Up to 4 players
- Methods in the **`YahtzeeDisplay`** class take player numbers that run from 1 to the number of players (not starting from **0**)

Each Turn

- Each player takes a turn:
  - Rolls dice 1st time
  - Selects a set of dice to reroll (if any) and reroll
  - Repeats step 2
  - Selects a category to use for that turn (**must not have been used before**)

Each Turn

- The total score is updated everytime any category score is updated (i.e., after each player's tur
- A player will sometimes have to choose a category that doesn’t match the configuration of the dice. Score will be 0 in selected category.
- 13 rounds in total (one score for each category)

Calculating Score

- Any roll is valid for 1s, 2s, 3s, 4s, 5s, 6s, and chance
- Not all rolls valid for:3 Of a Kind, 4 Of a Kind, Yahtzee, Full House, Straights (score = 0)
- When checking if roll fits category, think about dice value *frequencies* (e.g. what is 3 of a kind with respect to dice value frequencies?)

```
boolean p = YahtzeeMagicStub.checkCategory
                (dice, FULL_HOUSE);
```

End of Game

- Sum up Upper Bonus, Upper Score, Lower Score, Final Total
- Report winner

Tips

- Use System.out.println() to print testing messages to the Eclipse console (can’t use println()
- Hardcode dice array so you always control what the dice rolls are (great for testing!)
- A user **cannot** re-use any previous category. Print a message if you cannot honor their choice and have them select another
- Mark all methods as **private** unless you explicitly plan for them to be used outside the module
- Determining the validity for *Three of a Kind*, *Four of a Kind*, *Yahtzee*, and *Full House is similar*

ARRAYS

- Dice (`N_DICE`)
- Players (array of player names given to you in starter code as instance variable)
- Scorecard for all players (2D array representing scorecard)

ARRAYS

- Be deliberate and creative about when arrays might be useful data types to store other information
- Think about how to index into Arrays i.e. what the element at index `i` actually means

Libraries

- Read the **`YahtzeeDisplay`** very carefully and think about *when* to use *which method*
- You should not use **`YahtzeeMagicStub`** in your final submission! Implement your own method to calculate whether a set of dice satisfies a particular category

Strategy Advice

- Test your own method in stages
- Sketch out decomposition tree
- Use intermediate milestones e.g. implementing single-player game first is easier
- Think about how to incorporate **`YahtzeeDisplay`** and **`YahtzeeMagicStub`** routines