CSE 121 Section 13: Reference Semantics

This worksheet is primarily about this block of code:

```
1   int[] numCats = new int[]{7, 8, 9, 10};
2   // End of Part 1
3
4   numCats[0] = 1;
5   numCats[numCats.length - 1] = 0;
6   numCats[2] = -2;
7   // End of Part 2
8
9   int firstCat = numCats[0];
10  int secondCat = numCats[1];
11  firstCat++;
12  secondCat--;
13  numCats[3] = 21;
14  // End of Part 3
15
16  int[] numTuxedoCats = numCats;
17  int[] numOrangeCats = new int[4];
18  for (int i = 0; i < numCats.length; i++) {
19      numOrangeCats[i] = numCats[i];
20  }
21  // End of Part 4
22  numCats[0] += numCats[1];
23  numTuxedoCats[2] = numOrangeCats[3];
24  numOrangeCats[2] = numTuxedoCats[3];
25  // End of Part 5
26
```

You may assume that this code compiles and runs with no errors.

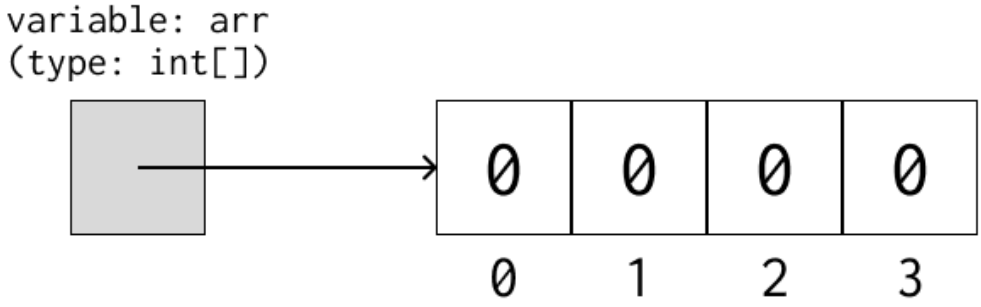
Reminder: Drawing Array Diagrams

Tracing through array code can get *very* tricky! We highly recommend drawing diagrams as you trace through this code. For example, let's say we ran the code:

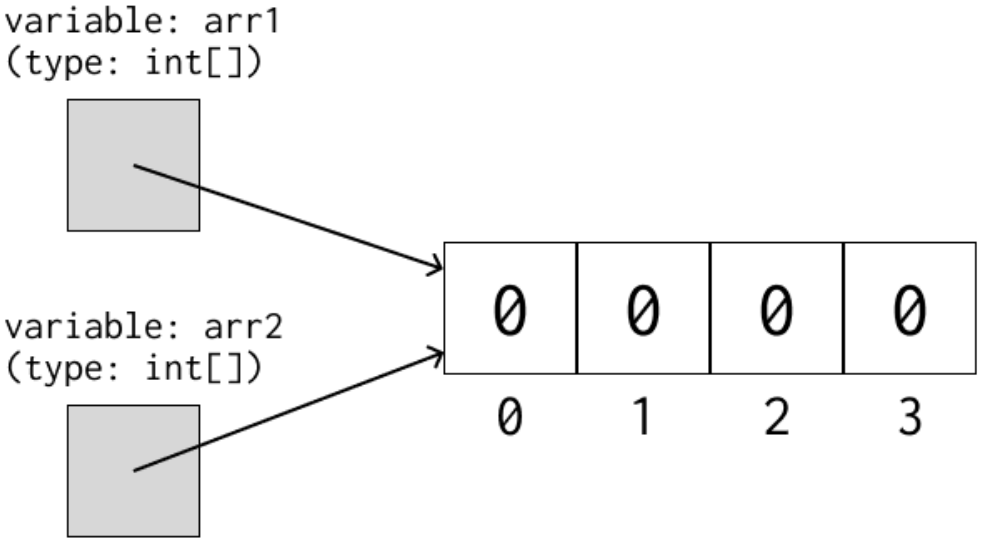
```
1  int[] arr = new int[4];
```

We could represent that code with the following diagram; note how:

- we draw a "box" for the variable `arr`, with its type
- the box refers to (or "points to") the underlying array
- when drawing the array, we label both
    - the values of each element (the `0`'s inside the boxes)
    - the indices for each element (the `0`, `1`, `2`, `3`) under the boxes



This extends to situations where two variables can point to the *same* array! Here's one such way to draw that diagram:



Part 1

Execute the code up to line 2. Draw an array diagram for `numCats`.

Code executed (line 1):

1	`int[] numCats = new int[]{7, 8, 9, 10};`

Part 2

Execute the code up to line 7. Draw an updated array diagram for `numCats`.

Code executed (lines 3–6):

3	`numCats[0] = 1;`
4	`numCats[numCats.length - 1] = 0;`
5	`numCats[2] = -2;`
6	`// End of Part 2`

`numCats[0] = 1` sets index 0 to 1.

`numCats[numCats.length - 1] = 0` evaluates `numCats.length - 1` as 3, then sets index 3 to 0.

