CS1020: Data Structures and Algorithms I

Tutorial 1 – Simple OO with Java

(Week 3, starting 23 January 2017)

1. *Variables & Message Passing*

In the following three code fragments, **some methods are able to swap** the desired `int` values passed into the argument (on the caller end), while others are **not able to**. Use diagrams to illustrate why this is so.

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;
}
```

Related Concepts

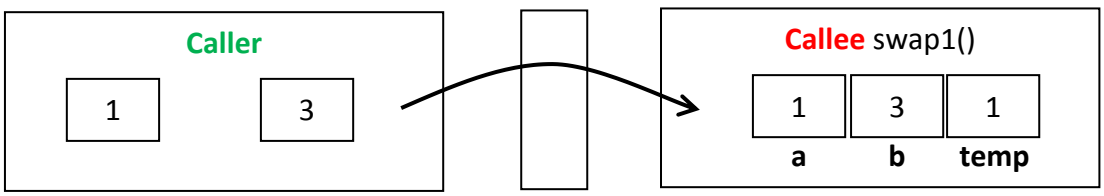
- Primitive vs reference data types
- (Caller, arguments) & (callee, params)
- Pass-by-value
- Dereferencing

```
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;
}
```

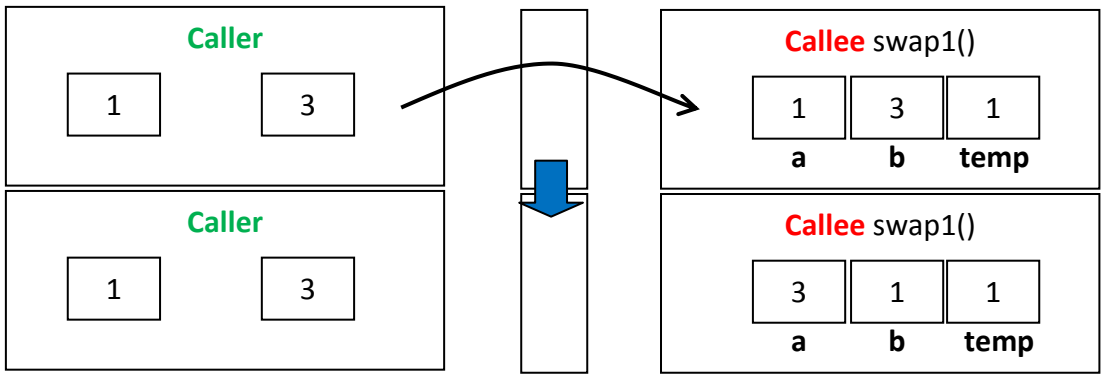
Example

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, parameters, and arguments, after all the statements in the method are executed?



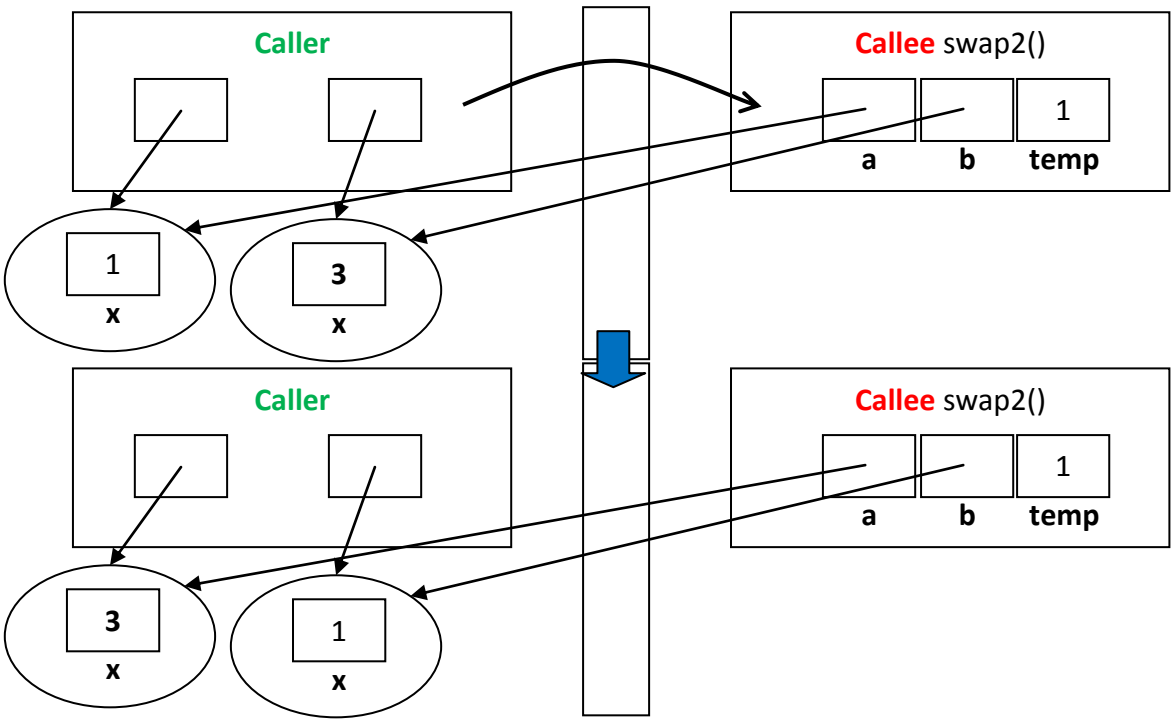
Answer

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.



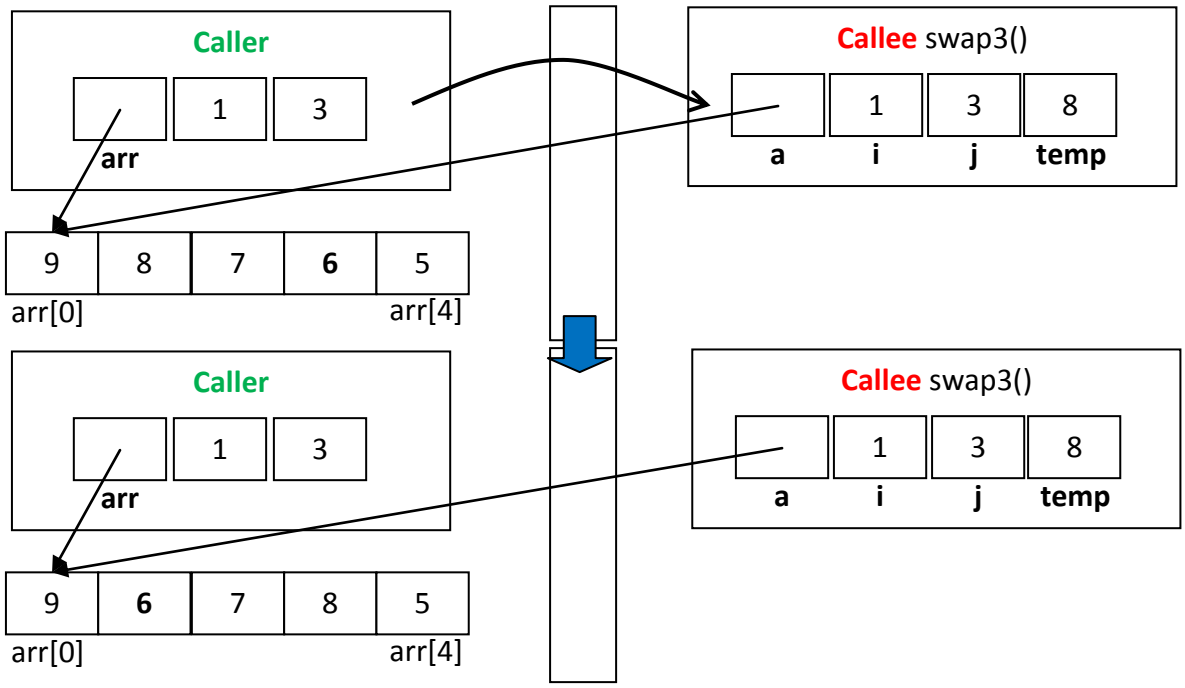
swap1() shows how **primitive** variables are stored. The **value** of the variable is **stored directly** in the location meant for the variable.

In swap2() and swap3(), we have variables of a **reference data type**, each containing either the memory address of an object or null (i.e. not pointing to any 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. Copying the entire object unnecessarily may be very inefficient. Imagine a 4GB object being copied repeatedly...

Similarly in swap3(), **ALL arrays** in Java are **reference data types**, even for arrays of primitives. Since an array can be very long, the value copied is the **starting address** of the array, i.e. the memory location at which `arr[0]` is. Hence, 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? Both the . and [] operators perform de-referencing. Conceptually, in:

- swap2()
  `a` stores the memory location of a MyInteger object; `a` is not the object itself
  (`a.`) **moves to the object**
  `a.x` reads the member variable `x` of THAT object

- 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**
  `a[i]` reads the integer element that is `i` spaces away

In subsequent tutorials, you are expected to **draw diagrams on your own**, 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];
```

`Point` here refers to the `java.awt.Point` class.

