

# Example: Computing prime numbers

-Write a program that lists all of the prime numbers from 1 to 10,000.

Remember a prime number is a # that is divisible only by 1 and itself

Suggestion: It probably will be useful to write a method `isPrime(int n)` which takes as input a number `n` and outputs whether it is prime or not.

Hint:  $x \% y == 0$  iff `x` is a multiple of `y`

# Last Week

- If statement
- While Loop
- For Loop
- Break/Continue

# This Week

- Arrays

- Reference Types

- Two Dimensional  
Arrays

# Example: Storing a spreadsheet

Suppose we wanted to write a Java program to store a spreadsheet.

Let's say we are going to read all of this from the keyboard.

We could make lots of variables and a scheme where we wrote

*sheet\_row\_column*

i.e. `int sheet_1_b` could store the first row and 2<sup>nd</sup> column.

# Example: Storing a spreadsheet

```
int sheet_1_a, sheet_1_b, sheet_1_m, sheet_1_c;  
int sheet_2_a, sheet_2_b, sheet_1_m, sheet_2_c;  
int sheet_3_a, sheet_3_b, sheet_1_m, sheet_3_c;  
int sheet_4_a, sheet_4_b, sheet_1_m, sheet_4_c;  
int sheet_5_a, sheet_5_b, sheet_1_m, sheet_5_c;  
int sheet_6_a, sheet_6_b, sheet_1_m, sheet_6_c;  
int sheet_7_a, sheet_7_b, sheet_1_m, sheet_7_c;  
int sheet_8_a, sheet_8_b, sheet_1_m, sheet_8_c;  
int sheet_9_a, sheet_9_b, sheet_1_m, sheet_9_c;
```

# Example: Storing a spreadsheet

```
Scanner s = new Scanner(System.in);
```

```
sheet_1_0 = s.nextInt();
```

```
sheet_1_1 = s.nextInt();
```

```
sheet_1_m = s.nextInt();
```

```
sheet_1_f = s.nextInt();
```

```
sheet_2_0 = s.nextInt();
```

```
sheet_2_1 = s.nextInt();
```

```
sheet_2_2 = s.nextInt();
```

```
sheet_2_3 = s.nextInt();
```

Once we go through all this trouble to enter the grades, we still have to work with the numbers!

One idea would be if we could do some sort of for loop.

For example:

```
for (int i=0; i<300; i++) {  
  
    System.out.println("The first column of row " + i + " is " +  
        sheet_i_a  
}
```

However, Java does not allow us to write variables inside our variable names. We can, however, do something pretty similar.

# **COMP-202**

## **Unit 6: Arrays**

### **CONTENTS:**

Array Usage

Multi-Dimensional Arrays

Reference Types

A variable of type *int*.

*int* number = 5

In memory:

5

A variable of type *int*.

*int* number = 5

In memory: 

5
---

An **integer array** corresponds to variable of type *int*[].

`int[] weights = {5, 6, 0, 4, 0, 1, 2, 12, 82, 1}`

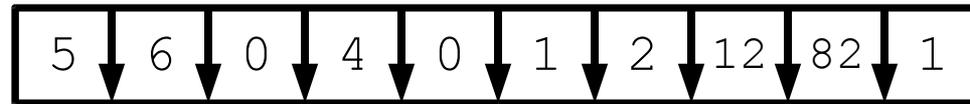
In memory: 

5	↓	6	↓	0	↓	4	↓	0	↓	1	↓	2	↓	12	↓	82	↓	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	----	---	----	---	---

An *array* is a fixed-size, ordered collection of elements of the same type.

```
int[] weights = {5,6,0,4,0,1,2,12,82,1}
```

In memory:

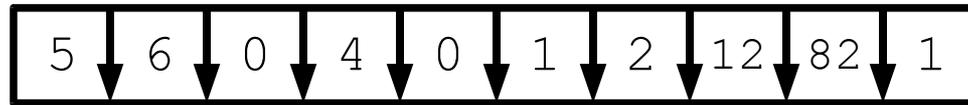


Why use arrays?

They make large amounts of data easier to handle.

```
int[] weights = {5, 6, 0, 4, 0, 1, 2, 12, 82, 1}
```

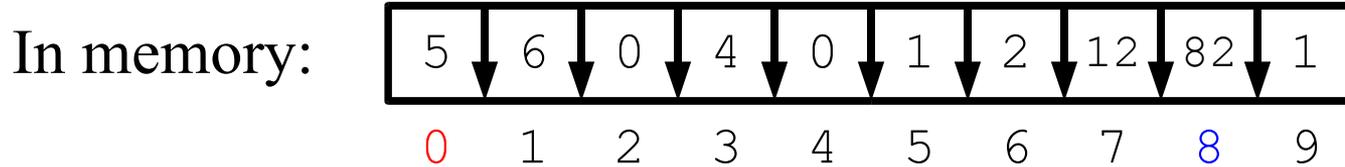
In memory:



Each cell in the array has an **index**.

e.g. The cell that contains 82 has index 8.

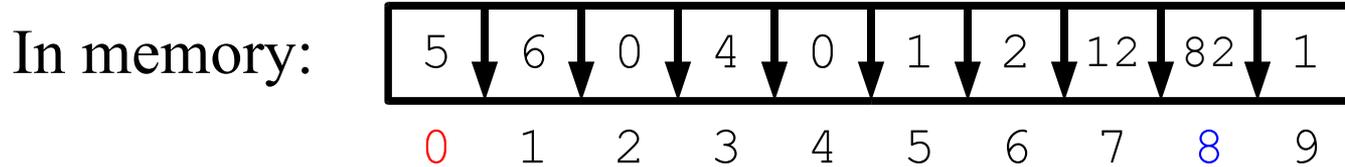
The cell that contains 5 has index 0.



Each cell in the array has an **index**.

e.g. The cell that contains 82 has index 8.

The cell that contains 5 has index 0.



So you can write:

```
int j = weights[8] + weights[0];
```

# **Part 1: Array Basics**

# Array Declaration Examples

- `double[] prices;`

- Declares an array called `prices`

- Each element in this array is a `double`; variable `prices` is of type `double[]`

- `char[] code;`

- Declares an array called `code`

- Each element of this array is a `char`; variable `code` is of type `char[]`

- `String[] names;`

- Declares an array called `names`

- Each element of this array is a `String`; variable `names` is of type `String[]`

# Assigning values to an array

If you know ahead of time how many numbers you want to store (and their values) you can assign values to an array when you declare it:

```
int[] someNumbers = {1,2,3,4,5};
```

# Assigning values to an array

If you do not know ahead of time how many numbers you want to store (or don't know their values), you have to assign the values in 2 phases:

- 1) Tell the computer how many values you want to store
- 2) Set these values

# Setting the size of an array

To specify how large an array should be, you do the following

```
sometype[] myArray;  
//declare an array of type sometype  
...  
  
myArray = new sometype[size];
```

# Accessing elements of an array

To get or set values in an array, you will always use both the array name, and the index of the value you want.

You can think of the index like a subscript.

# Accessing elements of an array

Array indices start from 0

```
//set x to be first value in  
//array  
int x = myArray[0];  
//set 3rd value in myArray  
myArray[2] = 10;
```

# What does this display?

```
public class FirstArray {
    public static void main(String[]
args){
        String[] names = {"Jordan",
"Jesse", "Joshua"};
        for(int i = 1; i >= -1; i = i - 1)
            System.out.println(names[i+1]);
        }
}
```

# What does this display?

```
public class FirstArray {
    public static void main(String[]
args){
        String[] names = {"Jordan",
"Jesse", "Joshua"};
        for(int i = 1; i >= -1; i = i - 1)
            System.out.println(names[i]);
        }
}
```

# Review: how primitive types are stored in memory

Example:

```
int number = 5
```

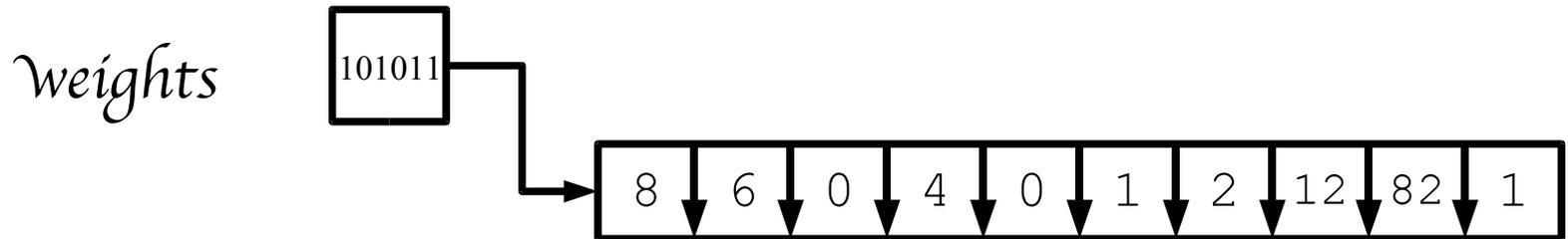
In memory:      `number` 

`number` represents a location in memory where the integer 5 is stored

# Array types are NOT primitive types

Example:

```
int[] weights = {8, 6, 0, 4, 0, 1, 2, 12, 82, 1}
```



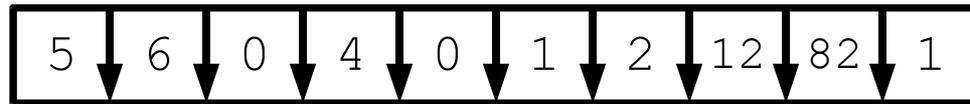
*weights* represents a location in memory where the **address** of the first array cell is stored.

# Primitive vs. reference types

- Primitive types:
  - The variable represents the location in memory at which an **actual value**, like the integer 175.
- Reference types:
  - The variable represents the location in memory at which **another memory address** (or “**reference**”) is stored.

```
int[] weights = {5,6,0,4,0,1,2,12,82,1}
```

In memory:



# Initializer Lists

```
int[] numbers = {2, 3, 5};
```

- The above statement does all the following in one step:
  - It declares a variable of type `int[]` called `numbers`
  - It creates an array which contains 3 elements
  - It stores the address in memory of the new array in variable `numbers`
  - It sets the value of first element of the array to 2, the value of the second element of the array to 3, and the value of the last element of the array to be 5
- Often, these steps are carried out separately.

# Array types are reference types

- The declaration

```
int[] numberArray;
```

creates a reference variable, which holds a *reference* to an `int[]`

- **No array is created yet, just a reference to an array.**

# Reference vs Primitive in methods

-Remember that when you call a method that takes input arguments, you are passing the *value* of the expressions you pass to the method. For example if I have the header:

```
public static void foo(int a, int b)
```

and I call it with :

```
foo(3+4, x+y),
```

I am passing the value 7 and whatever  $x+y$  is to foo, and these values are assigned in a and b

# Reference vs Primitive in methods

-When you call a method, whether it is a primitive or a reference type, remember that you are only passing to the method the *value* of the variable.

# Reference vs Primitive in methods

```
public static void badSwap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

If I call this method `badSwap(x,y)`, it will not swap the values in `x` and `y`. The method `badSwap` only knows the *values* of `x` and `y`

# Reference types

- When you call a method with input of *reference* types, we still pass the value of the variable, but the value now represents an *address* in memory.
- If I swap the *address* inside the method, nothing will change outside.
- But if I swap the *contents* at the address, it will be “permanent”

# Arrays as reference types

For example, if I want to swap two arrays in a method, I have to swap the *contents* of the arrays.

The array addresses will still be the same, but each array would now store what used to be in the other.

