Object Oriented Programming with Java

Arrays

Array Types

- The array type is indicated with **[ ]**'s.
- **Monomorphism:** Just as variables can only hold one type of value, Java arrays can only hold one specified type of value, in every slot.
- Example array types:

```
int[]   double[]   boolean[][]     Person[]
```

- The type does **not** record the size of each dimension

Array Declaration

- To declare an array, we simply declare a variable with an **array-type** as its type:

```
int[] nums;            // an array of only int values
double[] scores;       // an array of only double values
```

- Multiple dimensions can be stacked together

```
short[][] twoDims;       // a 2D array of short values
float[][][] space;       // a 3D array of float values
```

- Java arrays must entirely have the same ***number*** of dimensions. Each 'row' of a dimension will be the exact same type, e.g. `int[]`. They don't actually need to have the same length though.

Array Creation (Memory Allocation of Array)

There are two ways to allocate memory for an array:

1. explicit listing of values using `{}`'s:

```
int[] xs = {2,5,3,6,4};
double[][] ys = {{1.0, 2.2}, {0.3, 4}, {7.7, 8.9}};  //3x2 dim
```

   Only at declaration, not a later assignment. Syntax error if you attempt to use it later.

   Question: Can you pass an array to a method like this?

2. using the **new** keyword and specifying dimensions:

```
short[] xs = new short[10];       //holds exactly 10 shorts
double[][] ys = new double[10][15];  //holds 10x15 doubles
```

   In this case, all array slots have default values: `0`, `0.0`, `false`, `null`

Alternative syntax: `int[] zs = new int[]{0,1,2,3,4};`

Multidimensional arrays of varying size

Multi-dimensional arrays may have dimensions of varying size.

But there is a caveat...

All elements of the array must live in the same depth which is equal to the dimensionality of the array

Examples:

```
// correct - all elements are in depth 2
int[][] zs = {{0},{1,2,3},{4,5}};

// error - element 0 is in depth 1, not 2
int[][] zs = {0,{1,2,3},{4,5}};
```

Accessing & Modifying Arrays

Brackets **[ ]** are used to access and update values in arrays.

```
int a = xs[4];   //accesses 5th elt. of xs.
xs[0] = 7;  //replaces 1st xs elt. with a 7.
```

Any expression of type **`int`** may be used as an index, *regardless of the type in the array*:

```
xs[ a+4 ]        xs [ sc.nextInt() ]
xs[ i ]          ys[ i ][ j ]
```

The length of an array is available as an attribute:

```
xs.length      ys[i].length
```

Arrays & Loops

Loops let us address each item in an array. The basic power of procedural programming comes from this pairing of loops+arrays

```
int[] xs = new int[10];
Scanner sc = new Scanner(System.in);

//fill in an array using a loop
for (int i=0; i<xs.length; i++) {
    xs[i] = sc.nextInt();
}

// calculate the sum of #s in an array using a loop:
int sum = 0;
for (int i=0; i<xs.length; i++) {
    sum += xs[i];
}
System.out.println("Sum of your 10 #s is " + sum);
```

