

COMP 250

Lecture 5

arrays in Java

Jan. 17, 2022

Motivation for arrays

Often we have many data items that are all of some given type. These could be numbers, strings, ...

We don't want to define a separate variable for each data item.

```
int1, int2, int3, ..., int500
```

Rather we want a single data structure where we can access these data items using a number index.

Example: an array of integers

0	-7
1	1
2	-25
3	3
4	-15
5	302
6	67
7	13
8	290

This is an array of integers of length 9.

Declaring an array variable

Just like with primitive types, we can declare array variable without initializing it. Here we do not specify the size of the array that it will reference.

Let's work with the example of `double` rather than `int`.

```
int[]    arr1    ;  
double[] arr2    ;  
double   arr3[]  ;
```

} Both notations
are allowed.

We might not know in advance how big the array needs to be.

Constructing a new array

0	0.0
1	0.0
2	0.0
3	0.0
4	0.0
5	0.0
6	0.0
7	0.0
8	0.0

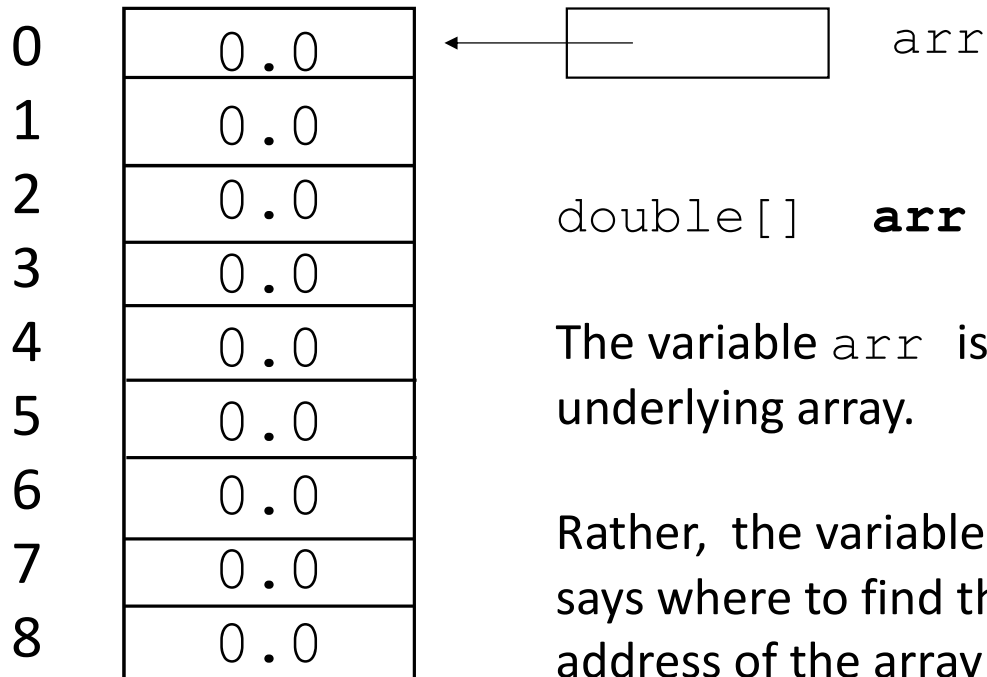
In Java, we *construct* a new array as follows.

```
double[] arr = new double[9];
```

The `new` keyword is required.

The values are initialized to 0.0 .