```
{
```

```
....
```

```
int[] array1 = {1,2,3,4,5};
```

```
int[] array2 = {6,7,8,9,10};
```

```
badSwap(array1,array2)
```

```
....
```

```
}
```

```
public static void badSwap(  
    int[] a1, int[] a2) {  
    int[] temp = a1;  
    a1 = a2;  
    a2 = temp;  
}
```

This swaps a1 and a2 indeed, but the change will not matter in the calling function

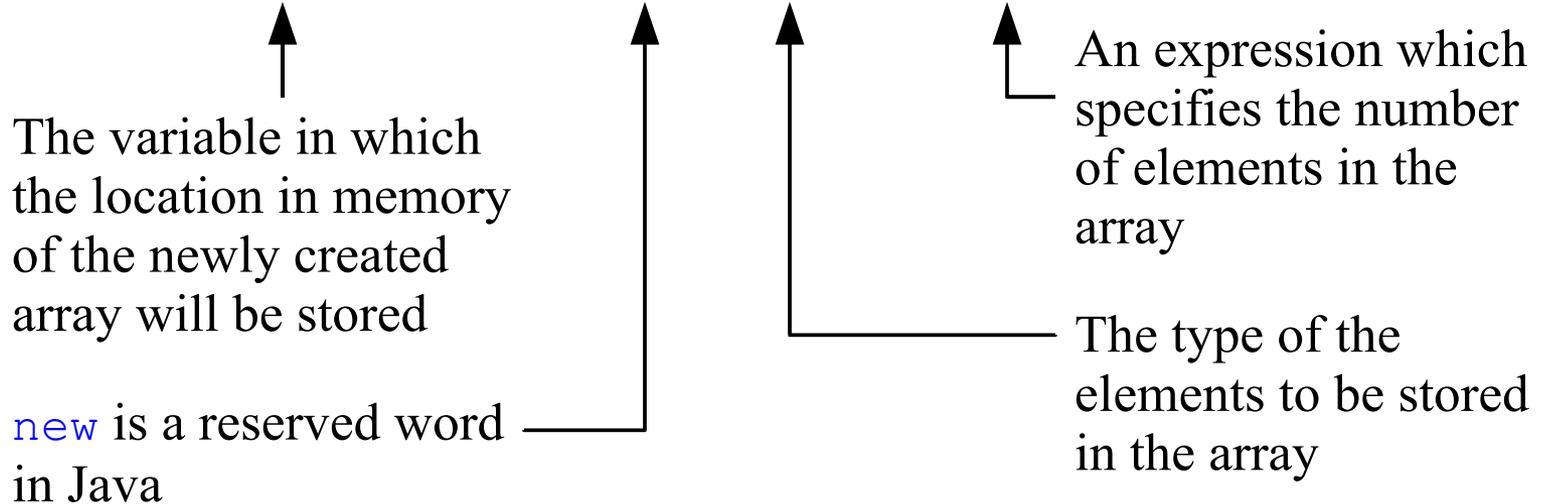
Aside: You can figure out how many elements are in an array by writing

*arrayname.length*

```
public static void goodSwap(int[] array1,  
    int[] array2) {  
    int temp;  
  
    for (int i=0; i < array1.length;i++) {  
        temp = array1[i];  
        array1[i] = array2[i];  
        array2[i] = temp;  
    }  
}
```

# Allocating Arrays

```
variableName = new type [ size ] ;
```



# Allocating Arrays

- Once an array has been created, its size cannot be changed
- As with regular variables, the array declaration and the array allocation operation can be combined:

*type*[] *variableName* = *new* *type*[*size*];

# Initializer Lists

```
int[] numbers = {2, 3, 5};
```

is equivalent to the following code fragment:

```
int[] numbers = new int[3];  
numbers[0] = 2;  
numbers[1] = 3;  
numbers[2] = 5;
```

# Exercise on reference types: what does this display?

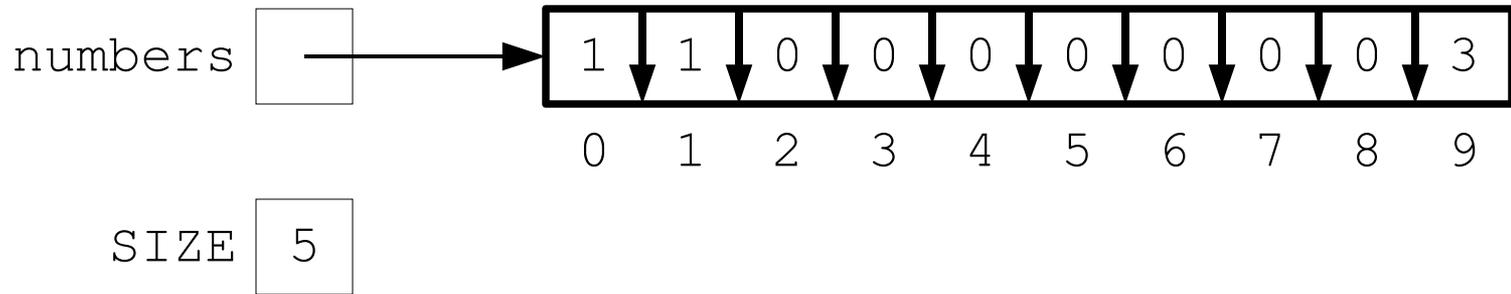
```
public class ArrayCopy {
    public static void main(String[] args){
        int[] numbers = {1, 2, 3};
        int[] differentNumbers = new int[3];
        differentNumbers = numbers;
        numbers[1] = 2;
        differentNumbers[1] = 3;
        System.out.println(numbers[1]);
        System.out.println(differentNumbers[1]);
    }
}
```

# Array Access Example

```
numbers[0] = 1;
```

```
numbers[1] = numbers[0];
```

```
numbers[2 * SIZE - 1] = 2 * numbers[0] + 1;
```



# Array Length

```
int[] weights = {5,6,0,4,0,1,2,12,82,1}  
total = weights.length;
```

We can get the length of any array  
with `[arrayName].length`

Here, `total` is assigned the value 10.

# Array Length

```
int[] weights = {5,6,0,4,0,1,2,12,82,7}  
total = weights.length;
```

What is the index of the cell containing **7**  
in terms of `weights.length`?

# Array Length

```
int[] weights = {5,6,0,4,0,1,2,12,82,7}  
total = weights.length;
```

What is the index of the cell containing **7**  
in terms of `weights.length`?

Answer: `weights.length - 1`

# The length of an array is a constant

```
int[] weights = {5,6,0,4,0,1,2,12,82,1}  
weights.length = 2; // illegal!
```

The `length` field of an array can be used like any other `final` variable of type `int`.

Fill in the method `init` so that it initializes the array `list` with a decreasing list of integers starting at `start`.

**e.g.** If `myArray` has length 5, a call to `init(myArray, 20)` will assign the following values to an array: `{20, 19, 18, 17, 16}`.

```
public static void init(int[] list, int start){  
    // use a loop to assign a value to each cell  
  
}
```

# Bounds Checking (1)

```
int[] myArray = new int[5];  
myArray[5] = 42;  
    // Array myArray contains 5 elements, so  
    // the valid indices for myArray are 0-4  
    // inclusive  
    // Therefore, the program crashes
```

- When this occurs, the error message will mention that the program threw an *ArrayIndexOutOfBoundsException*
- The error message should also mention which line in your program caused the latter to crash



# Reading Exception Output (1)

- The program's output:

- Attempting to retrieve the element at position 5 of an array of
- size 5
- **Exception in thread "main" java.lang.**
- **ArrayIndexOutOfBoundsException: 5**
- **at IndexOutOfBoundsDemo.main(IndexOutOfBoundsDemo.java:11)**
- 

Method where the problem occurred

File where the problem occurred

Line number where the problem occurred

- Nature of the problem and additional information
- Index we tried to access and caused the crash

# Off-By-One Errors Revisited

- Off-by-one errors are common when using arrays:

```
int[] array = new int[100];
int i;

i = 0;
while (i <= 100) {
    array[i] = 2 * i;
    i = i + 1;
}
```

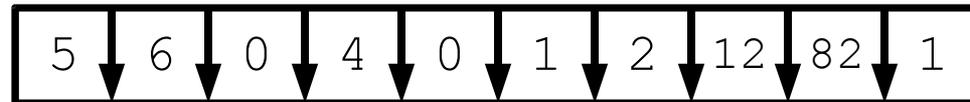
# Last Class

- Arrays
- Declaring an array
- Setting an array using an initializer list
- Indices of an array

An *array* is a fixed-size, ordered collection of elements of the same type.

```
int[] weights = {5,6,0,4,0,1,2,12,82,1}
```

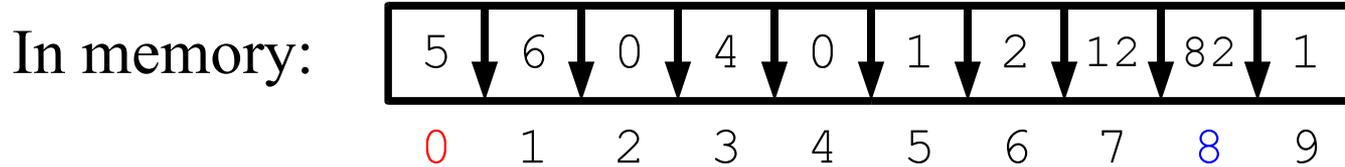
In memory:



Each cell in the array has an **index**.

e.g. The cell that contains 82 has index 8.

The cell that contains 5 has index 0.



So you can write:

```
int j = weights[8] + weights[0];
```

# Declaring an Array Variable

To declare an array variable, you simply write the type you want an array of, followed by [] followed by the name of the variable.

```
int[] myArray;
```

declares an array variable.

# Creating an Array

The previous slide just creates a variable to store an array later on. To actually create an array, you can do one of two things:

1) At the same time you declare the array variable, give the values you want to put into the array:

```
boolean[] rangersWinsByGame = { true, true,  
true, true, true, true, true };
```

# Creating an Array

2) Use the new operator:

```
variablename = new variabletype[size];
```

```
canadiensWins = new boolean[82];
```

Note: boolean array values are automatically initialized to false

# Setting or Getting Specific Values

Since an array is ordered, we can assign each element an index, starting from 0.

The first element is given index 0

The second element is given index 1

The last element is given index  $\text{size} - 1$

Now you can use `arrayname[index]` like any other expression.

# Example: Computing the sum of an array

What if we wanted to write a method that took as input an `int[]` and returned the sum of its values?

# Example: Computing the sum of an array

What if we wanted to write a method that took as input an `int[]` and returned the sum of its values?