`numCats[2] = -2` sets index 2 to -2.

Part 3

Execute the code up to line 14. Draw an updated array diagram for `numCats`.

Code executed (lines 9–13):

```
9   int firstCat = numCats[0];
10  int secondCat = numCats[1];
11  firstCat++;
12  secondCat--;
13  numCats[3] = 21;
```

`int firstCat = numCats[0]` sets `firstCat` to 1. This is a primitive copy, so changes to `firstCat` do NOT affect the array.

`int secondCat = numCats[1]` sets **`secondCat`** to 8. This is a primitive copy, so changes to `secondCat` do NOT affect the array.

`firstCat++` sets `firstCat` to 2. This does not change `numCats[0]`.

`secondCat--` sets `secondCat` to 7. This does not change `numCats[1]`.

`numCats[3] = 21` sets index 3 to 21.

Part 4

Execute the code up to line 21. Draw array diagrams for `numCats`, `numOrangeCats`, and `numTuxedoCats`. Pay special attention to the number of unique arrays.

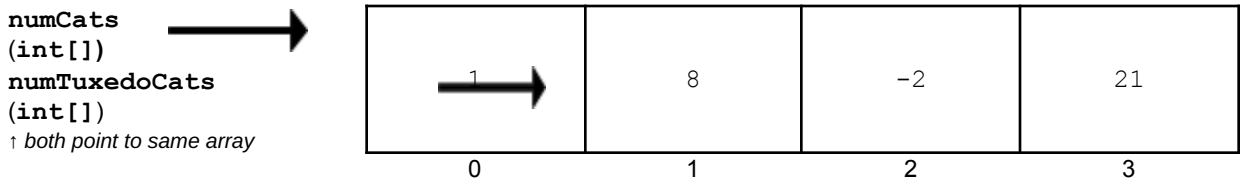
Code executed (lines 16–20):

```
16  int[] numTuxedoCats = numCats;
17  int[] numOrangeCats = new int[4];
18  for (int i = 0; i < numCats.length; i++) {
19      numOrangeCats[i] = numCats[i];
20  }
```

`int[] numTuxedoCats = numCats` assigns `numTuxedoCats` to refer to the same array as `numCats`. There is no copy made; there is still only one array in memory.
`int[] numOrangeCats = new int[4]` creates a new, separate array of 4 zeros, then the for loop copies values from `numCats` into it element by element.

Result: there are 2 unique arrays. `numCats` and `numTuxedoCats` share one array; `numOrangeCats` has its own separate copy with the same values.

numCats and numTuxedoCats both point to same shared array:



numOrangeCats is a separate array (deep copy of values):

Part 5

Execute the code up to line 26 (all of the code). Draw updated array diagrams for `numCats`, `numOrangeCats`, and `numTuxedoCats`.

Code executed (lines 22–24):

```
22  numCats[0] += numCats[1];
23  numTuxedoCats[2] = numOrangeCats[3];
24  numOrangeCats[2] = numTuxedoCats[3];
```

`numCats[0] += numCats[1]` sets `numCats[0]` to 1 + 8 = 9. Because `numTuxedoCats` is a reference to the same array, it also immediately reflects this change.

`numTuxedoCats[2] = numOrangeCats[3]` reads `numOrangeCats[3]` as 21, so `numTuxedoCats[2]` becomes 21.
This writes to the shared array, so `numCats[2]` also becomes 21.

`numOrangeCats[2] = numTuxedoCats[3]` reads `numTuxedoCats[3]` as 21 (unchanged), so `numOrangeCats[2]` becomes 21.
This writes to `numOrangeCats`'s separate array.

numCats and numTuxedoCats, same shared array (updated):

numOrangeCats, separate array (updated):

Part 6

Assume that we had the following code, executed *after* all the previous code:

```
27  for (int i = 0; i <= numCats.length; i++) {
28      numTuxedoCats[i] += 3;
29  }
```

This code will cause an **`ArrayIndexOutOfBoundsException`**!
Which value of `i` will cause this exception?

The loop condition is `i <= numCats.length` (note <=, not <).
`numCats.length = 4`, so `i` takes values 0, 1, 2, 3, then `4`. Valid indices for a length-4 array are 0–3.
Note: because `numTuxedoCats` and `numCats` point to the same array, accessing `numTuxedoCats[4]` is also out of bounds.

Exception thrown when **`i = 4 (numCats.length)`**

Part 7

Assume that we instead had the following code, executed *after* all the previous code:

```
27  for (int i = 0; i < numOrangeCats.length; i++) {
28      numOrangeCats[i] = numOrangeCats[i - 1] * 10;
29  }
```

This code will also cause an **`ArrayIndexOutOfBoundsException`**!

Which value of `i` will cause this exception?

The loop starts at `i = 0`. On the very first iteration, the right-hand side accesses `numOrangeCats[i - 1]` = `numOrangeCats[-1]`.
Negative indices are never valid in Java. The exception is thrown immediately on the first iteration.

Exception thrown when **`i = 0  (numOrangeCats[-1] is invalid)`**