Arrays vs Lists

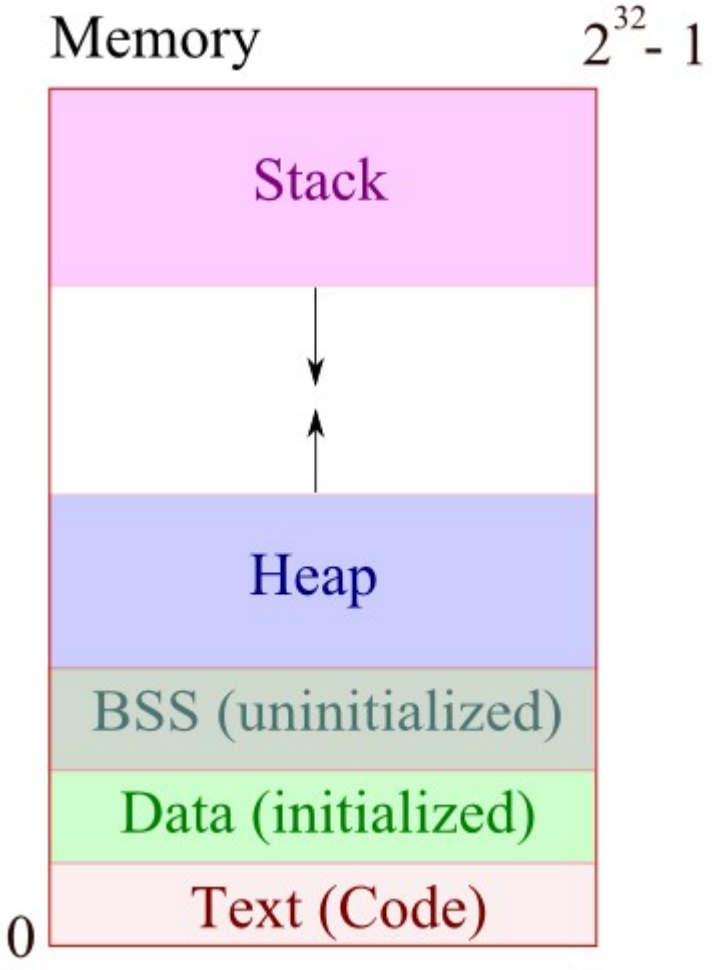
- An array is not the same as a list (e.g., Python lists)

→ Array: length permanently determined at creation. Fast access.

→ List: length can vary throughout. Slower access.

→ Lists are often implemented via intelligent use of arrays to regain some of the speed of access without losing the ease of usage.

Memory – Stack – Heap



The Stack is a **Last-In-First-Out** structure that keeps track of the function calls, their arguments and their local variables

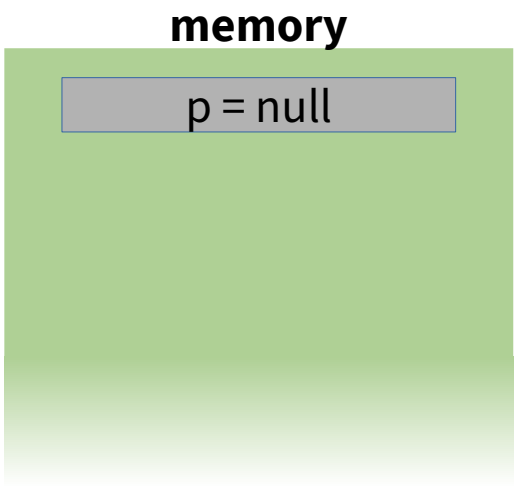
The Heap is a **Random Access** structure that stores the variables that are dynamically allocated during the execution, e.g. `array`, `objects`

The machine code of the program is generated from the source code, is stored in a separate location, and has a **fixed size** during execution