```
public static int sum(int[] array) {  
    int totalSoFar = 0;  
    for (int i=0; i < array.length; i++) {  
        totalSoFar += array[i];  
    }  
}
```

# This Class

- Reference variables in more detail
- Two dimensional arrays
- Strings

# **Part 2: Reference Types**

# Primitive Types

- We know that there are two broad kinds of types in Java: primitive types and reference types
  - We have already covered primitive types
  - 
  - There are exactly 8 primitive types.
  - With a primitive type, a variable stores a specific value in it.

# Reference Types

- Other than float, int, double, boolean, char, long, short, and byte every single other type is a reference type.

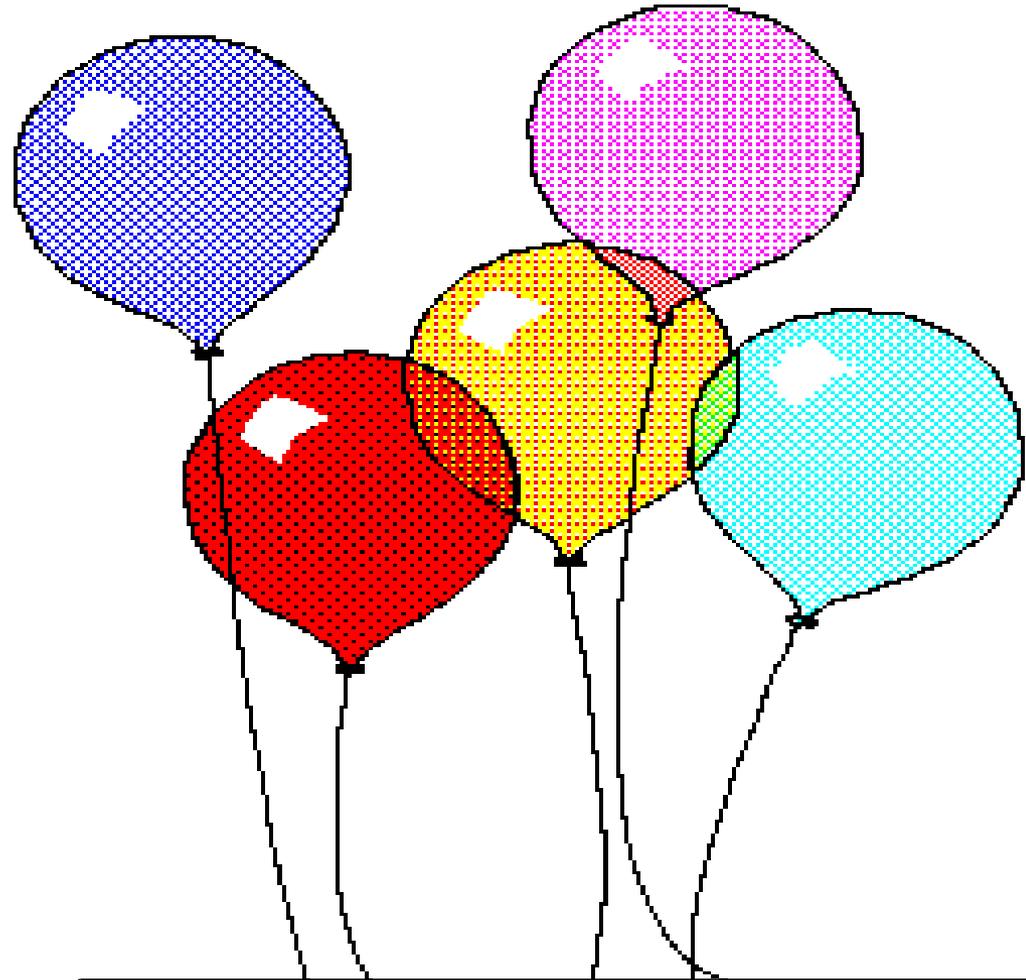
This means that the variable stores an *address* as its value. That address refers to a specific place in memory where some data is stored.

# Reference Types

It would be fair to say that there are 9 things that can be stored in a Java variable:

- 1) Each of the 8 primitive types
- 2) A memory address

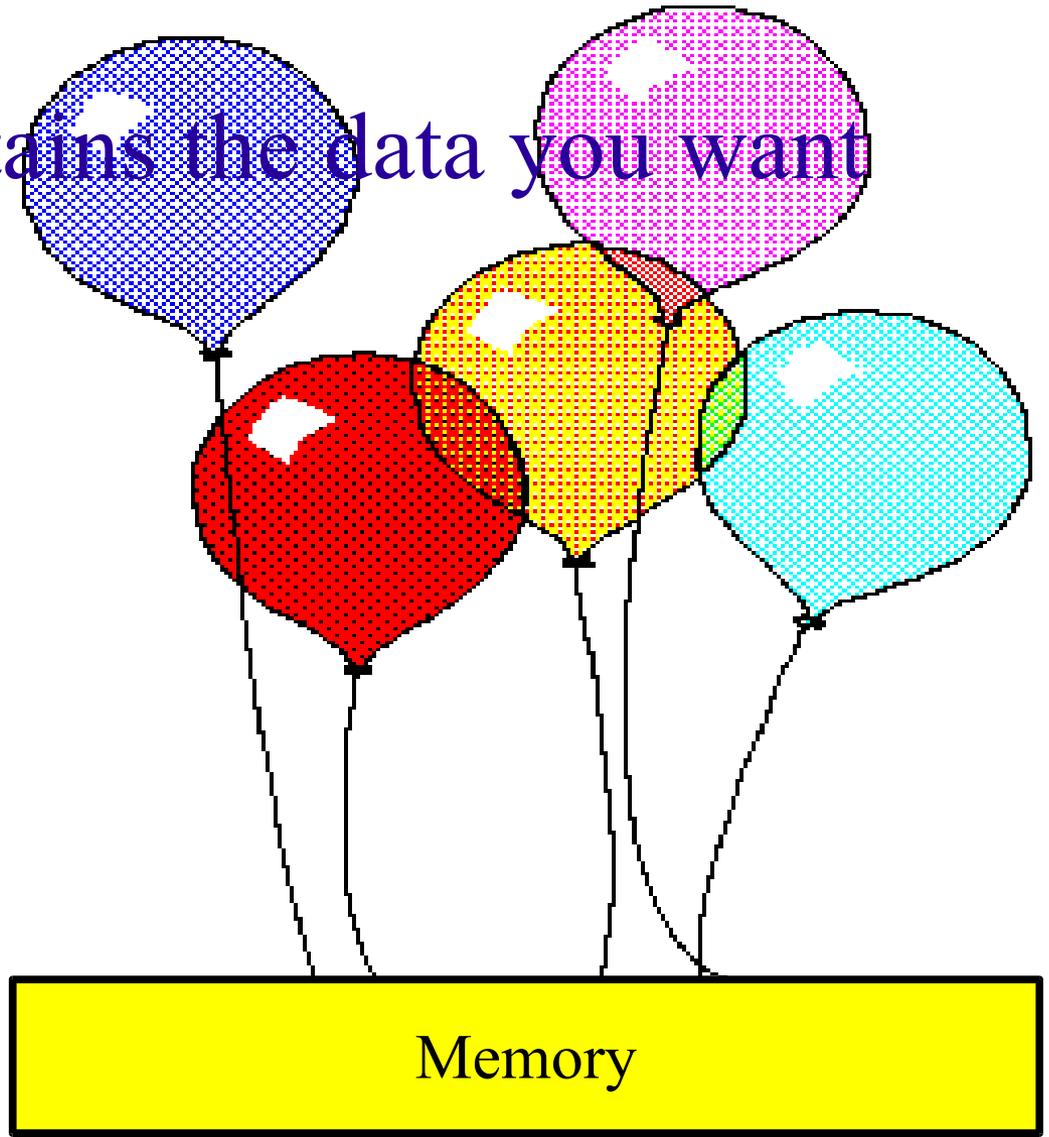
# Reference Types



Memory

# Reference Types

The balloon contains the data you want to manipulate



# Reference Types

The balloon itself can change (for example, by growing or shrinking) but the connection to memory will not



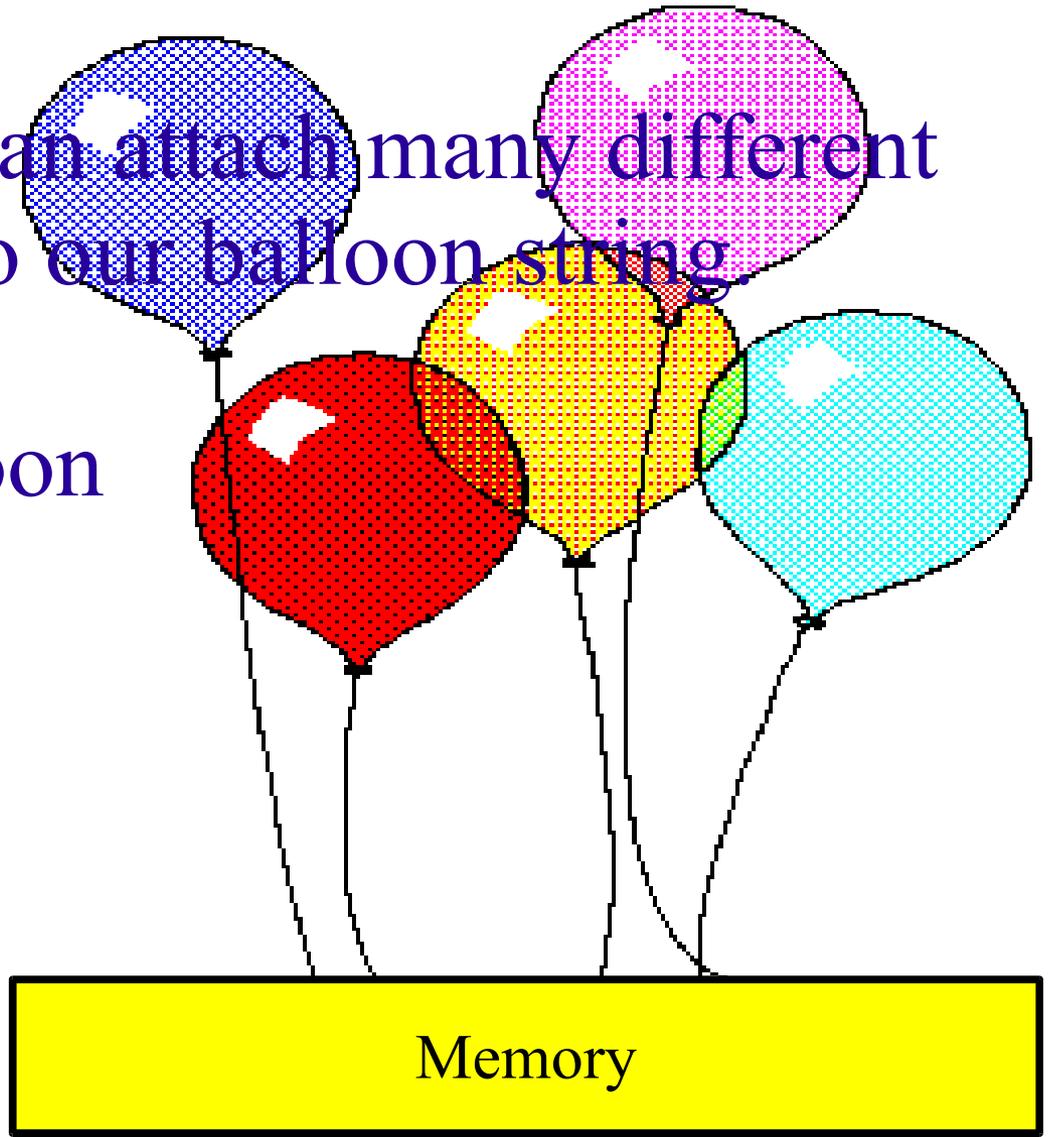
# Reference Types

In addition, we can attach many different kinds of things to our balloon string

Ex: Orange balloon

Blue balloon

Red balloon



# Reference Types

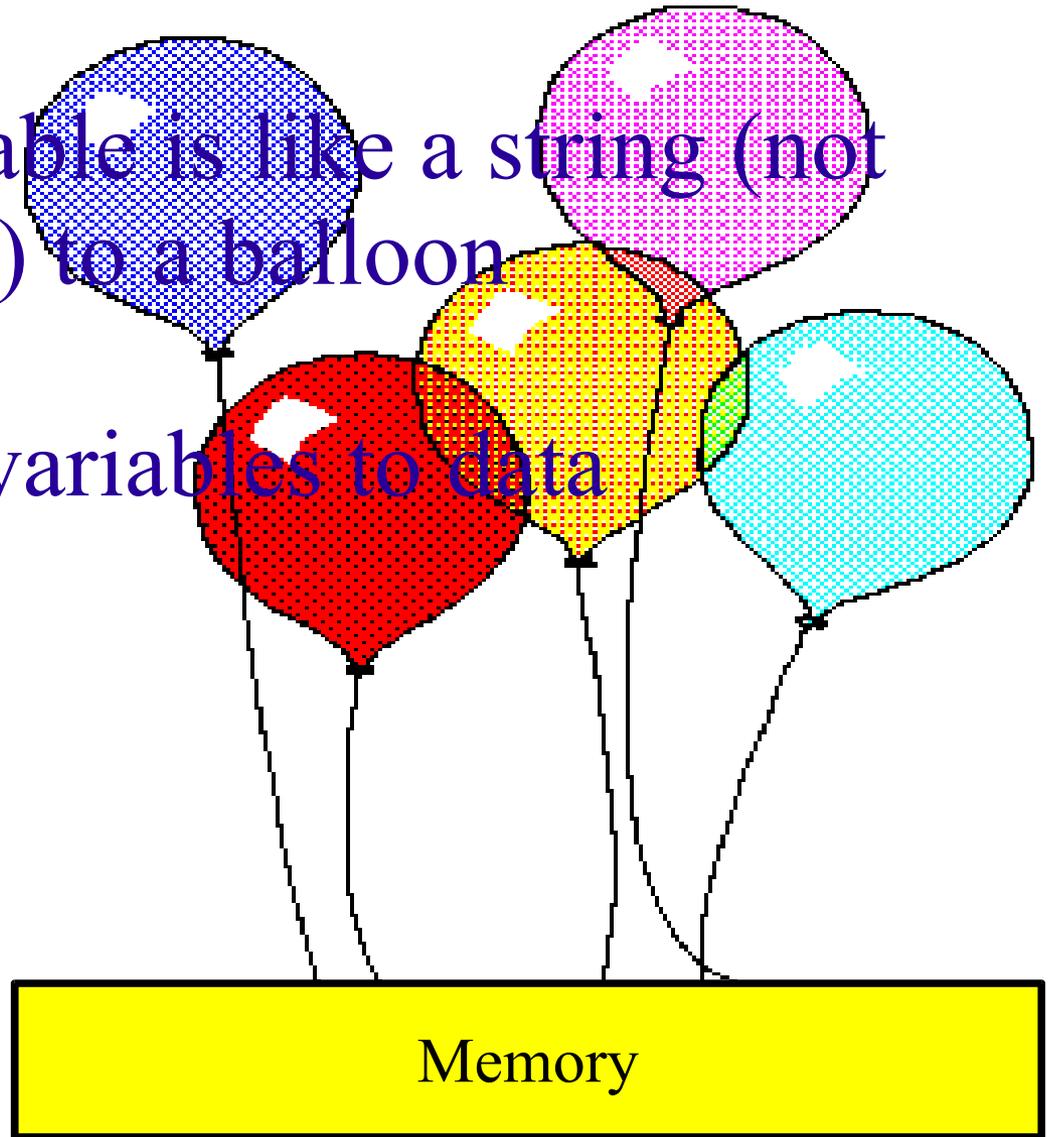
With primitive types, the data itself is stored in memory without a “string”

With reference types, memory just stores the string.

# Reference Types

A reference variable is like a string (not in the Java sense) to a balloon

It is a link from variables to data



# Reference Variables

When I declare a reference variable, by writing:

```
type variablename;
```

I am creating a balloon string, but no balloon.

For example:

```
int[] myArray;
```

creates a balloon string that later could be attached to a balloon that contains an int[] .

# Reference Variables

When you write:

```
new int[10];
```

you are actually creating the balloon-- an array of size 10

When you write

```
myArray = new int[10];
```

you are creating a balloon AND attaching the variable (i.e. balloon string) to that balloon

# Reference Variables

If I just write:

```
new int[10];
```

in a statement all by itself, I'll be creating a balloon, but not attaching it to any string.

What will happen in this case?

# Reference Variables

Much like a balloon, it will just drift away.

What happens if I write:

```
int[] myArray;
```

```
myArray = new int[10];
```

```
myArray = new int[100];
```

# Reference Variables

```
int[] myArray;
```

Creates a variable in Java to store the address of an int[] (i.e. it creates a balloon string)

# Reference Variables

Much like a balloon, it will just drift away.

What happens if I write:

```
myArray = new int[10];
```

Creates an int array of size 10 and sets the address stored in the variable myArray to be the location of this array. (i.e. it creates a balloon and attaches it to the balloon string)

# Reference Variables

Much like a balloon, it will just drift away.

What happens if I write:

```
myArray = new int[100];
```

Creates an int array of size 100 and sets the address stored in the variable myArray to be the location of this array. (i.e. it creates a 2nd balloon and attaches it to the balloon string. In doing so, it has to release the 1<sup>st</sup> balloon, causing the first balloon to drift away.)

# Using Reference Variables

For the most part, Java will conceal from you many of the details about reference variables. There are 2 things to be clear about:

1) Whenever you write

variablename = \_\_\_\_\_

you are changing the value stored in the variable (i.e. the balloon string)

Note: it has to be EXACTLY variablename = (not variablename[index] = for example)

# Using Reference Variables

For the most part, Java will conceal from you many of the details about reference variables. There are 2 things to be clear about:

2) Whenever you write anything else involving the variable, you are going to the address in memory and doing something to it (either reading some value(s) or writing some value(s))

In the metaphor, you are following the balloon string to the balloon itself and performing an action there.

# Passing Arguments to Methods

Remember, when you call a method with arguments, the arguments passed are evaluated in the calling function and then copied to the variables defined as the formal arguments to the method:

```
Ex: public static void methodTest(int x, int y) {  
    x++; y++;  
}  
  
public static void main(String[] args) {  
    int z = 1;  
    methodTest(z, z+1) ;  
}
```

# Passing Arguments to Methods

```
public static void methodTest(int x, int y) {
```

```
    x++; y++;
```

```
}
```

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

```
    int z = 1;
```

```
    methodTest(z, z+1) ;
```

```
}
```

First the expression “z” is evaluated to be 1 and so the number 1 is copied into the variable x.

Then the expression “z+1” is evaluated to be 2 and so the number 2 is copied into the variable y

# Passing Arguments to Methods

```
public static void methodTest(int x, int y) {  
    x++; y++;  
}  
  
public static void main(String[] args) {  
    int z = 1;  
    methodTest(z, z+1) ;  
}
```

After “y++” executes, the value of x is 2 and y is 3.

However, when the method returns back to the main method, the value of z is still 1. This is since x and y got their values from an expression related to z, but that's the only connection

# Passing Arguments to Methods

This idea will apply with arrays as well.

```
public static void methodTest(int[] x) {  
    x = new int[5];  
}  
  
public static void main(String[] args) {  
    int[] z = {1,2,3,4,5};  
    methodTest(z);  
}
```

First we create an `int[]` with values 1,2,3,4,5. We assign a variable `z` to store the address of the array. We then call the method `methodTest` with argument of `z`. This means we evaluate the expression `z`, which will evaluate to an address, and assign its value to the variable `x`.

In the analogy, this means `x` and `z` are 2 balloon strings on the same balloon  
`x = new int[5]` just creates a new balloon and attaches `x` to it

# Making a Change

```
public static void methodTest(int[] x) {  
    x[0]++;  
}  
  
public static void main(String[] args) {  
    int[] z = {1,2,3,4,5};  
    methodTest(z);  
}
```

Here, we are doing something different. The first few steps are the same. We create an array, attach the variable `z` to it. We then call the method `methodTest()` with 1 argument. This means we evaluate the expression `z` and assign it (an address) to the variable `x`.

However, inside the method we do something different. We increment the value `x[0]`

# Making a Change

```
public static void methodTest(int[] x) {  
    x[0]++;  
}  
  
public static void main(String[] args) {  
    int[] z = {1,2,3,4,5};  
    methodTest(z);  
}
```

Remember that this sort of statement means “go to the address stored in the variable x and increment its 0<sup>th</sup> element”

In the metaphor this would mean “follow the string to the balloon and add 1 to the 0<sup>th</sup> element of the balloon”

# What about this?

```
public static void methodTest(int[] x) {  
    x = new int[10];  
    x[0]++;  
}  
  
public static void main(String[] args) {  
    int[] z = {1,2,3,4,5};  
    methodTest(z);  
}
```

What are the values stored in the array linked to the variable z after the method methodTest() is complete?

# What about this?

```
public static void methodTest(int[] x) {  
    x = new int[10];  
    x[0]++;  
}  
  
public static void main(String[] args) {  
    int[] z = {1,2,3,4,5};  
    methodTest(z);  
}
```

What are the values stored in the array linked to the variable z after the method methodTest() is complete?

{1,2,3,4,5}

# Aliases (1)

- When the address in memory of one object is stored in two or more different variables, we say that the variables are *aliases*
- We need to be careful when working with aliases
  - Aliasing has side-effects that we need to be aware of
- Because the variables contain the address in memory of the same object, the result of changing the *object* using one of its aliases will be reflected through the other aliases
  - Recall that the object is not stored in the reference variables; instead, the reference variables specify where to find the object in memory
  - Thus, changing the object is not the same as changing the contents of the reference variable which contain the address of this object

# Aliases (2)

- To "de-alias" the variables, we simply need to store the address of a different object in one of them by using an assignment statement
- Note that aliasing is only a property of reference types; it is not a property of primitive types
- 
- In the balloon metaphor an alias means you have two balloon strings that each are attached to the same balloon.

# Alias Example

- Consider the following code fragment:

```
int[] a1 = {2, 3, 5};  
int[] a2 = a1;  
a1[0] = 7;  
int v = a2[0];
```

- What will be the value of variable `v` after the last line in the above fragment is executed?
- The answer is: 7 (?!?)
  - Because `a1` and `a2` are aliases for the same object, any change to the object via one of its aliases will be reflected through the other alias

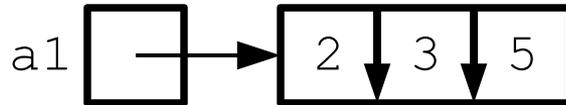
# Comparing Objects (1)

- Among the comparison operators, only `==` and `!=` are defined for reference types
  - If you try to compare two reference variables using `<`, `<=`, `>`, or `>=`, the compiler will report an error
- Moreover, when used to compare reference variables, the `==` and `!=` operators check whether the two reference variables refer to the same object in memory
  - In other words, for reference variables, the `==` and `!=` operators check whether the *addresses* stored in them are the same
  - In yet other words, the `==` and `!=` operators check whether two reference variables are aliases for the same object
  - But two different objects, stored at different addresses, can have the same contents...

# Comparing Objects Using == (1)

- `int[] a1 = {2, 3, 5};` // Line 1
- `int[] a2 = a1;` //  
Line 2

- 
- `boolean b = (a1 == a2);` // Line 3



Line 1:            0   1   2

---



Line 2:            0   1   2

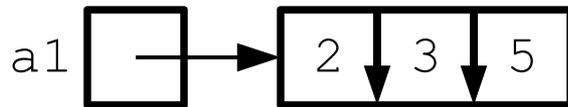
---



Line 3:            0   1   2

# Comparing Objects Using == (2)

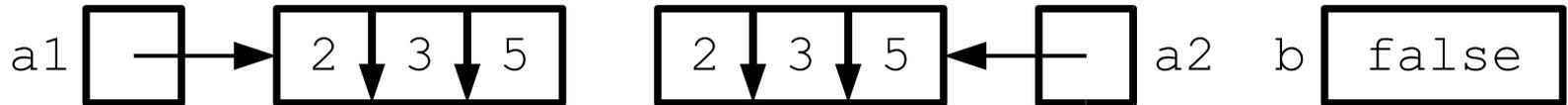
- `int[] a1 = {2, 3, 5};` // Line 1
- `int[] a2 = {2, 3, 5};` // Line 2
- `boolean b = (a1 == a2);` // Line 3



Line 1: 0 1 2



Line 2: 0 1 2 0 1 2



Line 3: 0 1 2 0 1 2

# More on Comparing Objects

- To see check whether two objects have the same contents, we need to compare their contents manually
  - For arrays, this can be done using a loop to verify whether all elements are the same
  - For other kinds of objects (like `Strings`), we can use the `equals()` method
- In fact, the reason `Strings` should not be compared using the `==` and `!=` operator is because `String` is a reference type
  - Therefore, when applied to `String` variables, the `==` and `!=` operators check whether the variables contain the address in memory of the same `String` object
  - But two `String` variables could contain the addresses of two different `String` objects which consist of the same characters in the same order

# More on Comparing Objects (2)

–In such a case, the expression involving the `==` operator would evaluate to `false`, even though the `Strings` really are equal

–Again, ***never** use the `==` and `!=` operators to compare `Strings` for equality (or inequality) unless you are **really** sure of what you are doing; always use the `equals()` method to compare `Strings`*

# The `null` Literal Value

- Sometimes, we want a reference variable to contain a special value to indicate that the variable intentionally does not contain the address in memory of a valid object
- The literal value `null` can be used for this purpose
  - Like the `boolean` literals `true` and `false`, `null` is not technically a reserved word, but you cannot use it for any purpose other than as a literal value for reference variables
  - `null` can be assigned to reference variables of any type
- A reference variable containing the value `null` is sometimes said to "point nowhere"
  - In memory diagrams, `null` is often represented with a ground symbol

# Using `null`

- One can assign `null` to a reference variable like one assigns an `int` literal like `42` to a variable of type `int`

