CS1020: DATA STRUCTURES AND ALGORITHMS I

Tutorial 1 – Simple OO with Java

(Week 3, starting 25 January 2016)

1. *Variables & Message Passing*

Explain the behavior of the following three code fragments. Specifically, **why some methods are able to swap** the desired `int` values passed into the argument (on the caller end), while others are **not able to**.

Tip: Try it out! Create a program in vim, paste the fragment in the right place. Compile and run in sunfire.

```
// Are the int values within a and b swapped?
public static void swap1(int a, int b) {
      int temp = a;
      a = b;
      b = temp;
}
```

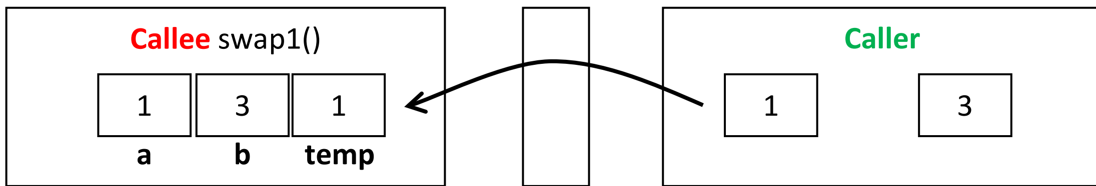
```
class MyInteger {
      public int x;
      public MyInteger(int n) {
            x = n;
      }
      // Are the int values swapped?
      public static void swap2(MyInteger a, MyInteger b) {
            int temp = a.x;
            a.x = b.x;
            b.x = temp;
      }
}
```

```
// Are the int values within a[i] and a[j] swapped?
public static void swap3(int[] a, int i, int j) {
      int temp = a[i];
      a[i] = a[j];
      a[j] = temp;
}
```

Related Concepts

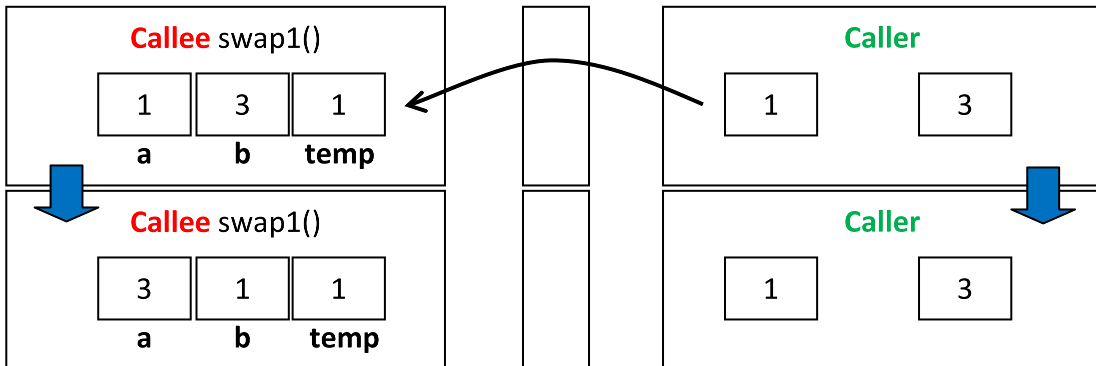
- Primitive vs reference data types
- (Caller, arguments) & (callee, params)
- Pass-by-value
- Dereferencing

Use diagrams to explain. For example, the following diagram shows the call `swap1(1, 3)` and the values of the local variables `a`, `b` and `temp` after the first statement in the method is executed. What are the values of the local variables after all the statements in the method are executed?