Related Concepts

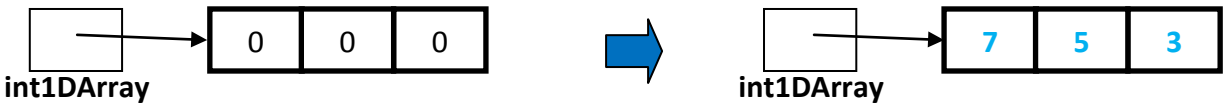
- All Java arrays are reference types
- Multidimensional arrays
- References vs objects

(b) After creating these arrays through the 4 statements, each array is then used to hold zero or more elements. If we then examine the memory for **each case**, 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 up to 3 objects.

Answer

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

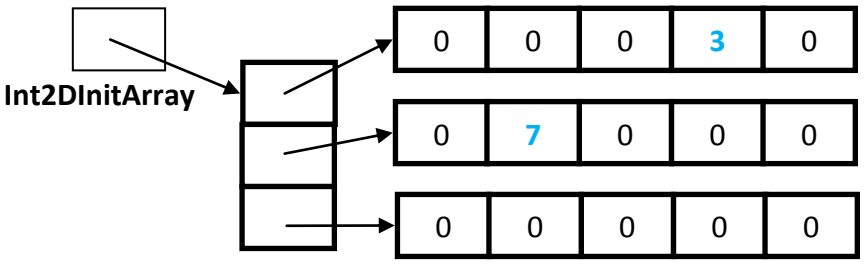


Minimum: 1 object

Maximum: 1 object

The array reference points to an array object. The **reference** itself is **NOT an 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];
```

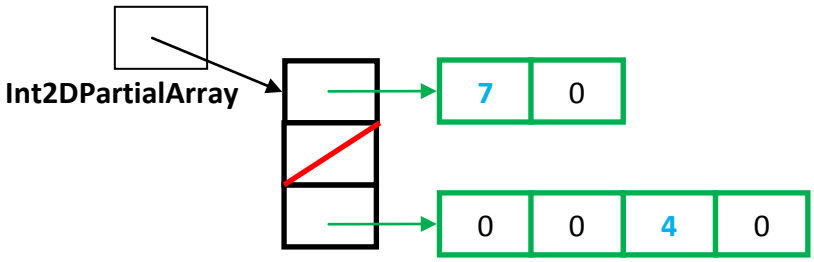


Minimum: 4 objects

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][];
```

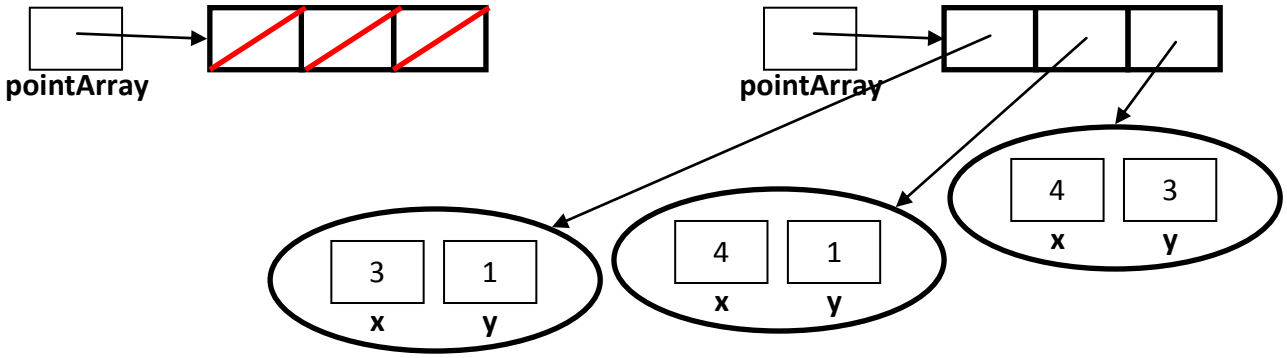


Minimum: 1 object Maximum: 4 objects

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 choose to **create arrays** of different lengths for the second dimension, which **store** `int` elements.