```
int[] a;  
a1 = null;
```

- One can check whether or not a reference variable contains the value `null` like one checks whether or not a variable of type `int` contains any value represented by an `int` literal like `42`

```
if (a == null) {  
    // do something  
} else {  
    // do something else  
}
```

# null-Related Caveats

- The address stored in a reference variable is always either the address in memory of a valid object of that variable's type, or `null`

- In Java, you cannot store an arbitrary memory address in a reference variable, nor manipulate memory addresses directly

- If you attempt to use access an object using a reference variable which contains `null`, your program will crash:

```
int[] a = null;  
a[0] = 1; // a1 contains null: crash!
```

- When this occurs, the error message will mention that the program threw a `NullPointerException`

- The error message should also mention which line in your program caused the latter to crash

# NullPointerExceptionDemo.java

```
• 1 • public class NullPointerExceptionDemo {
• 2 •     public static void main(String[] args) {
• 3 •         int[] a = null;
• 4 •
• 5 •         System.out.println("Attempting to retrieve the " +
• 6 •             "element stored at index 0 of an array through a "
• 7 •         +
• 8 •             "null reference variable...");
• 9 •         System.out.println("The value stored at index 0 of "
• 10 •         +
• 11 •             "this array is: " + a[0]);
• 12 •     }
• 13 • }
```



What will happen if we run this program?

# Reading Exception Output (2)

- The program's output:

- Attempting to retrieve the element stored at index 0 of an array
- through a null reference variable...
- **Exception in thread "main" java.lang.NullPointerException**
- **at NullPointerExceptionDemo.main(NullPointerExceptionDemo.java:8)**
- 

Nature of the problem



# Garbage Collection (1)

- When there are no more reference variables containing the address of an object, the object can no longer be accessed by the program
- It is useless, and therefore called *garbage*
- Java performs *automatic garbage collection* periodically
  - When garbage collection occurs, all the memory allocated to store garbage objects is made available so it be allocated to store new objects
- In other languages, the programmer has the responsibility for performing garbage collection
- Always ensure you do not lose the last reference to an object you still need

# Garbage Collection (2)

- On the other hand, if you no longer need an object, you should make sure that none of your reference variables contain its address in memory
  - You can do this by assigning `null` to the reference variables which contain the address of this object
  - This is so that the memory it occupies can be reclaimed by the garbage collector and reused to store new objects

# Passing Reference Types (1)

- Recall: Parameter passing works just like an assignment statement
  - The value of the actual parameter is copied into the method's formal parameter
- When passing reference types, the value of the actual parameter also is copied into the formal parameter just like in an assignment statement
- However, recall that the value stored in a reference variable is the address in memory where an object is located, not the object itself

# Passing Reference Types (2)

- Therefore, for parameters whose types are reference types, the value copied in the formal parameter is the address where an object is stored in memory
- This implies that:
  - When passing reference types, the formal parameter and the actual parameter become aliases
  - If a method changes the contents of an object whose address is stored in a formal parameter, the change will also be reflected by the actual parameter
  - However, if you change the *address* stored in the formal parameter, the address stored in the actual parameter will not change

# ReferenceAssignDemo.java (1 / 2)

```
• public class ReferenceAssignDemo {  
•     public static void main(String[] args) {  
•         int[] a1 = {1, 2, 3};  
•         int[] a2 = {4, 5, 6};  
•         int[] ac1, ac2; // *** Line 1  
•  
•         System.out.println("Value of a1[0]: " + a1[0]);  
•         System.out.println("Value of a2[0]: " + a2[0]);  
•  
•         ac1 = a1;  
•         ac2 = a2; // *** Line 2  
•  
•         System.out.println("Value of ac1[0]: " + ac1[0]);  
•         System.out.println("Value of ac2[0]: " + ac2[0]);  
•  
•         // Continued on next slide
```

# ReferenceAssignDemo.java (2 / 2)

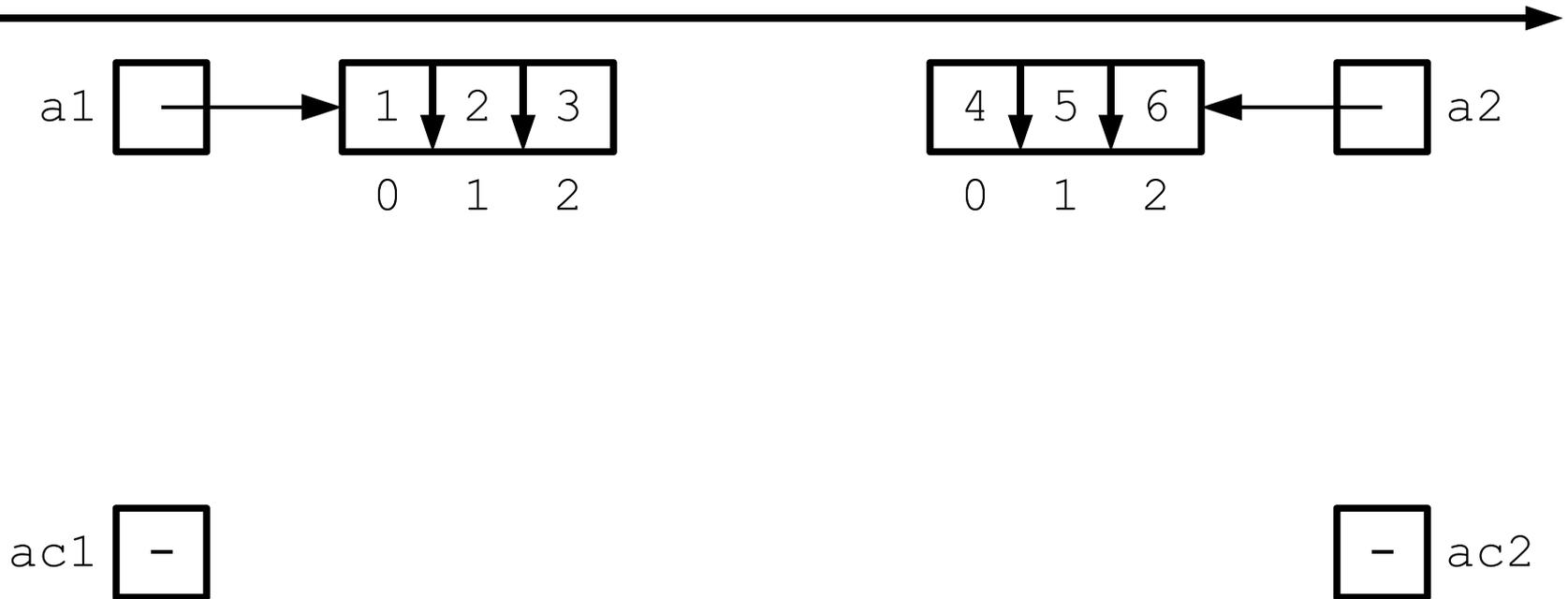
- `// Continued from previous slide`
- `ac1[0] = 7;`
- `ac2 = new int[3];`
- `ac2[0] = 8;`
- `ac2[1] = 9;`
- `ac2[2] = 10; // *** Line 3`
- 
- `System.out.println("New value of ac1[0]: " + ac1[0]);`
- `System.out.println("Value of new ac2[0]: " + ac2[0]);`
- `System.out.println("Value of a1[0] after assignment: " +`
- `a1[0]);`
- `System.out.println("Value of a2[0] after assignment: " +`
- `a2[0]);`
- `}`
- `}`

What does this display?

What does it look like in memory while it is running?

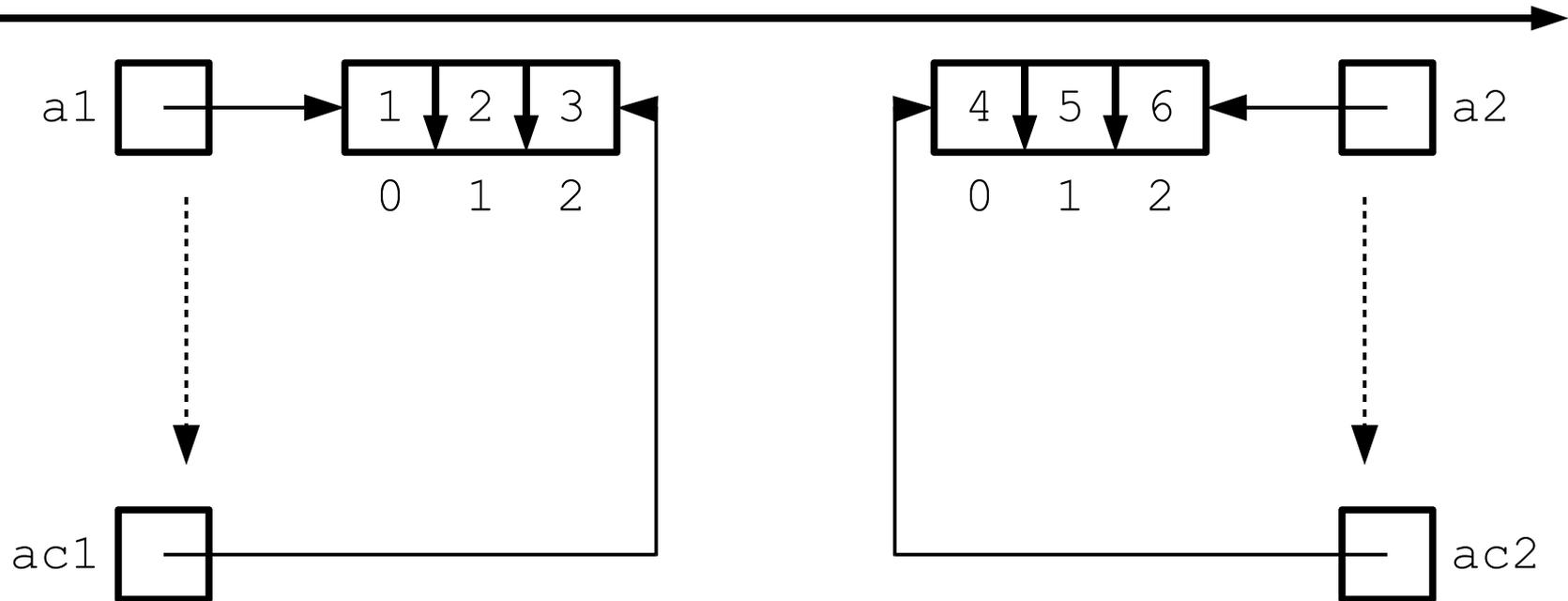
# More Aliases in Memory (1)

- Memory contents after execution of line 1:



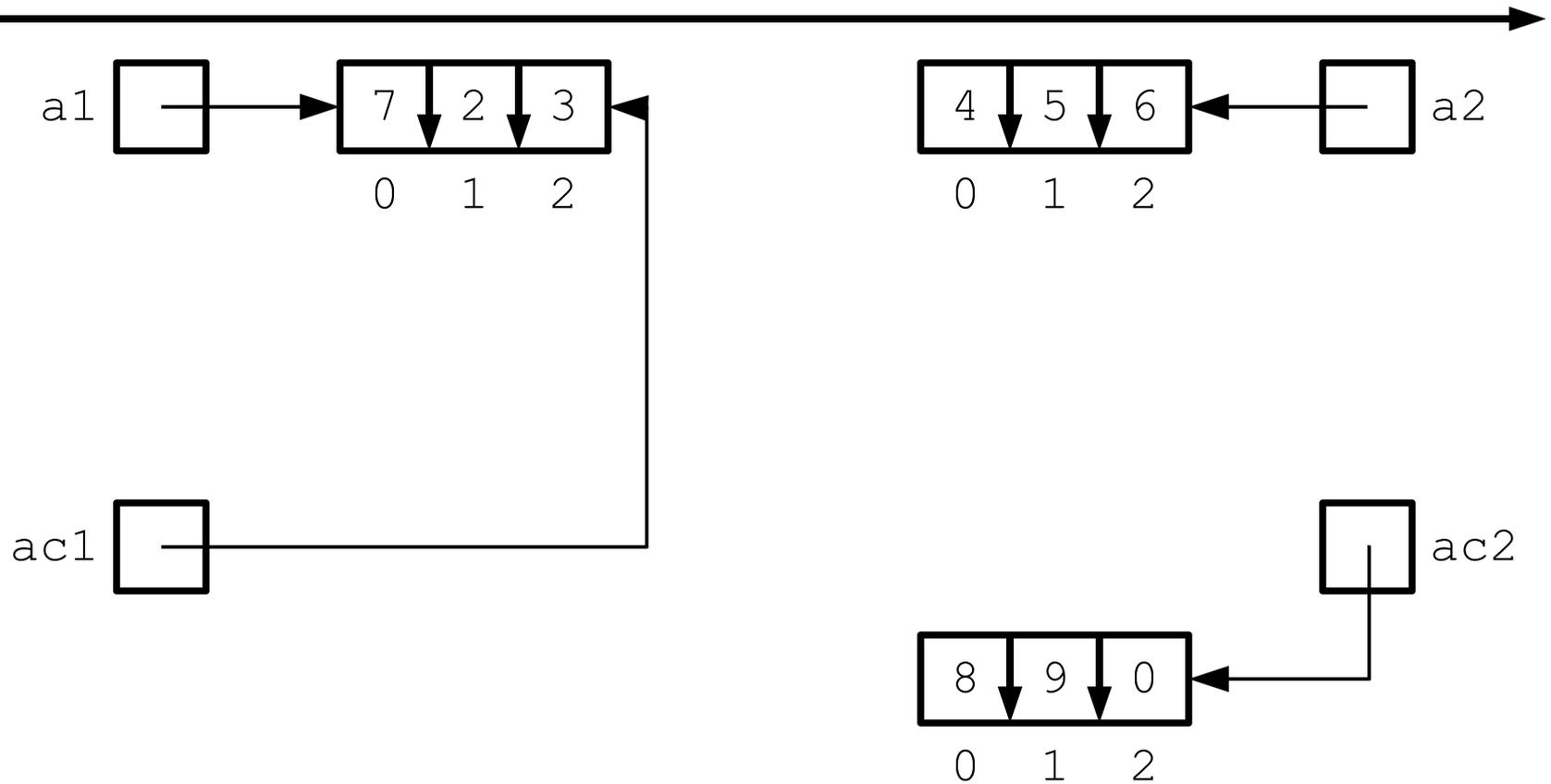
# More Aliases in Memory (2)

- Memory contents after execution of line 2:



# More Aliases in Memory (3)

- Memory contents after execution of line 3:



# ReferencePassingDemo.java (1 / 2)

