UNIVERSITÉ DU LUXEMBOURG

Draw the memory

Programming Fundamentals 2

Goals

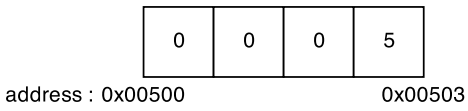
- ✯ Understand the memory representation of Java objects.
- ✯ Introduction to stack and heap memory.

Memory in Java

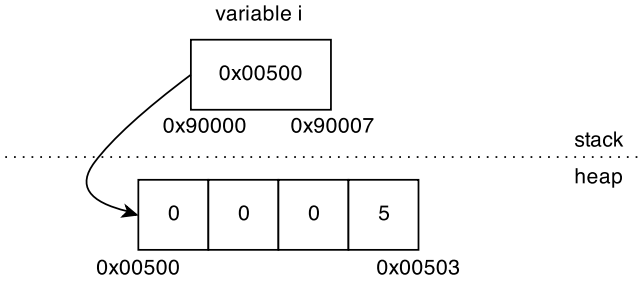
The memory is organized linearly, and you ask a block of adjacent memory through the operator `new`. We use the following class as an example:

```
public class MyInteger {
  private int x;
  public MyInteger(int x) { this.x = x; }
}
```

What does happen in memory when we execute `MyInteger i = new MyInteger(5);`? First, we reserve a memory zone of sufficient size to contain the attributes of the object, here an integer coded on 4 bytes:



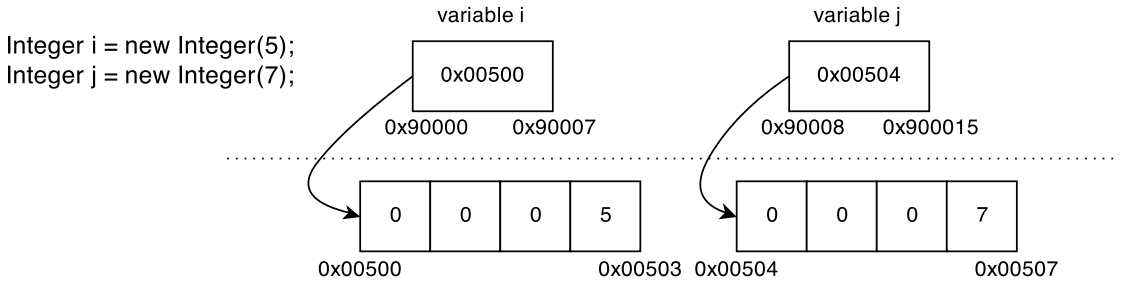
But that's not all, in Java, every variable containing an object use an *indirection*, which means that the variable `i` contains the address of the allocated memory zone, so we have:

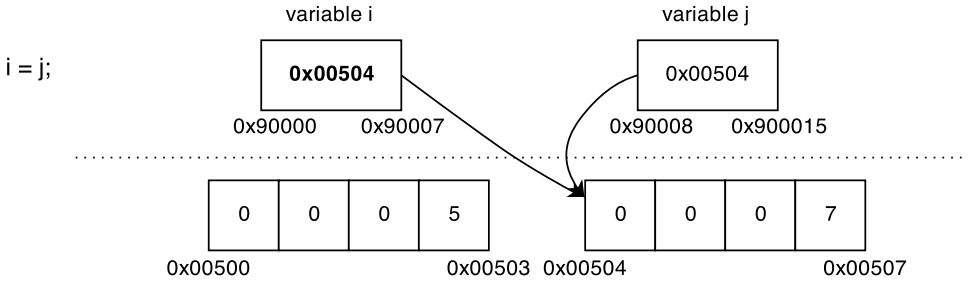


We note that memory is divided into two: the *stack* on one side, and the *heap* on the other side. Every variable you declare in your program is stored onto the stack. However, a stack variable can contain an address referring to a memory zone in the heap. You must remember, that in a program, we only access to the heap through a "stack variable" containing an address.
Suppose we have the following code:

```
MyInteger i = new MyInteger(5);
MyInteger j = new MyInteger(7);
i = j;
```

What does happen in memory? As we can observe on the next diagram, the variable i refers to the same memory zone than j, which means we can modify a same object through two variables:





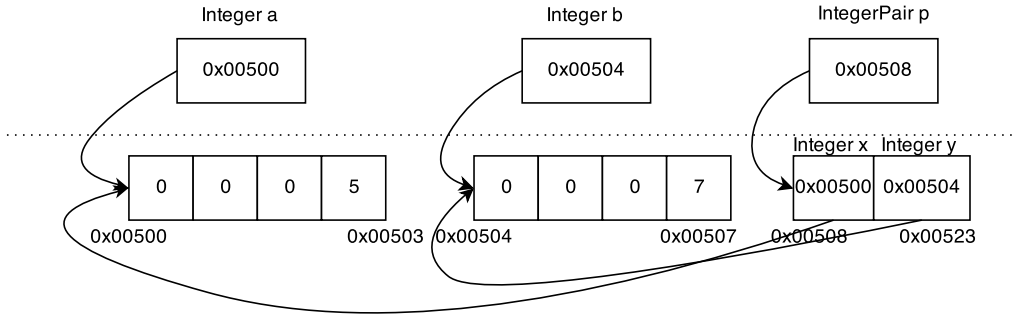
The memory zone initially pointed by i is now inaccessible, we can never use it again and the *garbage collector* will clean this zone and make it accessible again later.

Object's attributes can also point to other objects, consider the following code:

```
public class IntegerPair {
  private MyInteger x;
  private MyInteger y;
  public IntegerPair(MyInteger x, MyInteger y) {
    this.x = x;
    this.y = y;
  }
}

MyInteger a = new MyInteger(7);
MyInteger b = new MyInteger(5);
IntegerPair p = new IntegerPair(a, b);
```

We represent the memory of this object in the next diagram. Notice that the attributes x and y points to the same location than a and b.



Finally, we must distinguish between primitive types (`int`, `double`, `char`, ...) and objects (`IntegerPair`, `String`, `ArrayList`, ...) because primitive types do not request heap memory, but are automatically allocated on the stack. Consider `int i=9; int j=2; j=i;`, the value of i is copied in j. Therefore, we will obtain two distinct elements i and j, and modifying one will not change the other. If we declared them as `MyInteger`, then the address of the object would be copied, but not the value pointed by that address[1] Note that for *copying object*, you must use the method `clone`, which must be manually implemented for the corresponding object.

If you need more explanations about stack and heap, check out this Youtube video which explains it very nicely: `https://www.youtube.com/watch?v=ckYwv4_Qtmo`. It is worth taking the time needed to understand stack and heap memory, because it is a useful general concept.

[1]To confuse you a little bit more, there is an `Integer` class in the Java library, however this class is different from ours. Unlike ours, the Java `Integer` class is immutable and therefore will behave "like a primitive type", so it is impossible to modify `i` or `j`, and whether a copy occurs when executing `i=j` is irrelevant.