The above illustration shows a case in which there are 3 objects.

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



Minimum: 1 object Maximum: 4 objects

This case has already been explained in the question.

3. *Simple OOP in Java*

You want to print out the lyrics of this song[1], to teach (or confuse =P) kids about the sounds animals make:

Dog goes **woof**
Cat goes **meow**
Bird goes **tweet**
Mouse goes **squeak**
Cow goes **moo**

Related Concepts

- Class vs object
- Member variables & encapsulation
- Access modifiers, static, final
- `this` keyword

The lyrics can be generalized for different animals, each having a different *name* and **sound**. With knowledge of object-oriented programming, you want to demonstrate that it is possible to write a program that *displays* the song. To show that your program works, add the 5 animals above and test your program. Use the following skeleton to solve the problem:

```
class Animal {
   /* TODO: Implement data and functionality of an Animal here */
}
public class Song {

   private Animal[] _animals;
   private static final int ANIMAL_COUNT = 5;

   public Song() {
      /* TODO: Create your animal array here */
   }

   public void display() {
      for (int i = 0; i < ANIMAL_COUNT; i++)
         System.out.println( /* TODO: Add the lyrics here... */ );
   }

   public static void main(String[] args) {
      (new Song()).display();
   }
}
```

[1] Adapted from "The Fox" by Ylvis, 2013

Answer

Animal has 2 attributes: name and sound. These are modelled as String instance variables (aka fields) – EACH Animal object has references to two Strings, containing the name and sound. The data is **encapsulated** by declaring them with the `private` **access modifier**. This means, outside the animal class, we **cannot directly** access or modify any Animal object's fields.

`getName()` and `getSound()` are **instance methods** serving as accessors. As we do not need to modify an animal's attributes, there are no mutators here. `getName()` returns the value of the `_name` instance variable **belonging to ONE Animal object**, the one pointed to by the `this` reference. Therefore, invoking `getName()` on different Animal objects may return references to different names.

```
class Animal {
   private String _name; // e.g. Cow
   private String _sound; // e.g. moo
   public Animal(String name, String sound) {
      _name = name;
      _sound = sound;
   }
   public String getName() { return _name; }
   public String getSound() { return _sound; }
}
```

Now we can complete the Song class, which uses the functionalities specified in the Animal class, by invoking the instance methods of *different* animal objects. A constant has the keywords `static final`:

- `static`: Exists **only 1 copy** of the variable, **belonging to the Song class**, instead of each object
- `final`: Once the variable is initialized, a value **cannot be subsequently reassigned** to it

```
public class Song {

   private Animal[] _animals;
   private static final int ANIMAL_COUNT = 5;

   public Song() {
      _animals = new Animal[ANIMAL_COUNT];
      _animals[0] = new Animal("Dog", "woof");
      _animals[1] = new Animal("Cat", "meow");
      _animals[2] = new Animal("Bird", "tweet");
      _animals[3] = new Animal("Mouse", "squeak");
      _animals[4] = new Animal("Cow", "moo");
   }

   public void display() {
      for (int i = 0; i < ANIMAL_COUNT; i++)
         System.out.println(
            _animals[i].getName() + " goes " + _animals[i].getSound()
      );
   }
...
```

`_animals[2].getName()` is equivalent to `this._animals[2].getName()`. First, `this._animals` refers to the current Song object's `_animals` field, which points to an Animal array of size 5. Therefore, `_animals[2]` moves to the array, two elements away from the start. That element is another reference, to an Animal object. (`_animals[2].`) moves to that Animal object (Bird).

We then read the name of the Animal object (Bird) using the `getName()` instance method. Remember, that method does not return a String object itself, but a reference to the String "Bird".

As we have mentioned, **the first 100 times** you encounter some strange code, **draw out what the code does in memory**…! Don't view code as mere text.

Before & After

As you are attempting each tutorial question, ask yourself what the related concepts learnt in lectures are. If you are unsure of an answer, revise the concept, try reasoning and writing out your answer again, and then test your answer by coding in vim!

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

Before all tutorial groups have finished discussing this tutorial, you may clarify your doubts on the questions, but please do NOT post your answers. After the answers document has been uploaded, you may clarify your answers, even on the forum. Do NOT let your doubts accumulate!

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

Draw diagrams
Attempt tutorials
Test your solution