```
• public class ReferencePassingDemo {  
•     public static void main(String[] args) {  
•         int[] a1 = {1, 2, 3};  
•         int[] a2 = {4, 5, 6};  
•  
•         System.out.println("Value of a1[0] before calling m(): " +  
•             a1[0]);  
•         System.out.println("Value of a2[0] before calling m(): " +  
•             a2[0]);  
•         System.out.println("--");  
•         m(a1, a2);  
•         System.out.println("--");  
•         System.out.println("Value of a1[0] after calling m(): " +  
•             a1[0]);  
•         System.out.println("Value of a2[0] after calling m(): " +  
•             a2[0]);  
•     }  
•  
•     // Continued on next slide
```

# ReferencePassingDemo.java (2 / 2)

- `// Continued from previous slide`
- `public static void m(int[] ac1, int[] ac2) {`
- `System.out.println("Value of received ac1[0]: " + ac1[0]);`
- `System.out.println("Value of received ac2[0]: " + ac2[0]);`
- `// *** Line 2`
- `ac1[0] = 7;`
- `ac2 = new int[3]; // *** Line 3`
- `ac2[0] = 8;`
- `ac1[1] = 9;`
- `ac2[2] = 0;`
- `System.out.println("New value of ac1[0]: " +`
- `ac1[0]);`
- `System.out.println("Value of new ac2[0]: " +`
- `ac2[0]);`
- `}`
- `}`

What does this display?

What does it look like in memory while it is running?

# Reference Passing in Memory (1)

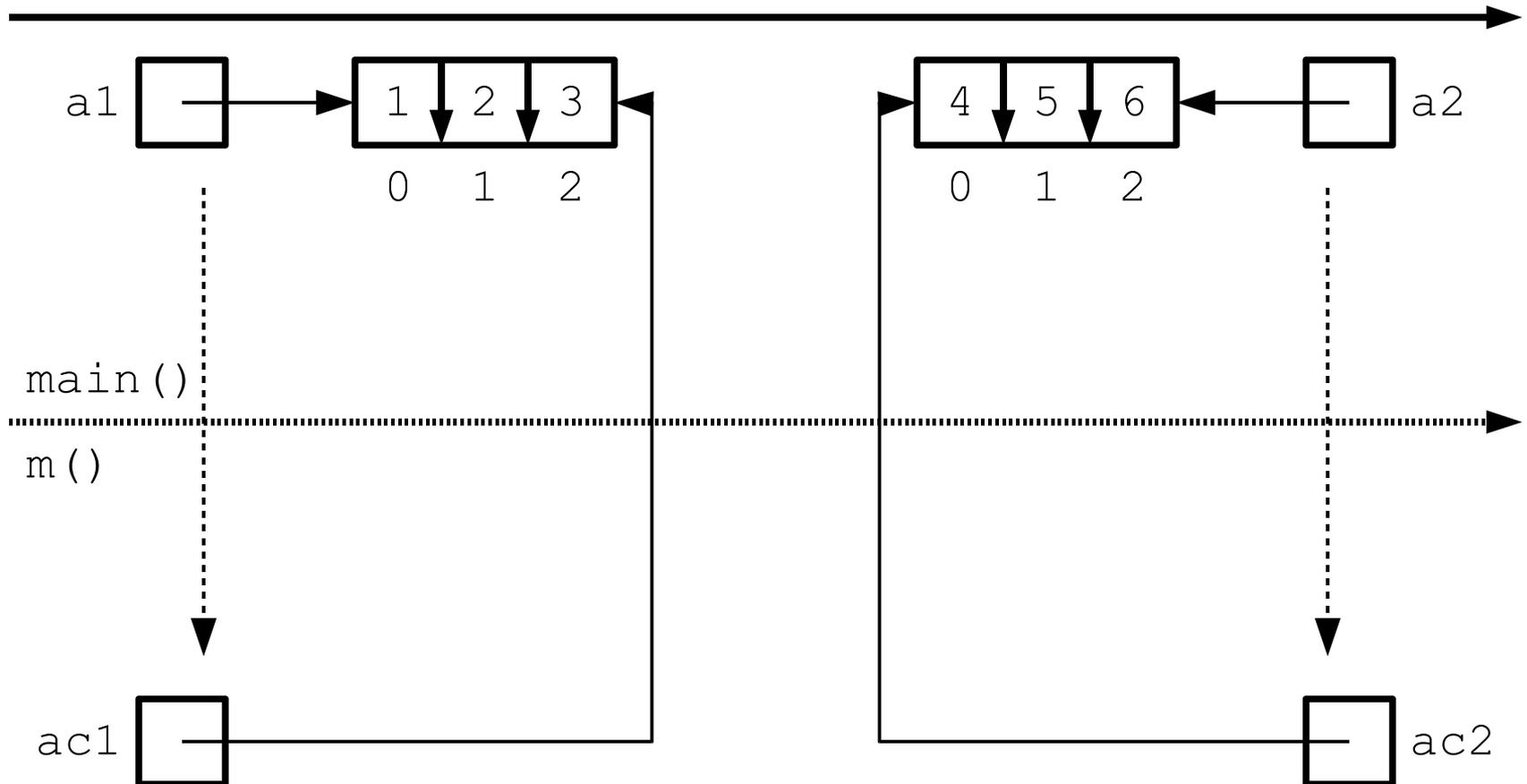
- Memory contents after execution of line 1:



main ()

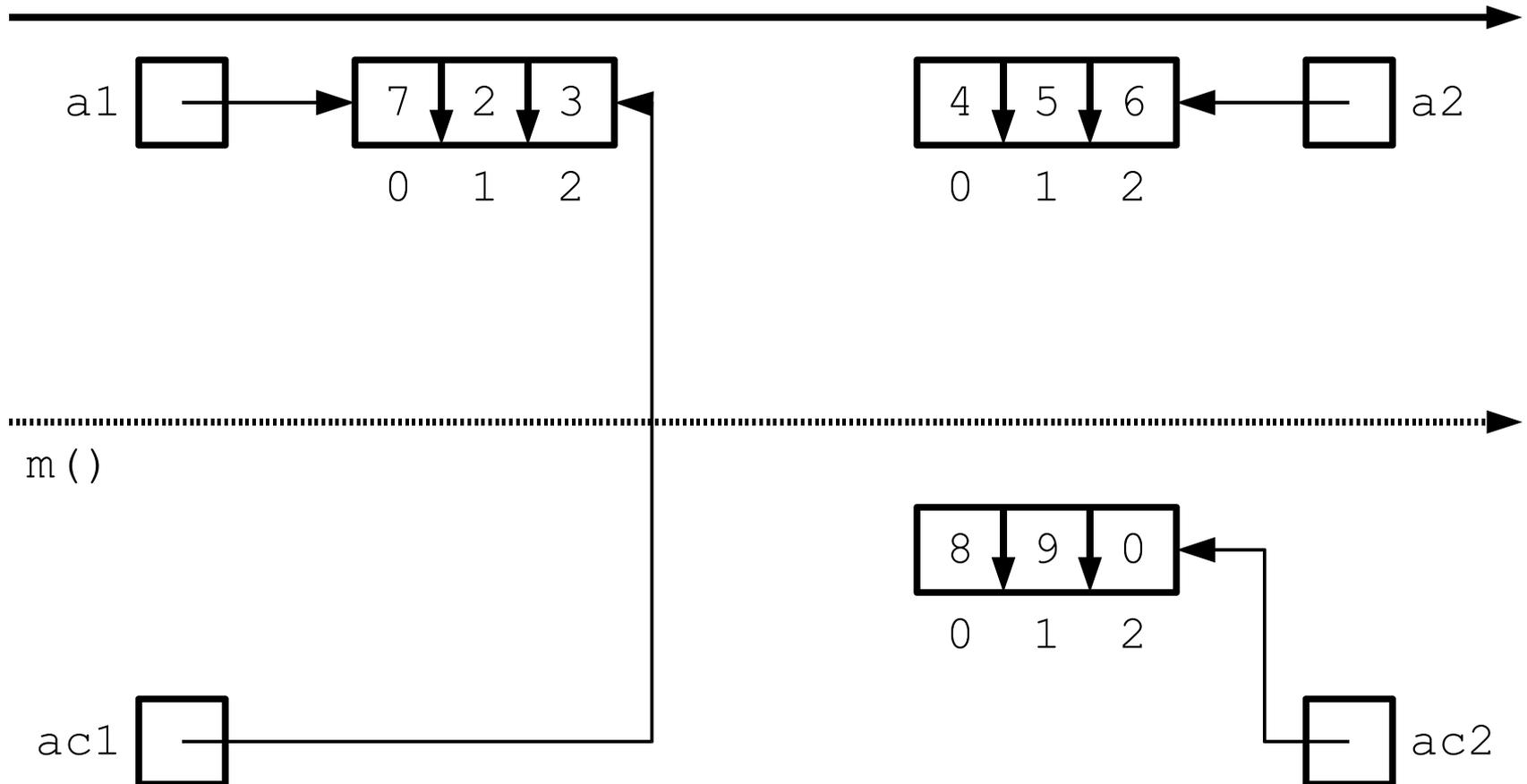
# Reference Passing in Memory (2)

- Memory contents after execution of line 2:



# Reference Passing in Memory (3)

- Memory contents after execution of line 3:



# **Part 3: String (not not the balloon ones this time)**

# String

- A commonly used reference type in Java is a String
- A String can be used to store an ordered list of char
- A String is a type of Java Object. An Object in Java is simply any thing that a reference type can refer to (in other words, anything that is a “balloon” is an Object)
- Normally, to create an Object, you must write “new”

There are many special things about Strings. For starters, you can create a String by putting text between “ “ w/o using “new”