Reference variable

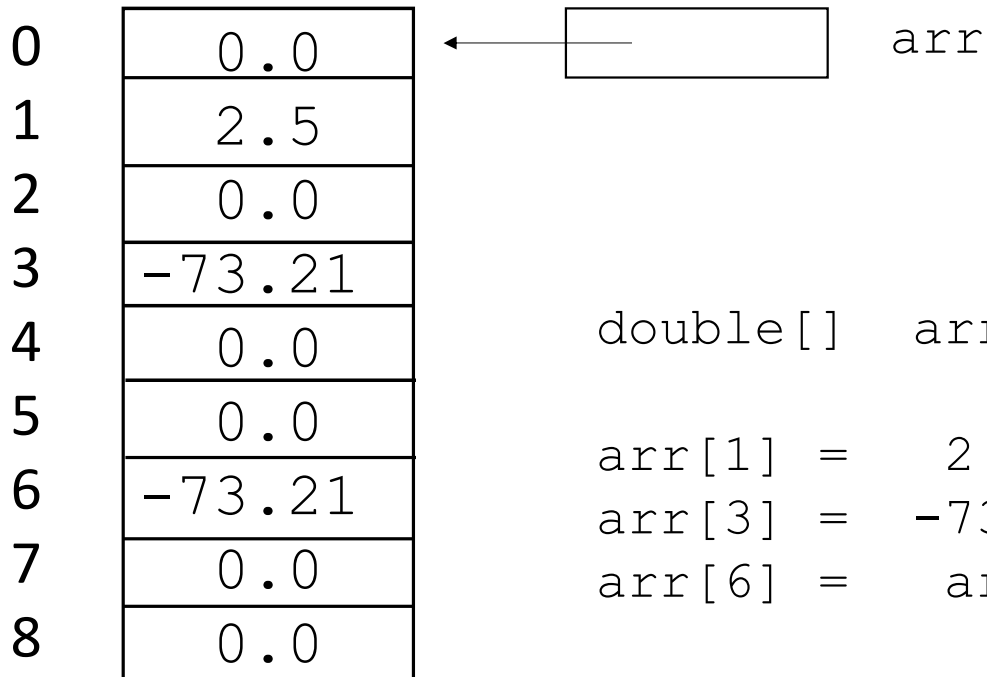


```
double[] arr = new double[9];
```

The variable `arr` is not the same thing as the underlying array.

Rather, the variable `arr` “references” the array. It says where to find the array. Think of it as holding the address of the array in memory.

Example



```
double[] arr = new double[9];
```

```
arr[1] = 2.5;
```

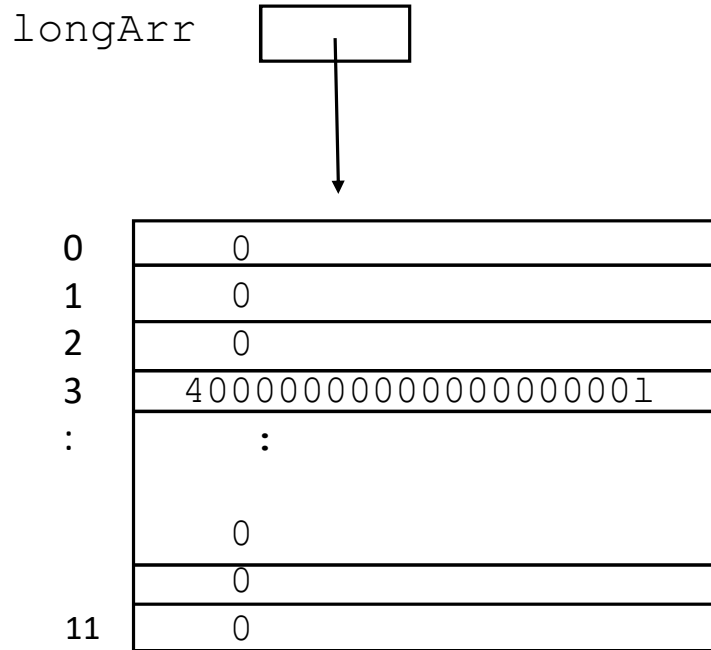
```
arr[3] = -73.21;
```

```
arr[6] = arr[3];
```

.

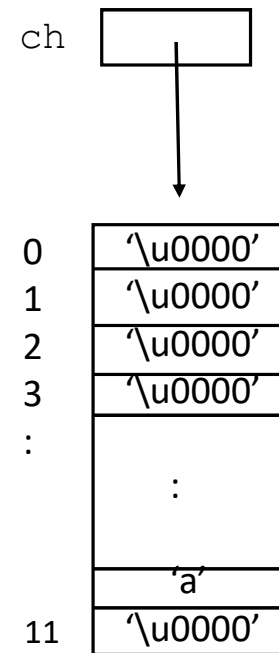
```
long[] longArr = new long[12];
```

```
longArr[3] = 40000000000000000000000001L;;
```



To define a literal that is larger than the largest int, we have to append an L. This is similar to how we needed to append an f when we want to define a float literal rather than double.