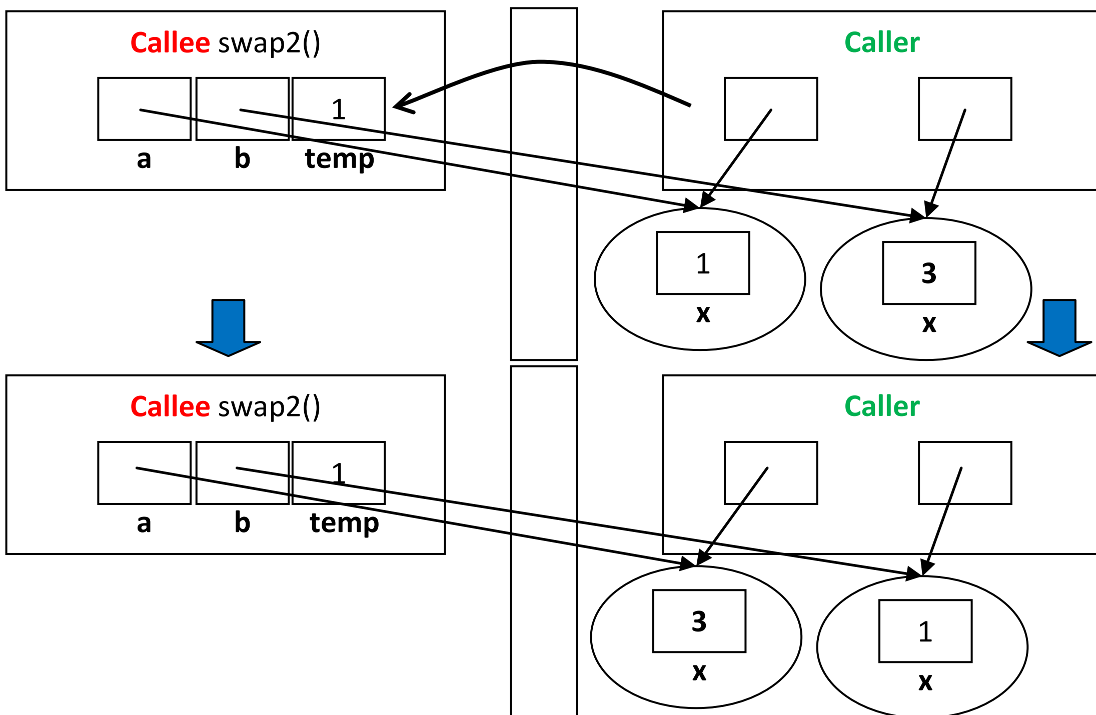


Answer

On the caller's end, only the values passed into swap1() are not swapped, while those passed into swap2() and swap3() are swapped. Java uses **pass-by-value** for method calls. **Argument** values (**caller** end) are **copied** to **parameters** (**callee** end). Here, we show `swap1(1, 3)` being invoked from `main()` method.

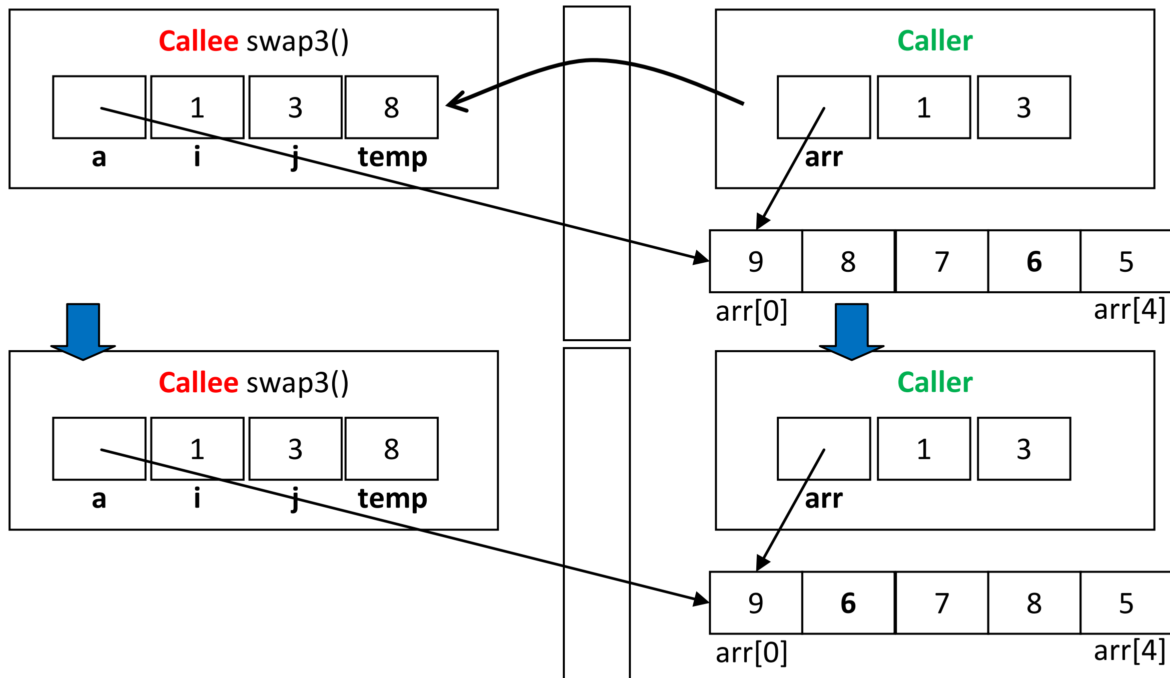


swap1() shows how **primitive** variables are stored. In swap2() and swap3(), we have variables of a **reference data type**, the value of which is a memory address of an object. Therefore, the value that is copied from argument to parameter is the **address of the SAME object**. Here, we show `swap2(new MyInteger(1), new MyInteger(3))` being invoked from the `main()` method.