Arrays of Primitive Types in Memory

instruction pointer 

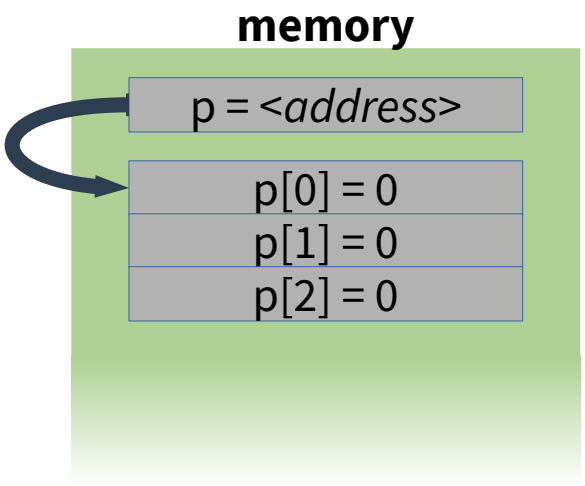
```
int[] p;

p = new int[3];
```



Arrays of Primitive Types in Memory

instruction pointer

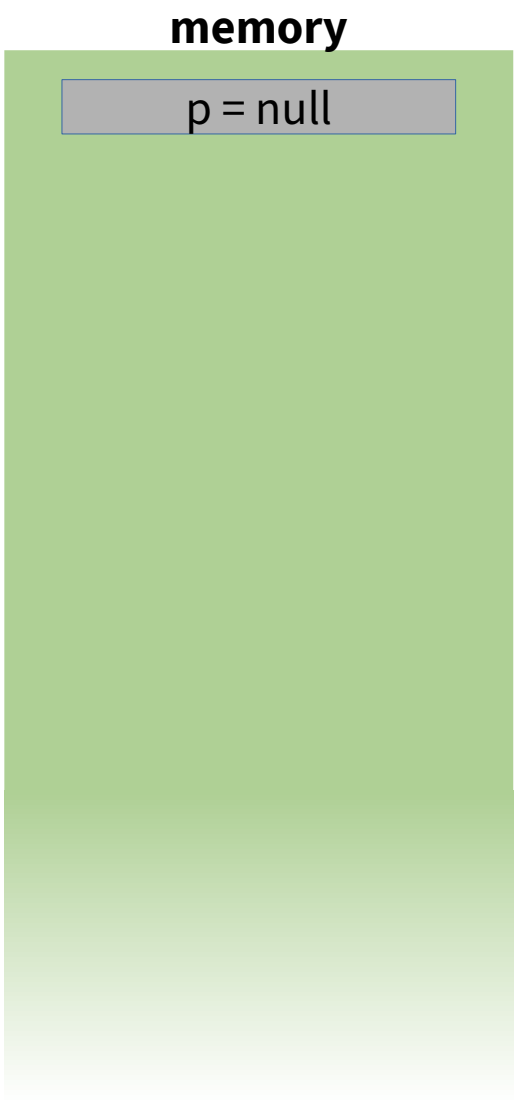
```
int[] p;

p = new int[3];
```



Arrays of Objects in Memory

instruction pointer 

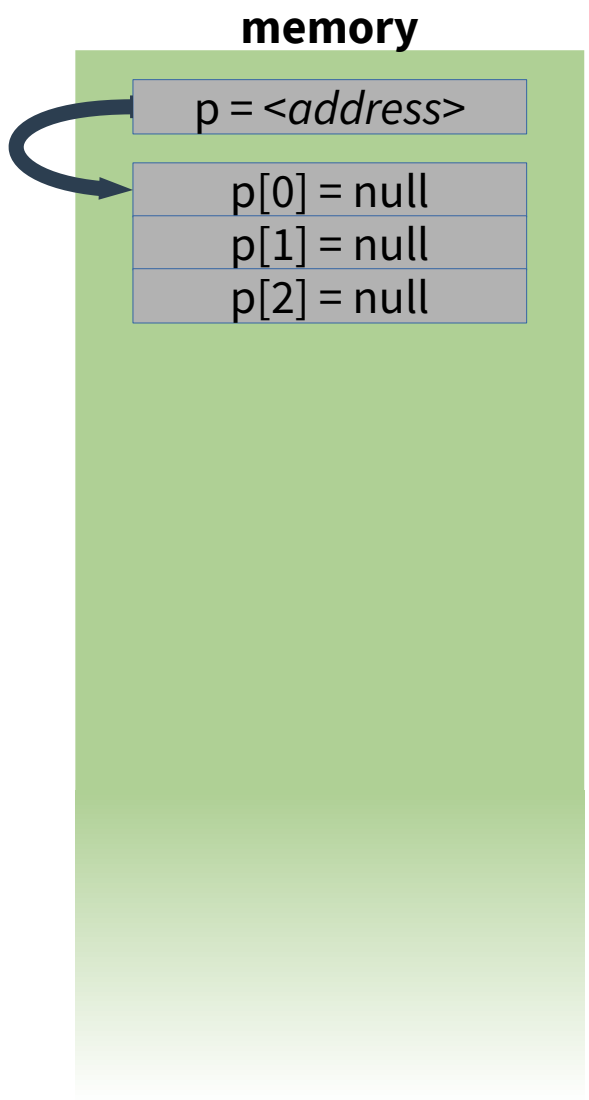
```
Point[] p;

p = new Point[3];

p[0] = new Point(1,2);

p[1] = new Point(3,4);

p[2] = new Point(5,6);
```