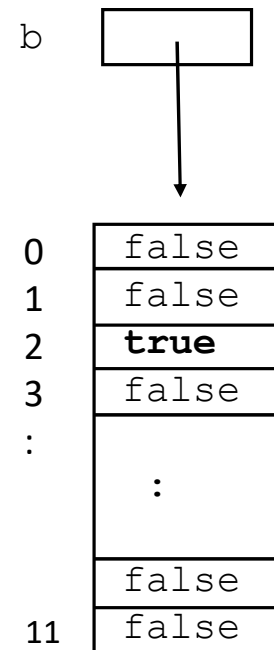

```
char[] ch = new char[12];  
ch[10] = 'a';
```



Initialized to
Unicode value 0.

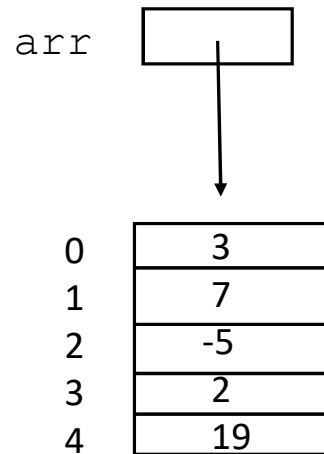
```
boolean[] b = new boolean[12];  
b[2] = true;
```

Initialized to false



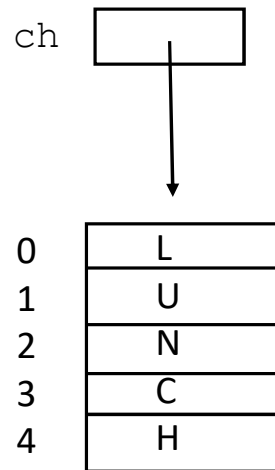
Array Initialization (hard coded)

```
int[] arr = {3, 7, -5, 2, 19};
```



Array Initialization (hard coded)

```
char[] ch = { 'L', 'U', 'N', 'C', 'H' };
```



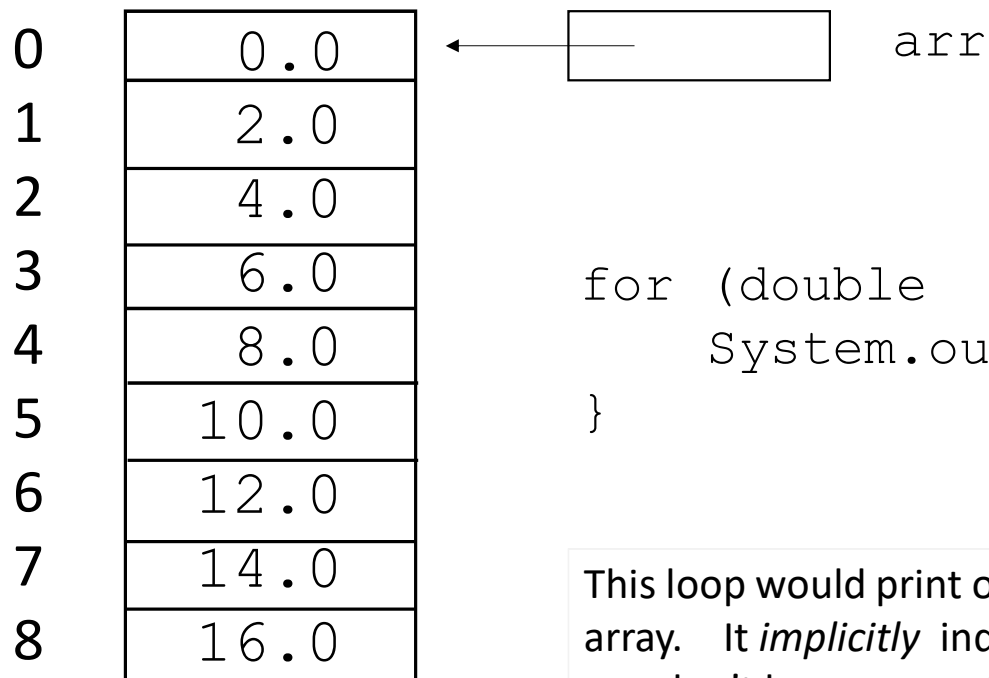
Array length

0	0.0	←		arr
1	2.0			
2	4.0			
3	6.0			
4	8.0			
5	10.0			
6	12.0			
7	14.0			
8	16.0			

```
for (int i=0; i < arr.length; i++) {  
    arr[i] = 2.0 * i;  
}
```

This dot notation will come up often.

