References

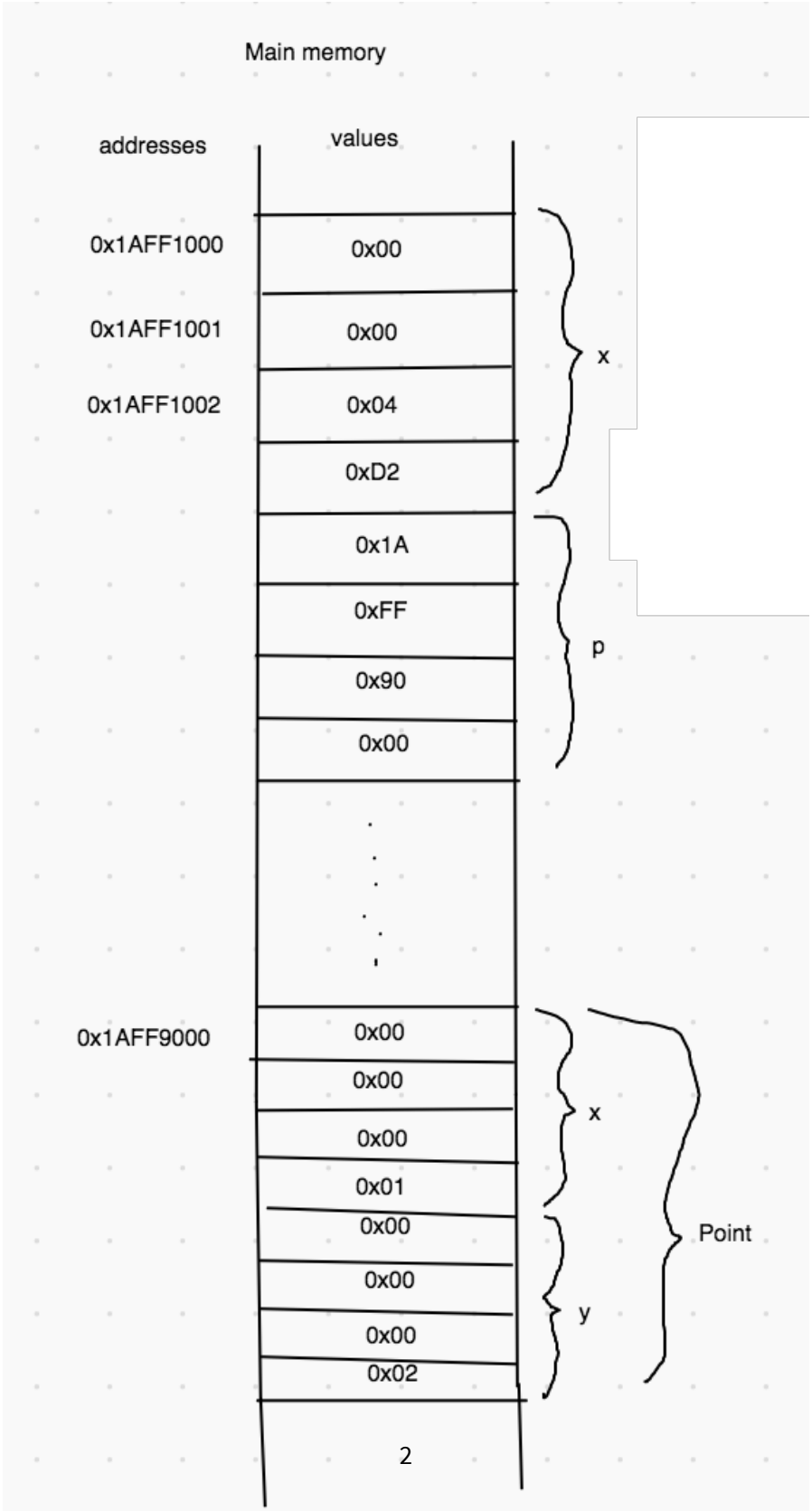
Primitive types in Java are the following, with their respective size:

int	4 bytes
short	2 bytes
long	8 bytes
double	16 bytes
float	8 bytes
byte	1 byte
char	2 bytes
boolean	? bytes