Arrays of Objects in Memory

instruction pointer

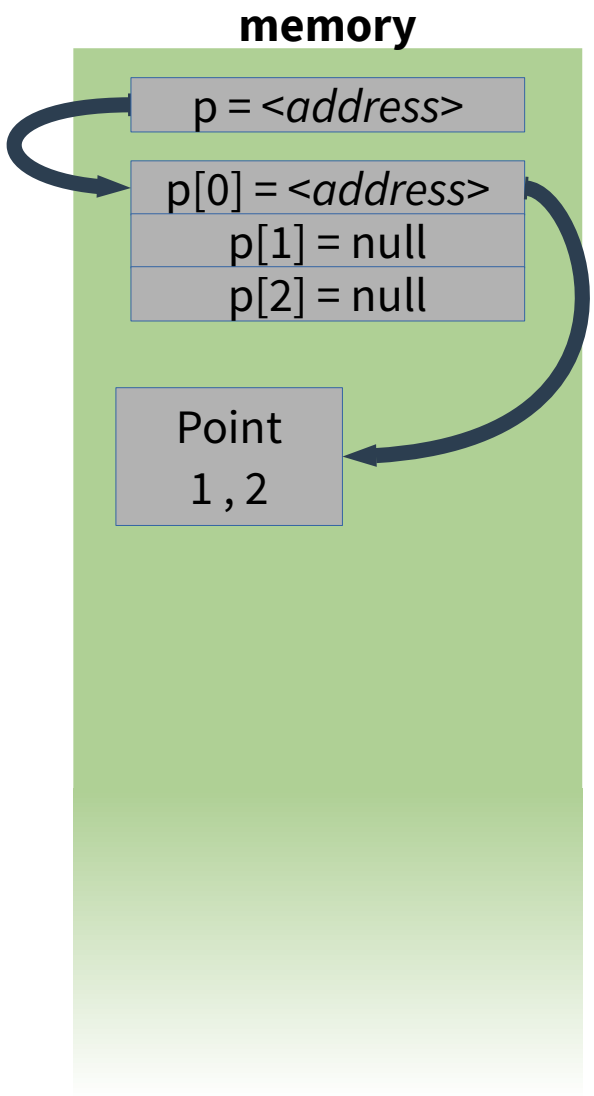
```
Point[] p;

p = new Point[3];

p[0] = new Point(1,2);

p[1] = new Point(3,4);

p[2] = new Point(5,6);
```