Java “enhanced for loop”



```
for (double d : arr){  
    System.out.println( d );  
}
```

This loop would print out the *values* that are stored in the array. It *implicitly* indexes the elements of the array, but you don't have access to the indices.

Shifting elements in an array

```
for (int j=1; j < arr.length; j++){  
    arr[j-1] = arr[j];  
}
```

BEFORE

0	2.0
1	4.0
2	6.0
3	8.0
4	10.0
5	12.0
6	14.0

AFTER ?

Does this shift backwards or forwards?

What are the “edge cases” ?

Shifting elements in an array

```
for (int j=1; j < arr.length; j++) {  
    arr[j-1] = arr[j];  
}
```

BTW, notice that I am not drawing the `arr` variable in this sequence of slides. This is just to remove the clutter.

BEFORE

0	2.0
1	4.0
2	6.0
3	8.0
4	10.0
5	12.0
6	14.0

AFTER

0	4.0
1	6.0
2	8.0
3	10.0
4	12.0
5	14.0
6	14.0

How to shift forwards ?

```
for (int j=1; j < arr.length; j++){  
    arr[j] = arr[j-1];  
}
```

BEFORE

0	2.0
1	4.0
2	6.0
3	8.0
4	10.0
5	12.0
6	14.0

AFTER ?

How to shift forwards ?

```
for (int j=1; j < arr.length; j++){  
    arr[j] = arr[j-1];  
}
```

BEFORE

0	2.0
1	4.0
2	6.0
3	8.0
4	10.0
5	12.0
6	14.0

AFTER (*oopsie!*)

0	2.0
1	2.0
2	2.0
3	2.0
4	2.0
5	2.0
6	2.0

Suggestions?

How to shift forwards ?

```
for (int j = arr.length-1; j > 0; j--){  
    arr[j] = arr[j-1];  
}
```

BEFORE

0	2.0
1	4.0
2	6.0
3	8.0
4	10.0
5	12.0
6	14.0

AFTER

0	2.0
1	2.0
2	4.0
3	6.0
4	8.0
5	10.0
6	12.0

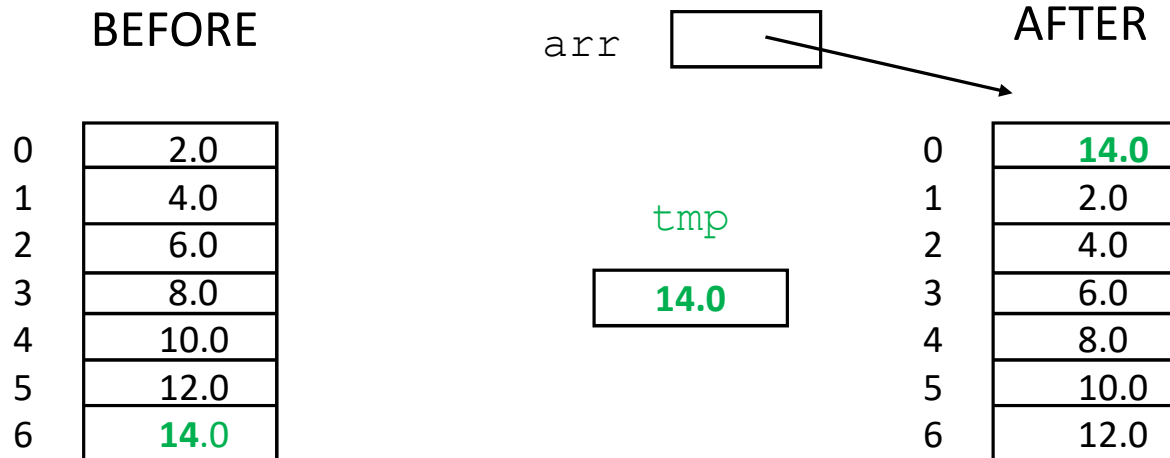
“Circular shift” (forward)

How can we shift forward, *and move the last element to the beginning?*

BEFORE		AFTER	
0	2.0	0	14.0
1	4.0	1	2.0
2	6.0	2	4.0
3	8.0	3	6.0
4	10.0	4	8.0
5	12.0	5	10.0
6	14.0	6	12.0

“Circular shift” (forward)