From the diagram, it is clear that the values 1 and 3 in swap2() will be interchanged on both ends. Why are objects not stored directly in a variable? This is because an object can be arbitrarily large. Thus, copying the entire object unnecessarily may not be efficient (imagine a 4GB object being copied repeatedly).

Similarly in swap3(), since an array can be very long, the value copied is the starting address of the array, i.e. arr[0]. Hence, the value the value of arr[i] and arr[j] are swapped on both ends. Here, we show `int[] arr = {9, 8, 7, 6, 5}; swap3(arr, 1, 3)` being invoked from `main()` method.



Now what exactly does `a.x` in swap2(), and `a[i]` in swap3(), mean? In swap2(), `a` stores the memory location of a MyInteger object. `a` is not the object itself. (`a.`) **moves to the object**, and `a.x` reads the member variable `x` of THAT object.

In swap3(), `a` stores the memory location of an object, a 1-dimensional int array (`int[]`). `a` is not the array object itself. (`a[]`) **moves to the array object**, and `a[i]` reads the integer element that is `i` spaces away.

In subsequent tutorials, you are expected to **draw diagrams** to help yourself understand **what happens in memory**, just as in Q1 and Q2.

2. *Arrays*

(a) Draw out what each of these 4 statements does in memory:

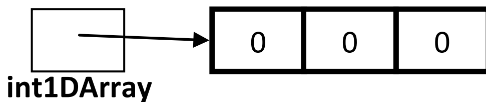
```
int[] int1DArray = new int[3];
int[][] int2DInitArray = new int[3][5];
int[][] int2DPartialArray = new int[3][];
Point[] pointArray = new Point[3];
```

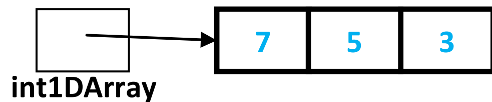
(b) After creating these 4 arrays, each array is then used to hold zero or more elements. If we then examine the memory for **each array**, what is the **minimum** and **maximum** number of objects that could possibly be present?

For example, the `pointArray` variable could have 1 to 4 objects present. When the array is first created, it is the only object being referred to by the `pointArray` reference. It could also hold 3 more objects.

Answer

```
int[] int1DArray = new int[3];
```



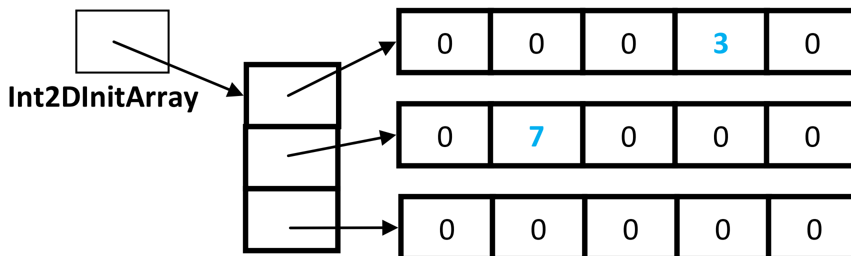


Minimum: 1 object

Maximum: 1 object

The array reference points to an array object. Within the array object, each element is a primitive `int`. Even if we **store** some `int` values in the array, no other objects are created.

```
int[][] int2DInitArray = new int[3][5];
```

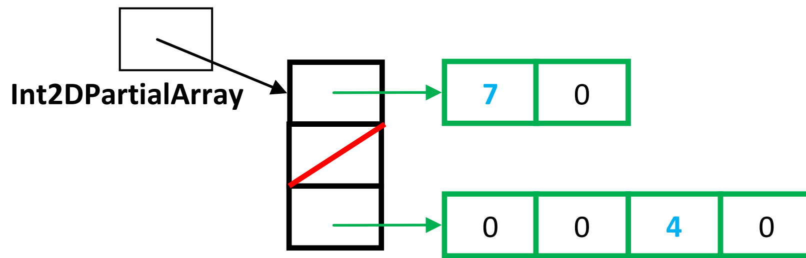


Minimum: 4 objects

Maximum: 4 objects

The array reference points to an array object of length 3, which in turn points to an array object of length 5. Again, since `int` is a primitive type, no other objects are created.

```
int[][] int2DPartialArray = new int[3][];
```