Arrays of Objects in Memory

instruction pointer

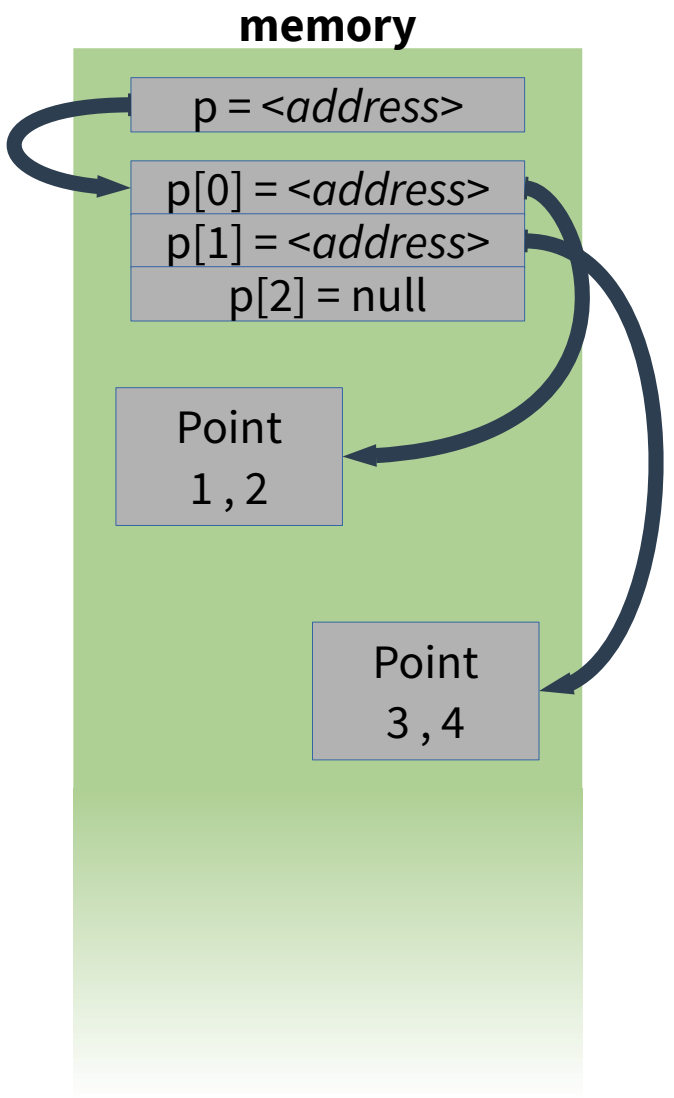
```
Point[] p;

p = new Point[3];

p[0] = new Point(1,2);

p[1] = new Point(3,4);

p[2] = new Point(5,6);
```



Arrays of Objects in Memory

instruction pointer

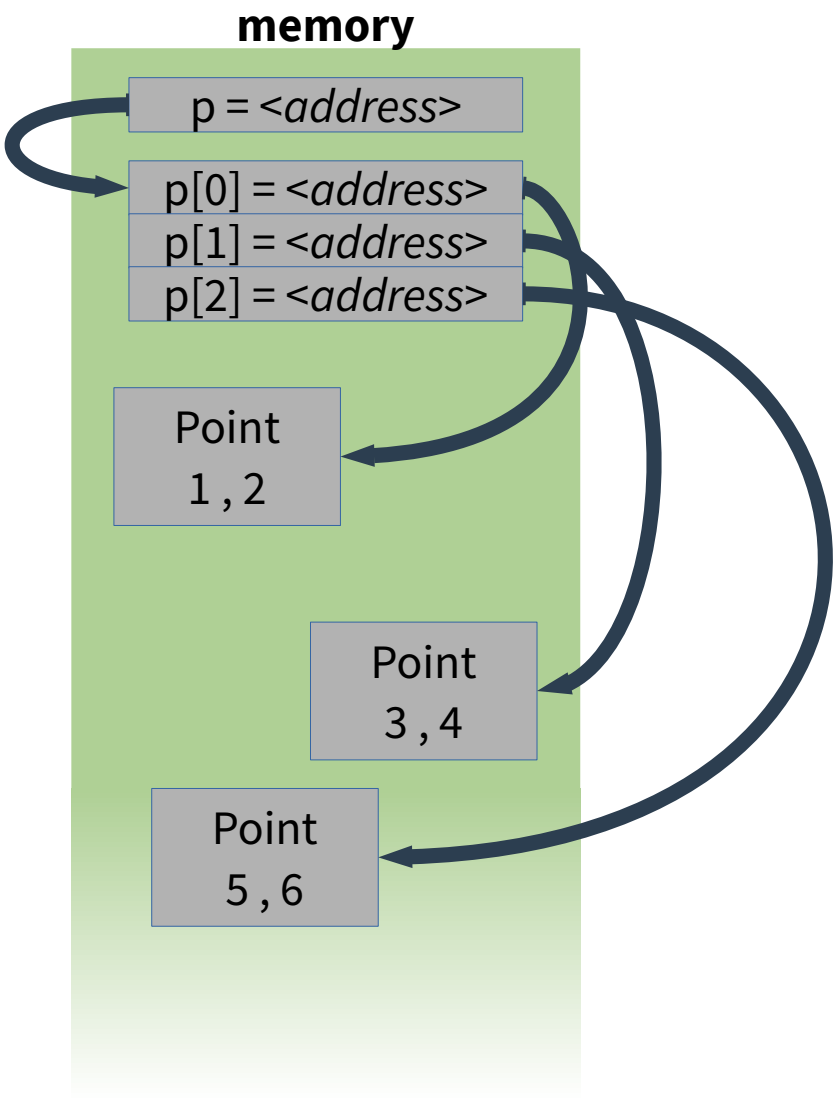
```
Point[] p;

p = new Point[3];

p[0] = new Point(1,2);

p[1] = new Point(3,4);

p[2] = new Point(5,6);
```