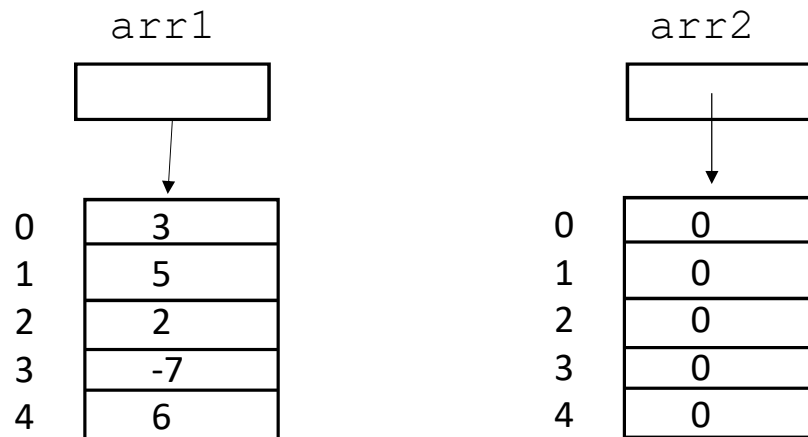
```
int tmp = arr[ arr.length-1 ];  
for (int j = arr.length-1; j > 0; j--){  
    arr[j] = arr[j-1];  
}  
arr[0] = tmp;
```



Duplicating an array

```
int[] arr1 = {3, 5, 2, -7, 6};  
int[] arr2 = new int[ arr1.length ];  
  
arr2 = arr1;    // What would this do ?
```

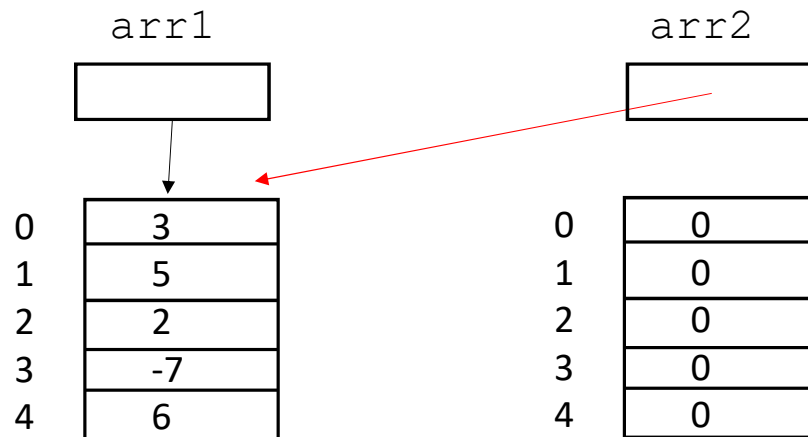
BEFORE



Duplicating an array

```
int[] arr1 = {3, 5, 2, -7, 6};  
int[] arr2 = new int[ arr1.length ];  
  
arr2 = arr1;           // oopsie!!!
```

AFTER

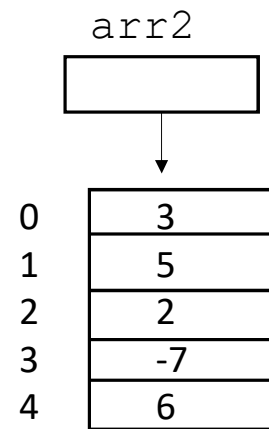
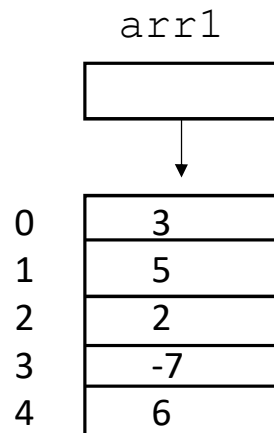


Now, nothing references this array. It becomes "garbage".

Duplicating an array


```
int[] arr1 = {3, 5, 2, -7, 6};  
int[] arr2 = new int[ arr1.length ];  
for (int m=0; m < arr1.length; m++){  
    arr2[m] = arr1[m];  
}
```

AFTER



Explained in a few lectures

Passing an array to a method



```
static void demoPassArray ( double[] doubleArray ) {  
    doubleArray[0] = 23.45;  
}
```

Suppose you call this method in the code below e.g. in main method:

```
double[] arr = {3.0, 5.2, 2.1, -7.78, 6.0};  
demoPassArray( arr );  
System.out.print( arr[0] );
```

Q: What is printed ?