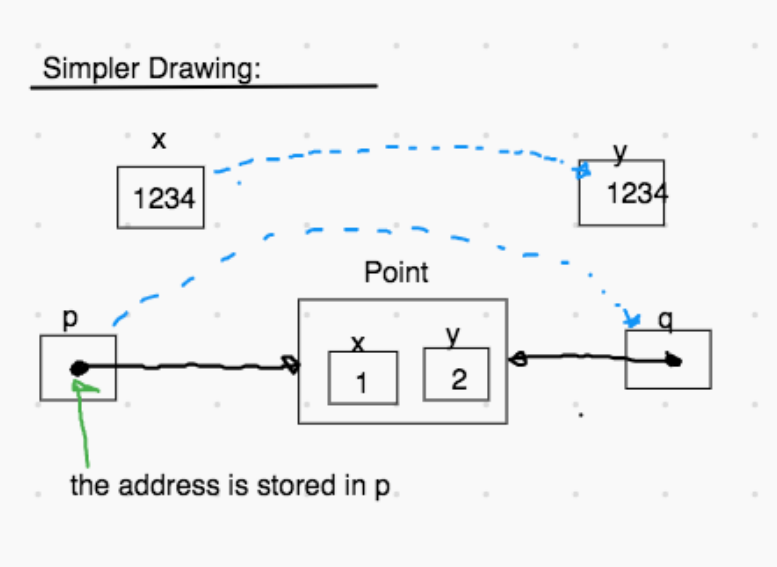
All other types on call *non-primitives*, and they are usually objects. For example, `String`, `Date`, `Stack`, and `Scanner`. In this section we will make use of the following simple class:

```
class Point {

    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // getters and setters

    public double dist(Point p) {
        return Math.sqrt(Math.pow(p.x - x, 2) + Math.pow(p.y - y, 2);
    }
}
```

Both types are stored in main memory, where the running program stores variables, objects, parameters, etc… Primitives are stored directly in memory at a specific address. Non-primitive variables store the address (memory location) of the object in main memory, not the object itself.


Main memory
addresses
values
0x1AFF1000
0x00
0x1AFF1001
0x00
0x1AFF1002
0x04
0xD2
x
0x1A
0xFF
0x90
0x00
p
0x1AFF9000
0x00
0x00
0x00
0x01
0x00
0x00
0x00
0x02
x
y
Point

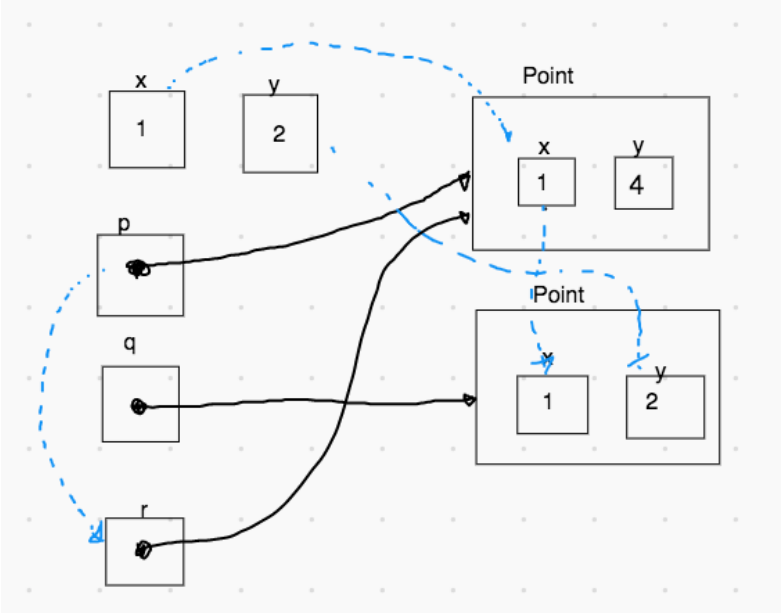
Instead of drawing memory in full detail we will use the follow short-hand:



Aliases

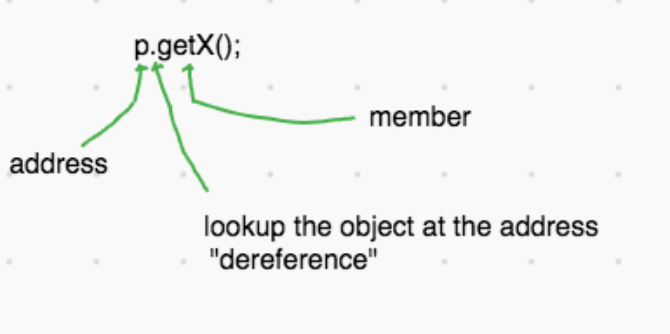
How does assigment work (= operator). The same for both primitives and non-primitives: it copies the binary value stored from one memory location to another. So for, non-primitive type, assignment creates an ”alias”. Ex:

```
int x = 1;
int y = 2;
Point p = new Point(x, x+1);
Point q = new Point(p.getX(), y);
Point r = p;
r.setY(4);
```



Dereferencing

Non-primitives variables access their members by using the `.`-operator called *dereferencing*.

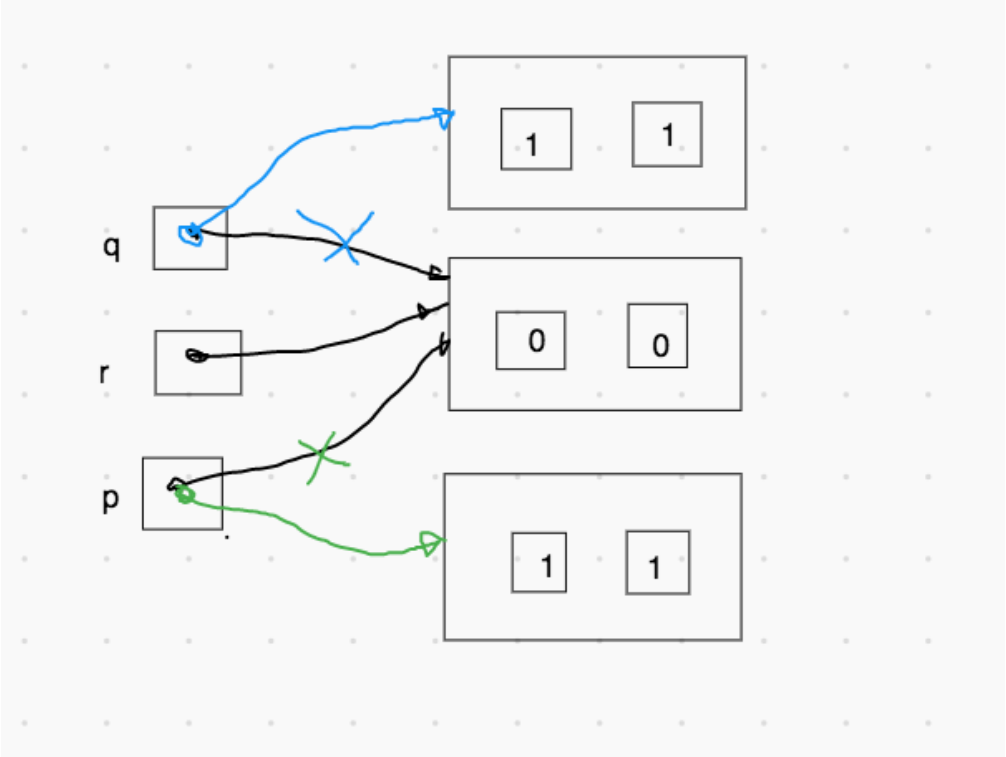


You can think of it as ”following the arrow”.

Comparaison operators: == and ≠

Operators == (and ≠) compare primitives as values and the non-primitives are compared as addresses. So `p == q` is true of `p` and `q` are the same object.

```
Point p = new Point(0,0);
Point q = p;
Point r = q;
r == p ? true
q = new Point(1, 1);
q == p ? false
r == p ? true
p = new Point(1, 1);
p == q ? false
p.equals(q) ? true
p == r <-- is this an error???? no it's fine
```