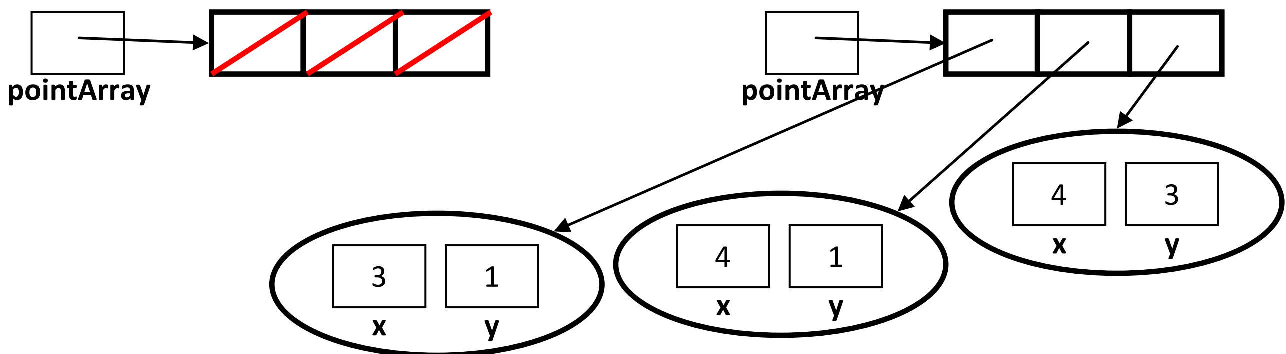


Minimum: 1 object

Maximum: 4 objects

The array reference points to an array object of length 3. Each element in the first dimension does NOT point to anything, i.e. a `null` reference. Therefore, there is only 1 object at the start. We can then **create arrays** of different lengths for the second dimension, which **store** `int` elements.

```
Point[] pointArray = new Point[3];
```



Minimum: 1 object

Maximum: 4 objects

As explained in the question.

3. *Objects*

The `java.awt.Point` class was mentioned in lecture 3. View the API documentation at:

https://docs.oracle.com/javase/7/docs/api/java/awt/Point.html

Complete the following method to "draw" a few identical diamonds at different points on the Cartesian plane. Each diamond is made up of an array of Points.

You are provided with the height and width of each diamond, which is a positive even integer. You are also provided with the centres of where the diamonds are to be drawn, in a 1D `Point` array. The method returns an array of diamonds, i.e. a 2D `Point` array.

```
public static Point[][] drawDiamonds(int width, int ht, Point[] ctrs) {
     /* TODO: create diamonds and return array of diamonds */
}
```

Answer

Remember, understand the requirements and try out a few possible cases first. Draw out how the input and output should look like in memory.

Then, design your algorithm. Think of the steps that can be repeated. When should they repeat until? What should be done before and after the repeatable part of the problem?

```
public static Point[][] drawDiamonds(int width, int ht, Point[] ctrs) {
     Point[][] diamonds = new Point[ctrs.length][];
     for (int idx = 0; idx < ctrs.length; idx++) {
          int ctrX = (int) ctrs[idx].getX(); // or ctrs[idx].x;
          int ctrY = (int) ctrs[idx].getY(); // or ctrs[idx].y;
          Point[] currDiamond = new Point[4];
          currDiamond[0] = new Point(ctrX, ctrY + ht/2); // top
          currDiamond[1] = new Point(ctrX – width/2, ctrY); // left
          currDiamond[2] = new Point(ctrX, ctrY – ht/2); // bottom
          currDiamond[3] = new Point(ctrX + width/2, ctrY); // right
          diamonds[idx] = currDiamond;
     }
     return diamonds;
}
```

A picture (diagram) speaks a thousand words. Don't forget to draw objects in memory and references to them!

Note: If you have any queries on tutorials, please post on the "Tutorial" forum in IVLE.

- Hope you had fun, prepare well for tutorial 2 ☻ -

Revise lecture material
Draw diagrams, code
Attempt tutorials
Test your solution