Passing an array to a method

```
static void demoPassArray ( double[] doubleArray ) {  
    doubleArray[0] = 23.45;  
}
```

Suppose you call this method in the code below e.g. in main method:

```
double[] arr = {3.0, 5.2, 2.1, -7.78, 6.0};  
demoPassArray( arr );  
System.out.print( arr[0] ); A: 23.45
```

Passing a primitive type to a method

```
static void demoPassDouble ( double x ) {  
    x = 175.0;  
}
```

Suppose you call this method in the code below e.g. in main method:

```
double y = 2.0;  
demoPassDouble( y )  
System.out.print( y );
```

Q: What is printed ?

Passing a primitive type to a method

```
static void demoPassDouble ( double x ) {  
    x = 175.0;  
}
```

Suppose you call this method in the code below e.g. in main method

```
double y = 2.0;  
demoPassDouble ( y )  
System.out.print ( y ); A: 2.0
```

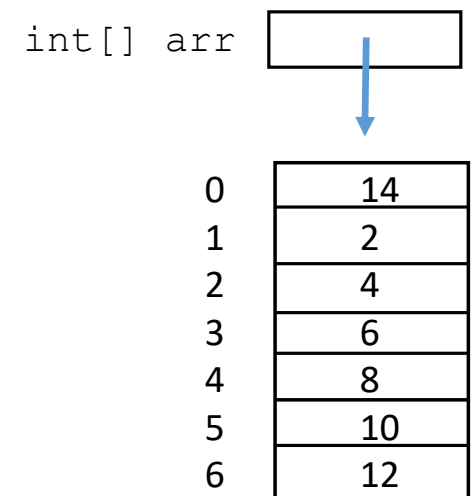
Note the behavior is very different here!

Java methods: pass by 'value'

In Java, the *value* of a parameter is passed to the method.

In the case of the array, the value is a reference to an array.
You can think of this value as a location in memory.

In the case of a primitive type, the value is the
number/character/Boolean stored in memory rather than
the location in memory.



Method for duplicating an array

The following method makes a copy of an array and returns a reference to this copy.

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

Two Dimensional (2D) Arrays

```
int[][] matrix1 = new int[4][5];
```

```
matrix1[2][4] = 345;
```

0	0	0	0	0
0	0	0	0	0
0	0	0	0	345
0	0	0	0	0

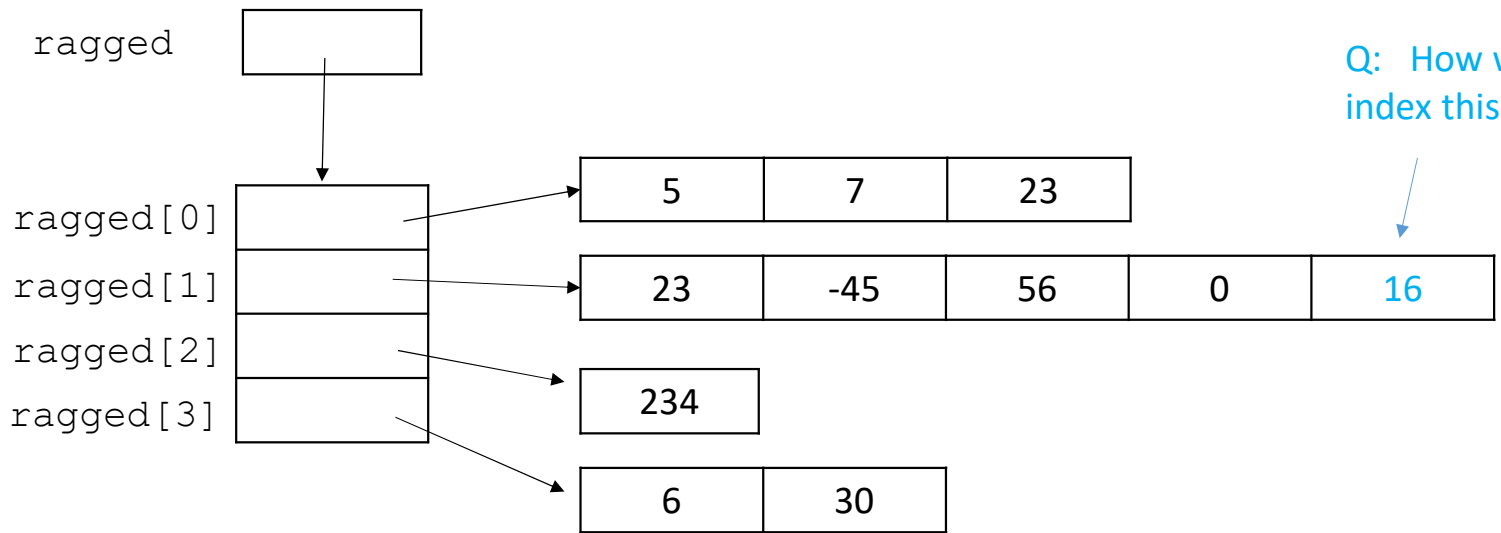
Two Dimensional (2D) Arrays

```
int[][] matrix2 = { {5, 7, 23, 3, 65},  
                    {23, -45, 56, 0, 16},  
                    {234, 3, -564, 3, 345},  
                    {6, 30, 46, 23, 23} };
```

