Arrays: 1D and 2D

Arrays

- Primitive data types (e.g. int, float) can store single values (42, 3.14)
- Arrays are a way to store multiple values *of the same type*
  - Can be *any* type

Properties of Arrays

- Arrays:
  - have a *fixed* length
  - contain multiple values of the same type (homogeneous type)
    - This is incredibly important
    - We call these values *elements*
  - give meaning to the *position* of the value in the array
    - Ordering matters!
    - We we call the position a value is in the *index*

The Array Class

- As objects:
  - are directly supported by Java, but not as a class
  - implicitly extend `Object`
  - are accessed using a *reference variable*
- There is an `Array` class
  - We will not use this class much; it contains static methods used to fiddle with arrays

Declaring Arrays

Declaring an Array

```
int [] numbers; // or int numbers[]
Person [] people;
```

General Array Declaration

```
TYPE [] NAME; // or TYPE NAME[]
```

A Quick Aside About Programming

A note: there are multiple ways to declare arrays; as with anything in programming, *consistency* is key. Choose the one that makes the most sense to you and stick with it.
This actually applies to all aspects of writing code; the more consistent you are with style, format, and the constructs you use, the easier it will be to work with your code.

Creating Arrays

- Declaring the array creates the *reference*, but does not allocate space for the array
- You use the new operator to allocate space

Array Allocation

```
numbers = new int[5]; // allocs space for 5 ints
```

This can be done simultaneously with creation:

Short Hand

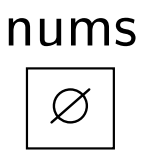
```
float[] scores = new float[50];
```

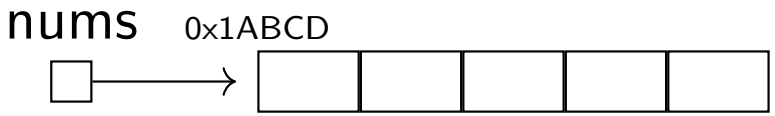
- new returns a reference to a *contiguous* block of memory
  - thus, arrays use contiguous allocation

Memory Allocation

```
int [] nums;

nums = new int[5];
```





Accessing Elements

- new returns a *reference* to the array
  - basically, the memory address
- It would be a pain to have to remember both the starting address and the address of *every* element
  - easier to just use multiple variables... gross
- BUT every element is same type, which means same size
- Knowing the position of the element we want, the size of the elements, and the starting address, we can *quickly* determine where in meory the value we want is
  - We call this position the index

Indexing

- Elements are accessed through their index
- The first element in the array has an index of 0, second has index of 1, etc
  - Called Zero Indexing
  - Maximum allowable index: array length - 1
- Elements are accessed using the `[]` operator
  1. Example: `numbers[2] = 4;`
- This is a *Cool Thing*™
  - The exact memory location can be determined with a simple, fast calculation:

$$addr = addr_{start} + index * size_{element}$$

Example

If the starting address of an array of integers is 10 and the size of an `int` is 4 bytes, what is the address of the element at index 3?
Note: I'm not a jerk, so assume base 10 (regular human numbers).

Solution

If the starting address of an array of integers is 10 and the size of an `int` is 4 bytes, what is the address of the element at index 3? Note: I'm not a jerk, so assume base 10 (regular human numbers).

$$
\begin{aligned}
addr &= addr_{start} + index * size_{element} \\
addr &= 10 + 3 * 4 \\
&= 10 + 12 \\
&= 22
\end{aligned}
$$

Processing Arrays

Arrays have a `.length` attribute (not method), so we can walk through all elements in the array with a simple for loop:

Regular For Loop: Arrays

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

Processing Arrays

Alternately, Java 5.0 introduced the Enhanced For Loop (I may call this a ranged based for loop) that is similar to Python's for loop:

Enhanced For Loop (Java 5.0 +)

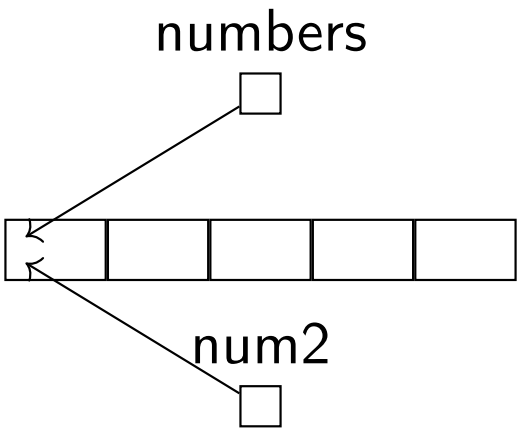
```
int sum = 0;
for (int num: numbers)
    sum += num;
```

Note that this will visit all elements *even if they are not filled/initialized*

Reminder

- Arrays are *REFERENCES*

```
int [] num2;
num2 = numbers; // NOT A COPY!
```



Copying Arrays

Copying Arrays

```
num2 = numbers.clone();
// You can also useL:
//System.arraycopy(from, start_index
//                 to, to_start_idx, num)
int num3 = new int[5];
System.arraycopy(numbers, 0, num3, 0, 5);
```

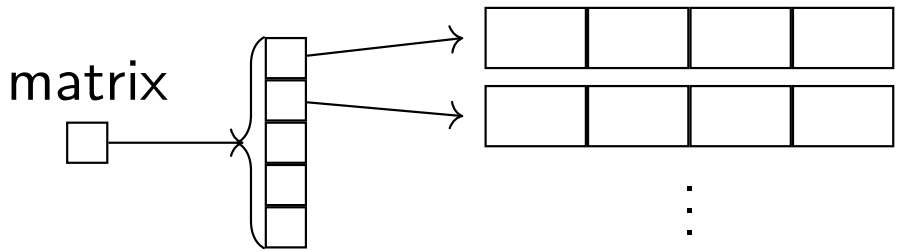
2D Arrays

- Sometimes it is useful to store a table of data
  - Matrices
  - Sudoku boards
  - Graphs (more on this at the tail end of the semester)
  - ...
- 2D Arrays in java are no different than arrays
  - arrays of arrays...

Creating 2D Arrays

Creating a 2D Array

```
int [][] matrix;
// or int matrix [][], int [] matrix []
matrix = new int[5][4];
```



Matrix Dimensions

```
matrix.length // # of rows
matrix[0].length // # of cols in row 0
```

Printing the Matrix

```
public static void printmat(int [][] mat)
{
    for (int i=0; i<mat.length; i++)
    {
        for (int j=0; j<mat[i].length; j++)
            System.out.printf("%3d", mat[i][j]);
        System.out.println("");
    }
}
```

```
printmat(mat):
0   0   0   0
0   0   0   0
0   0   0   0
0   0   0   0
0   0   0   0
```

Filing the Matrix

```
for (int i=0; i<matrix.length; i++)
    for (int j=0; j<matrix[i].length; i++)
        matrix[i][j] = i * 10 + j;
printmat(matrix);
```

```
printmat(mat):
0  1  2  3
10 11 12 13
20 21 22 23
30 31 32 33
40 41 42 43
```

Nonsquare 2D Arrays

```
int matrix2[][] = new int[3][];
for (int i=0; i<matrix2.length; i++)
    matrix2[i] = new int[i+1];
printmat(matrix2);
for (int i=0; i<matrix2.length; i++)
    for (int j=0; j<matrix2[i].length; j++)
        matrix2[i][j] = i * 10 + j;
printmat(matrix2);
```

Nonsquare 2D Arrays

```
printmat(mat):
0
0  0
0  0  0

printmat(mat):
0
10 11
20 21 22
```