```
String s = “foo”;
```

creates a String.

# String

You also could create a String using the new operator:

```
String s = new String("foo"); /*creates a new String based on the  
literal "foo" */
```

```
String a = new String(s); /*creates a 2nd new String based on the first  
String */
```

Every time you use the new keyword, you are creating a new Object.

In the case of String, when you write new String(s) you are creating a new String based on the old String (it will have the same contents).

# Attributes of Objects

Objects in Java can have certain *attributes*

You can access these attributes using the . operator

For example, an attribute of an array is the length attribute. If I have an array, I can write

```
arrayname.length
```

to access the attribute length of an array.

# Behaviors or Methods of Objects

Objects in Java can also have *methods* or *behaviors* which act upon the Object

You can also access these methods using the . operator

Any String object has a *behavior* defined on it that calculates its length:

```
String s = "foo";  
int length = s.length();
```

Notice that length has () after it because length is a *behavior* not an *attribute*

# Other Behaviors of String

charAt()

length()

substring()

concat()

indexOf()

.....

You are not responsible for knowing most of these other than charAt() and length() but you ARE responsible for knowing how to look up more about them

<http://download.oracle.com/javase/6/docs/api/java/lang/String.html>

# **Part 4: Multidimensional Arrays**

# Two-Dimensional Arrays (1)

- So far, the elements of all the arrays we have seen have been simple values: primitive types or `Strings`
  - Such arrays are *one-dimensional*
- However, we can create arrays whose elements are *themselves* one-dimensional arrays
  - Such an array is really an array of arrays
  - These arrays are *two-dimensional* arrays (or 2D arrays)
- Elements in a two dimensional array are accessed using two indices
  - The first index specifies the one-dimensional array containing the element we want to access

# Two-Dimensional Arrays (2)

- The second index specifies the element we want to access within the one-dimensional array specified by the first index
- It may be useful to think of two-dimensional arrays as tables of values with rows and columns
- The first index specifies a row or column
- The second index specifies an element within a row or column

# 2D Array Declarations (1)

- The declaration for a two-dimensional array has the same basic syntax and semantics as the declaration of a one-dimensional array
- The only difference is that there are two pairs of square brackets ([]) instead of just one
- Examples:

```
double[][] matrix;  
char[][] mysteryWord;
```

- We can also declare two-dimensional arrays by placing the square brackets after the variable name instead of after the type

```
double table[][];
```

# 2D Array Declarations (2)

–Again, placing the brackets before the type is the preferred declaration style

- More array declaration examples:

–`double[][] distances, prices;`

Declares two variables of type `double[][]`; one is called `distances` and the other is called `prices`

–`double distances[][], totalPrice;`

Declares a variable of type `double[][]` called `distances`, along with a single variable of type `double` called `totalPrice`

–`double[] distances[], averages;`

Declares a variable of type `double[][]` called `distances`, along with a variable of type `double[]` called `averages`

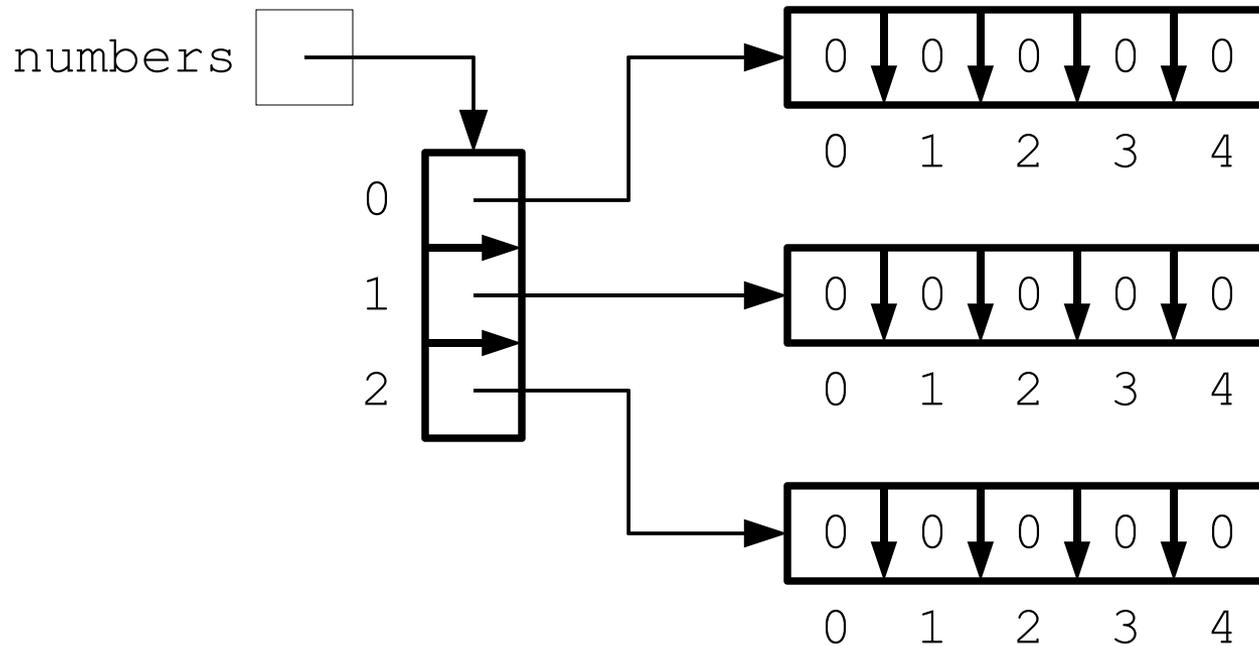
# 2D Array Allocations (1)

- We can allocate a two-dimensional array using the same basic syntax as when we allocate a one-dimensional array
- The only difference is that we can specify two sizes
  - The first size specifies the number of one-dimensional arrays that the array of arrays will contain
  - The second size component specifies the number of elements in each one-dimensional array
- For example, if a variable of type `int[][]` called `numbers` has already been declared, we can allocate a two-dimensional array of `int` and store its address in variable `numbers` like this:

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

# 2D Array Allocations (2)

- This allocates an array that can store 3 elements, each of which is the address of an array of `ints`
- Each of these 3 arrays of `ints` can store 5 elements, each of which is of type `int`



# Accessing 2D Array Elements

- We can access the individual elements of a two-dimensional array by using two indices, between two sets of square brackets
  - The first index specifies the one-dimensional array which contains the element we want to access
  - The second index specifies the element we want to access within the one-dimensional array specified by the first index
- The individual elements of a two-dimensional array can be used in the same manner as regular variables of the same type
- We can also access an entire one-dimensional array by specifying only one index between a pair of square brackets
  - When we do this, the one-dimensional array can be used in the same manner as a regular one-dimensional array

# 2D Array Access Example (1)

- For example, suppose we have the following array declaration / allocation statement:

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

- Consider the expression `numbers[1][3]`
  - The above expression refers to the element at position 3 of a one-dimensional array of `ints`
  - This one dimensional array of `ints` is stored in position 1 of a two-dimensional array of `ints`
  - The address of this two-dimensional array of `ints` is stored in variable `matrix`

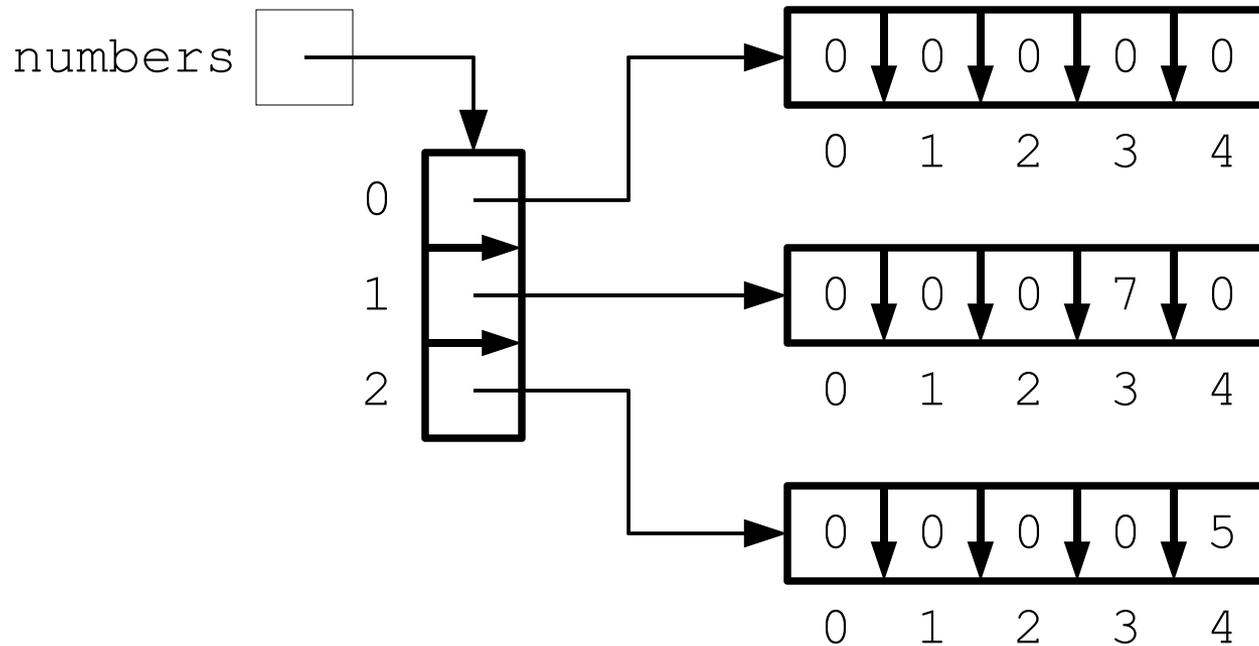
# 2D Array Access Example (2)

- Consider the expression `numbers[1]`
  - This expression refers to the element at position 1 of a two-dimensional array of `ints`
  - This element is an entire one-dimensional array of `ints`
  - Its type is `int[]`, and therefore can be used in the same manner as a regular variable of type `int[]`

# 2D Array Access Example (3)