5	7	23	3	65
23	-45	56	0	16
234	3	-564	3	345
6	30	46	23	23

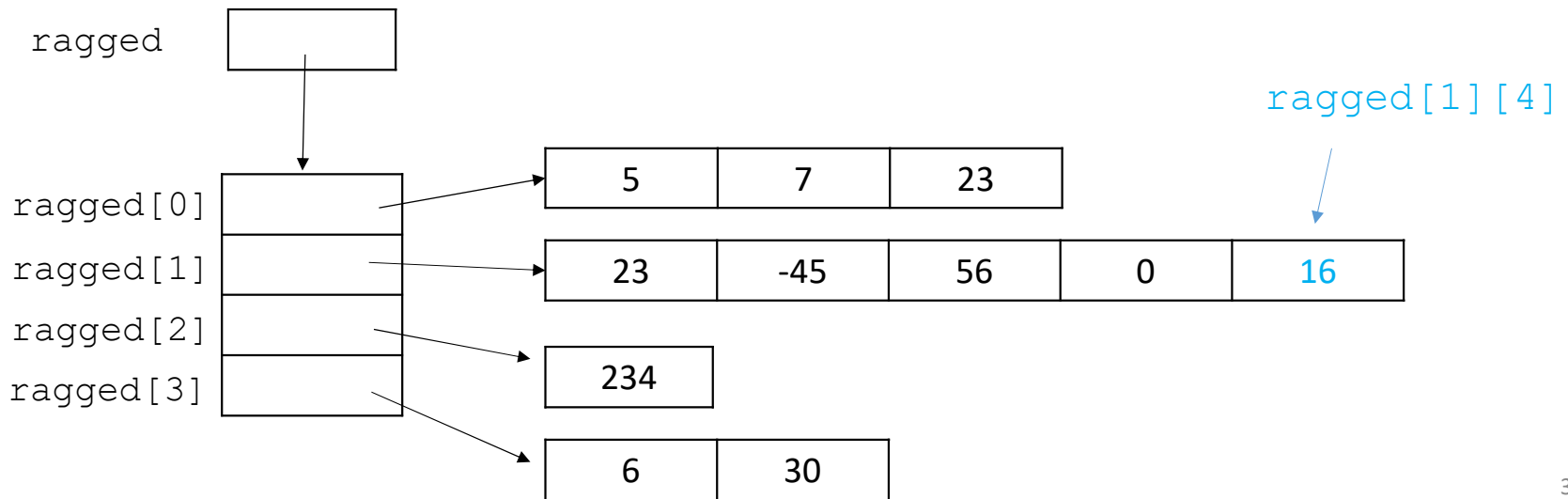
More general 2D Array: a 1D Array of 1D Arrays (a.k.a. “Jagged” or “Ragged” Arrays)

```
int[][] ragged = { {5, 7, 23},  
                  {23, -45, 56, 0, 16},  
                  {234},  
                  {6, 30} };
```



More general 2D Array: a 1D Array of 1D Arrays (a.k.a. “Jagged” or “Ragged” Arrays)

```
int[][] ragged = { {5, 7, 23},  
                  {23, -45, 56, 0, 16},  
                  {234},  
                  {6, 30} };
```



N-Dimensional Arrays

For example, a video is a sequence of image frames.
It is an N-dimensional array.