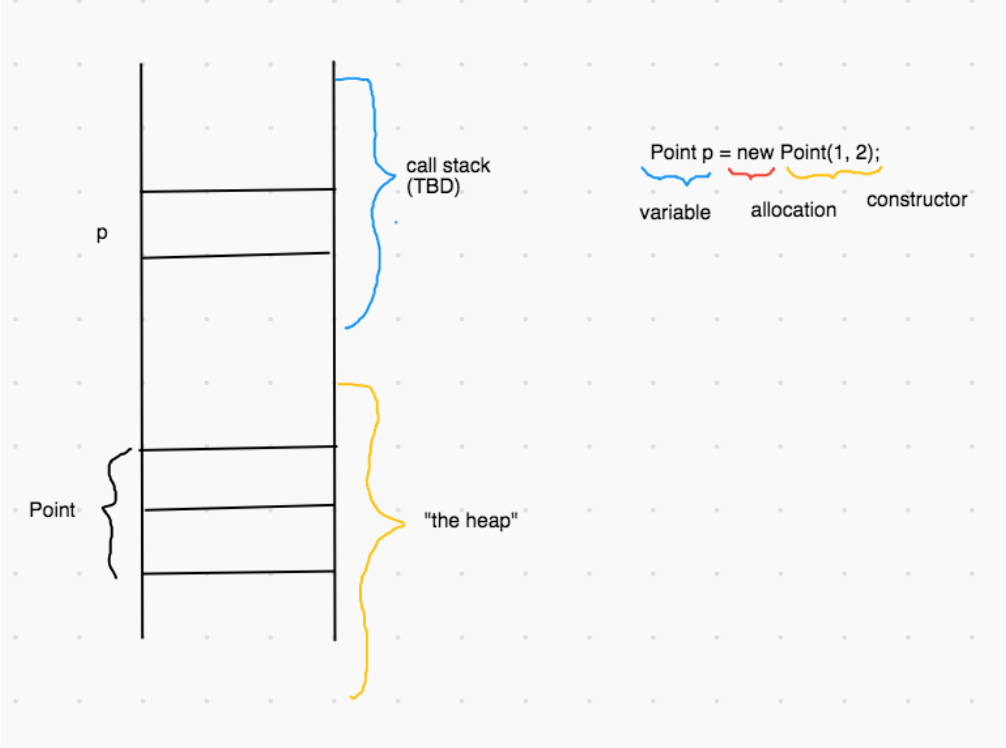


The idea is that `p == q` is what we call *identity*, whereas a call to `p.equals(q)` is *equality*.

Allocation and deallocation

In this statement, `Point p = new Point(1, 2);`:

- The ”new” gives us space in memory (in the heap) to store the object. The amount of space corresponds to the fields of the object.
- The call to the constructor initalizes this space!



How do we give back space? When there are no references to an object, the ”garbage collector” will eventually reclaim it. When does this happen?

Some ways an object can be abandoned:

- When a variable is reassigned:

```
Point p = new Point(1, 2);
Point q = new Point(3, 4);
p = q;  // the Point(1, 2) can be collected after this statement.
```

- When a variable goes out of scope:

```
public void foo() {
   Point p = new Point(1, 2);
   ...
}  //  the Point(1, 2) can be collected at the end of this method
```

- When a variable is nullified (next section)

null

What is null? Null is the absence of an object. We can think of it as a bad address. In C++ it's usually 0x000000. Setting a variable to null either removes an alias or abandons an object:

```
Point p = new Point(1,2);
...
p = null; // nullify 'p' --> the point (1,2) can be reclaimed
```

We can use `null` with our comparaison, ex: `if(p == null)`.

Dereferencing a null will throw a `NullPointerException`

```
Point p = null;
sout(p.getX());
      ^
      dereferencing a null
```

Primitive wrapper classes

Each primitive (`int`, etc…) has an object version, `Integer`, etc… Why? It's a reference type version useful in generics, interfaces.

```
public class Integer implements Comparable<Integer> {

    private int value;  // wrap the primitive in a class

    public Integer(int value) {
       this.value = value;
    }

    public int valueOf() {
        return value;
    }

    public int compareTo(int x) {
        return value - x;
```

```
    }

}
```

Java auto-wraps and unwraps the primitives when needed.

```
int x = 123;
Integer i = x;   // compiler converts to "Integer i = new Integer(x);"
int y = i;       // compiler converts to "int y = i.valueOf();"
```