

Arrays + Loops Warmup

```
int[] nums = {4, 8, 2, 7}
```

```
for (int i = 0; i < nums.length; i++) {  
    println(nums[i]);  
}
```

- Arrays hold pieces of data in one place
- Loops allow us to iterate thru arrays to perform some action on each element

We can modify
every element:

```
for (int i = 0; i < nums.length; i++) {  
    nums[i] = nums[i] * 2;  
}
```

Primitive Variables vs. Array References

int x = 5;

x 5

int y = x;

y ~~5~~ 10

y = 10;

← doesn't change x

with primitive variables, the
value set is just a copy.

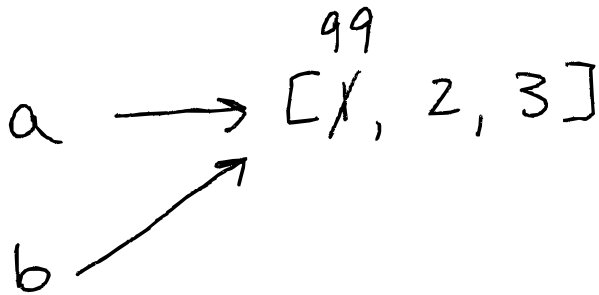
What would this print?

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

```
int[] b = a;
```

```
b[0] = 99;
```

```
println(a[0]);
```



```
b[0] = 99
```

Takeaway: an array variable does not contain the entire array. It contains a reference to the array.

You can have more than one reference to an array (as shown in our last example)

References Through Methods

Consider this method:

```
public static void changeFirst(int[] nums) {  
    nums[0] = 100;  
}
```

consider this code:

```
int[] values = {1, 2, 3};  
changeFirst(values);  
System.out.println(values[0]);
```

* What does this print?

values $\longrightarrow$ [1, 2, 3]

values is the name of the main method's reference to [1, 2, 3].

We pass values into changeFirst().

The method's reference to [1, 2, 3] is called nums.

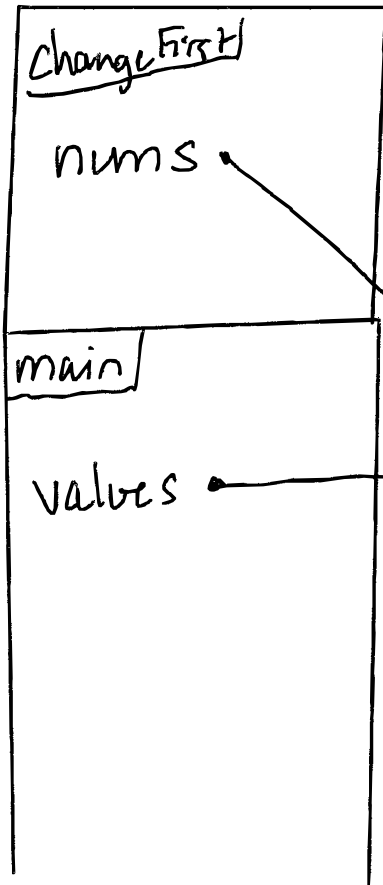
when we call changeFirst(values), we create the method's reference to [1, 2, 3] (nums).

values $\longrightarrow$ [1, 2, 3]

nums $\nearrow$

Values and nums are both references to the same array.

values scope is the main method



nums scope is our call to `changeFirst()`

100

~~1~~, 2, 3]

Let's try the same
idea but with a primitive
type instead of an
array.

```
public static void change(int x) {  
    x = 100;  
}
```

```
int n = 5;  
change(n);
```

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

Prints 5!

```
public static void change(int x) {  
    x=100;  
}
```

Passing a primitive type
to a method creates a
copy of the variable.

Changes are only reflected
in the copy, and then
are lost when the program
returns.

(Draw stack diagram)

But wait. Aren't Strings references too?

```
String s = "hello";
```

```
String t = s;
```

```
t = t.toUpperCase();
```

This doesn't make `s = "HELLO"`

This is because Strings are immutable (can't change).

- An array let's us change the elements themselves
- operations that "modify" Strings actually produce a new String.

```
String s = "hello";
```

```
String t = s;
```

```
t = t.toUpperCase();
```

if we print t : "HELLO"

What if we change the code a bit?

```
String s = "hello";
```

```
String t = s;
```

```
t.toUpperCase(); ←
```

if we print t : "hello"

Arrays are great.

But we can't change their size.

What if we don't know ahead
of time if you need 5, 10, 500...

Solution: ArrayList

```
ArrayList<String> names =  
    new ArrayList<>();
```

```
names.add("Milo");
```

```
names.add("Kelly");
```

```
names.add("Alex");
```

ArrayLists:

- can change size
- all elements must be the same type
- are ordered

ArrayLists are ordered:

if we add items to a list, the order in which they are added is preserved.

```
ArrayList<String> names =  
    new ArrayList<>();
```

```
names.add("Milo");
```

```
names.add("Kelly");
```

```
names.add("Alex");
```

```
names.get(0); //Milo
```

```
names.get(1); //Kelly
```

Useful ArrayList Methods

```
ArrayList<String> names =
```

```
    new ArrayList<>();
```

```
names.add("Milo");
```

```
names.add("Kelly");
```

```
String Kelly = names.get(1);
```

```
names.set(0, "Bob");
```

```
names.remove(0);
```

```
names.size();
```

get(index) → retrieve element

set(index, value) → update an existing element

remove(index) → remove existing element

size() → get length/size of ArrayList

add(value) → add new element to end of list with value

Arrays vs ArrayLists

array.length

list.size()

array[i]

list.get(i)

array[i] = value;

list.set(i, value);

Empty Initialization

```
List<String> list = new ArrayList<>();
```

```
list.add("apple");
```

```
list.add("banana");
```

```
list.add("orange");
```

```
String[] array = new String[2];
```

```
array[0] = "apple";
```

```
array[1] = "banana";
```

```
// can't add "orange" because arrays  
// have fixed size upon initialization.
```

Another Way to Initialize an ArrayList : with known elements

```
List<String> list =
```

```
new ArrayList<>(List.of("apple",  
                        "banana", "orange"));
```

// we can still add elements
after initialization!

```
list.add("papaya");
```

And remember: Arrays & ArrayLists are
ordered: ["apple", "banana", "orange",
"papaya"]

Remember : Arrays can be

initialized with elements:

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

* but we cannot change its size after initialization

Arrays vs ArrayLists

- arrays are memory (space) efficient
- arrays are contiguous in memory, meaning right next to each other, which makes access fast
- ArrayLists allow us to resize
- ArrayLists live in a special part of memory called the heap, they are not contiguous in memory

Exercise: Print Backwards

Write a program that asks the user for 5 integers, and then prints the list of numbers in reverse order

How do we break this into steps?

Exercise: Print Backwards

Write a program that asks the user for any number of integers and prints them in reverse order.