Q: What is N here ?

N-Dimensional Arrays

For example, a video is a sequence of image frames.
It is an N-dimensional array.

Q: What is N here ?

A: 4 (2 for pixel image position, 1 for color, 1 for time)

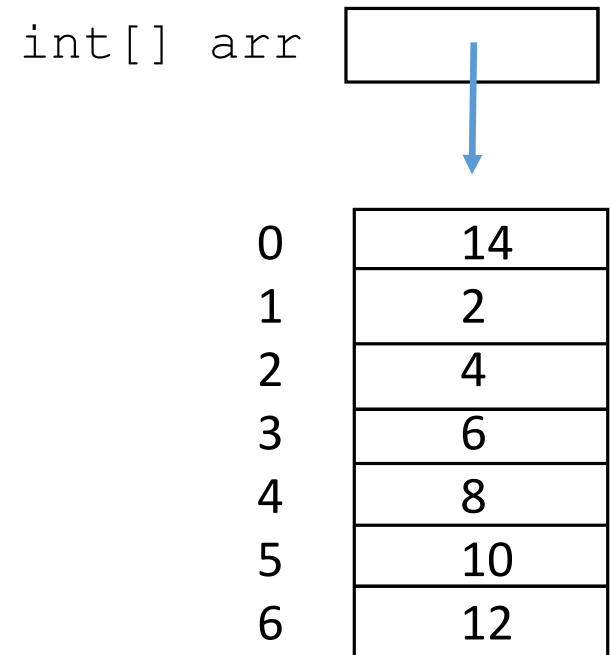
```
int [Nrows] [Ncols] [3] [Nframes] video;
```



three color channels (RGB, or red, green, blue)

Array access time

The array itself is a sequence of consecutive locations (slots) in memory. Each slot requires the same number of bytes, depending on whether the array is a primitive type (byte, int, double, ...) or a reference type (next lecture).



Arrays have “constant time” access

The time it takes the computer to read from or write to an element in an array is constant, i.e. independent of the length N of the array.

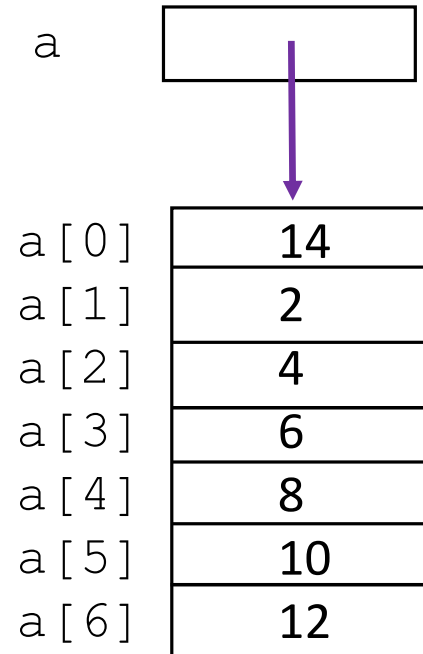
```
x      =  a[k];           //  read
a[k]   =  x ;            //  write
```

ASIDE: What happens *at the level of machine code* ?

The location or address of $a[k]$ is computed as follows,

where **address(a)** is the address of the first slot of the array:

$$\text{address}(a) + k * \text{number_of_bytes_per_array_slot}$$



Coming up...

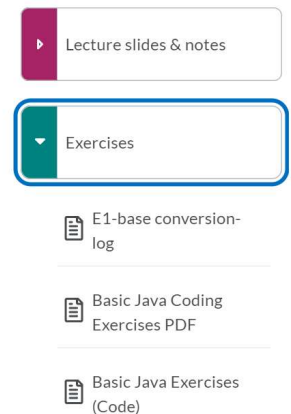
Lectures

Wed. Jan. 19 objects & classes 1:
(wrapper classes, strings)

Fri. Jan. 21 objects & classes 2

Homework (TODO)

Basic Java coding exercises
(with solutions)



Assignment 1 to be posted Fri. Jan. 28 (2 weeks).