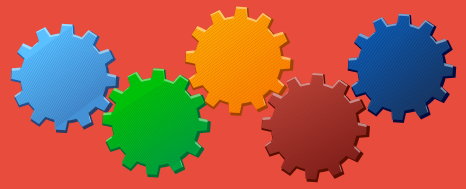
Arrays of Objects in Memory

```
Point[] p;

p = new Point[3];

p[0] = new Point(1,2);

p[1] = new Point(3,4);

p[2] = new Point(5,6);
```

instruction pointer

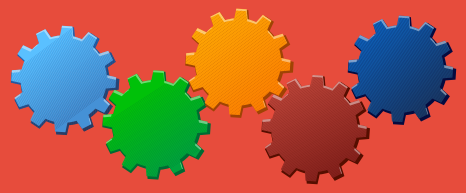


Practice Problems

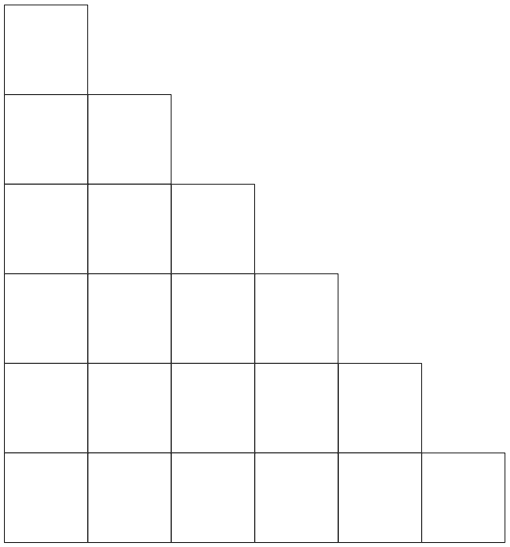


- Sum every third value in an array
- Find the maximum value in an array
- Find the index of the maximum value in an array
- Reverse the elements of an array
- Merge two sorted arrays efficiently (i.e. with a single pass)

Practice Problems



1. Create a triangular array. Number of rows is given by user input

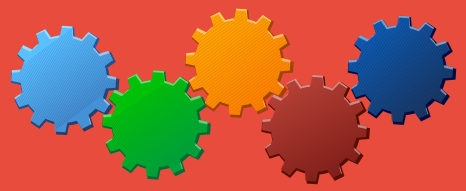


2. Double the size of row X

3. Modify the above code to also copy data into the resized row

4. Can you make it a 3-dimensional array with depth 2?

Practice Problems



- Write the method **`breakup`** that transforms a 1D array into a 2D array where each row contains the maximum number of items, from left to right, that their sum doesn't exceed a certain value. Examples:

```
breakup({1,0,0,0,1,1,1,0,1,0,0},2)     →   {{1,0,0,0,1},{1,1,0},{1,0,0}}

breakup({0,1,0,1,1,1,0,1,0,1,1},3)     →   {{0,1,0,1,1},{1,0,1,0,1},{1}}
```

- Write the method **`reshape`** that transforms a 2D array into a lower triangular 2D array that preserves the order of the items. Keep in mind that the last row might be incomplete. The array must be modified *in place*. Examples:

```
reshape({{1,2,3,4,5},{6,7,8,9,10}})    →   {{1},{2,3},{4,5,6},{7,8,9,10}}

reshape({{1,2},{3,4},{5,6},{7,8}})     →   {{1},{2,3},{4,5,6},{7,8}}
```