```
numbers[1][3] = 7;
```

```
numbers[2][4] = numbers[1][3] - 2;
```



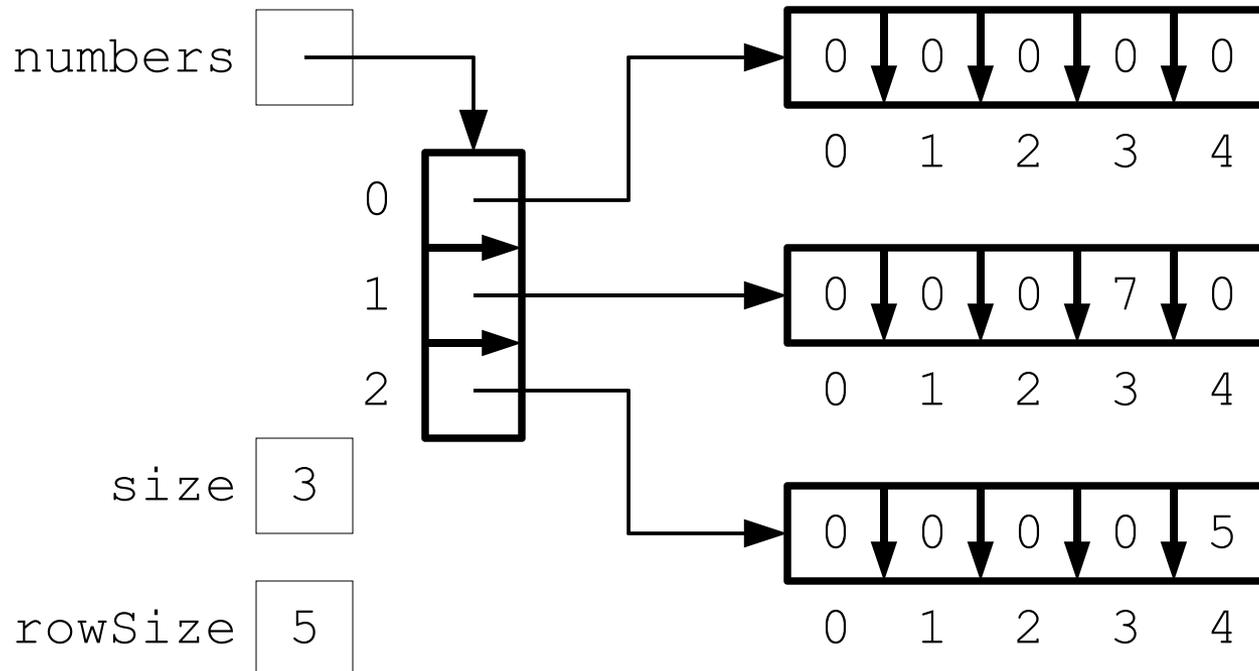
# 2D Arrays and length Fields (1)

- Because a two-dimensional array is an array of arrays, the actual value of the `length` field depends on the array whose length is being accessed
  - The expression `variableName.length` specifies the number of one-dimensional arrays in the two-dimensional array whose address is stored in variable `variableName`
  - The expression `variableName[index].length` refers to the maximum number of individual elements in the one-dimensional array whose address is stored at position `index` of the two-dimensional array whose address is stored in `variableName`

# 2D Array and length Fields (2)

```
int size = numbers.length;
```

```
int rowSize = numbers[0].length;
```



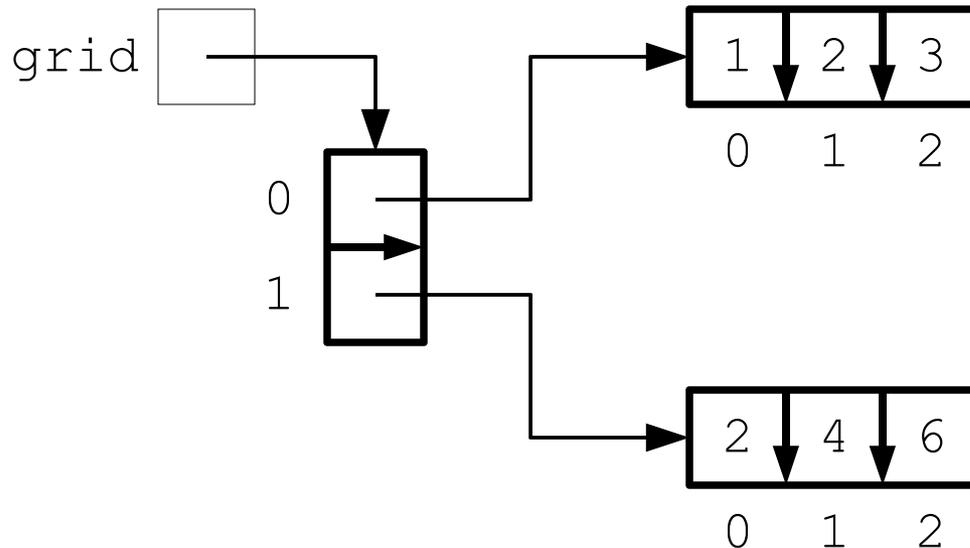
# 2D Arrays and Initializer Lists (1)

- Initializer lists can be used to initialize two-dimensional arrays upon declaration
  - The basic syntax, semantics, and restrictions are the same as when an initializer list is used to initialize a one-dimensional array
- The only difference is that each element in the initializer list for a two-dimensional array is itself the initializer list for a one dimensional array
- The following statement declares and allocates a two-dimensional array of `int` called `grid`:

```
int[][] grid = {  
    {1, 2, 3},  
    {2, 4, 6}  
}
```

# 2D Array and Initializer Lists (2)

- grid contains 2 one-dimensional arrays as elements
- Each of these one-dimensional arrays contains 3 values of type `int`



# Variation: Jagged Arrays (1)

- In a two-dimensional array, each one-dimensional array can have a different length
  - Such arrays are called *jagged arrays*, or *ragged arrays*
- We can accomplish this by not assigning a length to the one-dimensional arrays when we create the two-dimensional array, like this:

```
variableName = new type[size][];
```

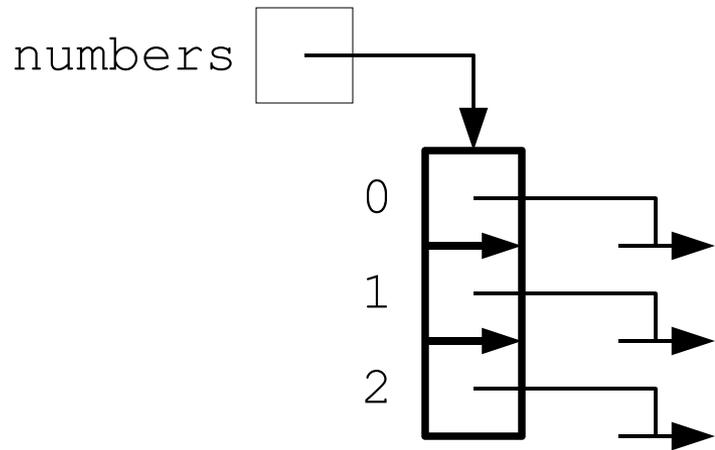
No size information 

# Variation: Jagged Arrays (2)

- The previous statement allocates an array which can hold the addresses of *size* one-dimensional arrays, and stores its address in *variableName*, of type *type*[][]
- However, none the one-dimensional arrays themselves are allocated by the above statement, and their size is unknown
- The one-dimensional arrays are allocated one-by-one normally in separate steps
  - Their sizes do not have to be equal, and are totally arbitrary
- Actual use of jagged arrays is uncommon

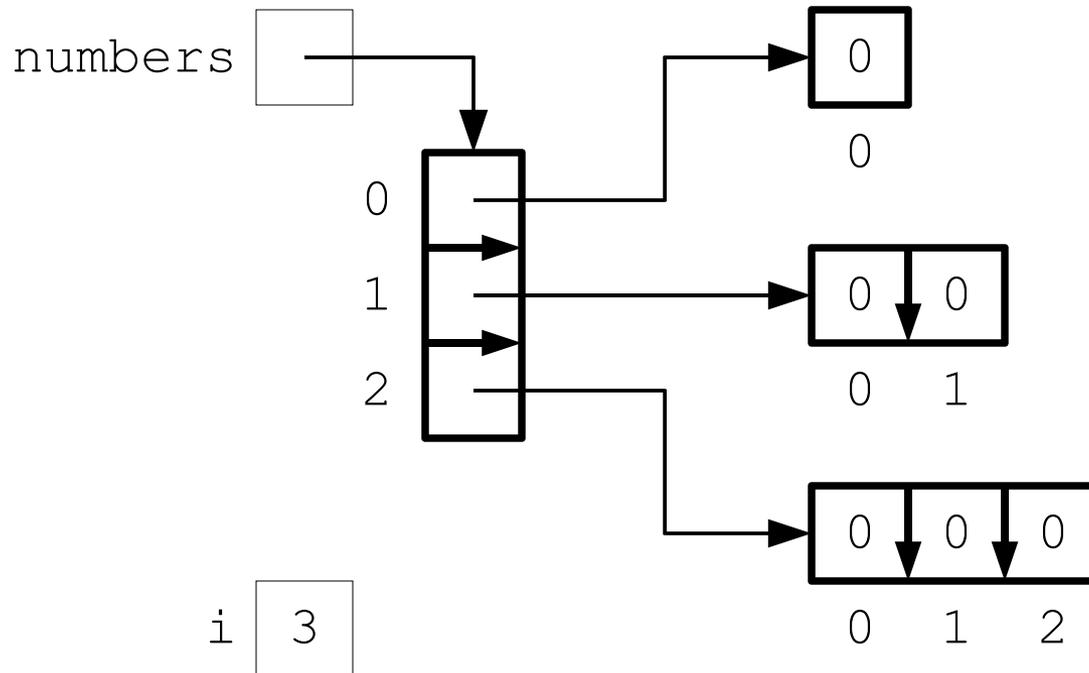
# Jagged Array Example (1)

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



# Jagged Array Example (2)

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



# Multidimensional Arrays (1)

- Multidimensional arrays are a more general case of two-dimensional arrays
- Syntactically, multidimensional arrays work just like two-dimensional arrays, except for the fact that they have more dimensions
  - All syntactic rules for one-dimensional arrays and two-dimensional arrays generalize to handle more dimensions
  - These generalizations are based on the idea that the number of square bracket pairs specifies the number of dimensions
  - For example, when accessing an element in a multidimensional array, we must specify as many indices between square bracket pairs as the array has dimensions

# Multidimensional Arrays (2)

–If the multidimensional array has  $n$  dimensions, then specifying  $m$  dimensions will refer to an  $(n-m)$ -dimensional array

- Multidimensional arrays can be perfectly rectangular, or they can be jagged
- Actual use of "pure" arrays with more than 2 dimensions is rare