Allocating Memory to Variables

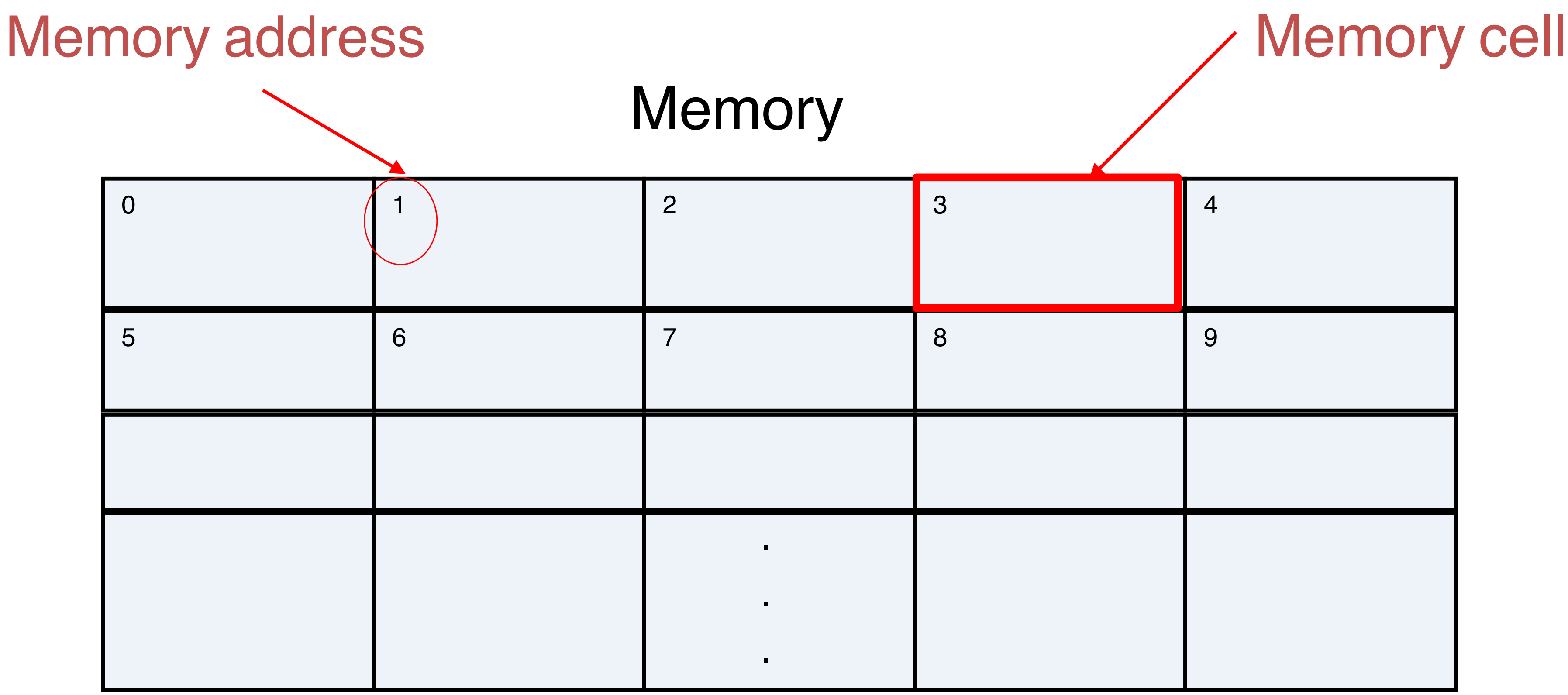
Consider the following fragment of java code

```
int a;
int b;
```

When this program is compiled memory is allocated to both variables. Since a and b are of type int and this is a primitive type in Java, the amount of memory allocated to a is large enough to store any value given to this variable; similarly for the memory allocated to b.

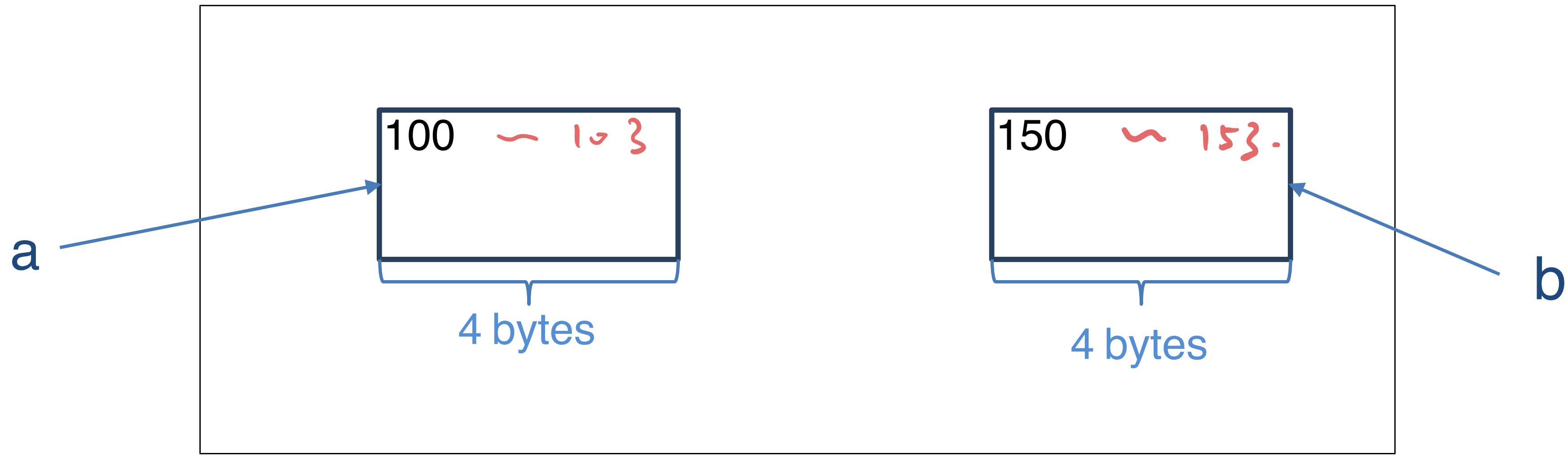
Allocating Memory to Variables

We can think of the memory of the computer as a group of cells where we can store values. Each cell has a unique address that can be used to access it. Each cell, for example, might consist of 1 byte (or 8 bits).



Allocating Memory to Variables

For example, if a is allocated to address 100 and b is allocated to address 150, the computer's memory will look like this:



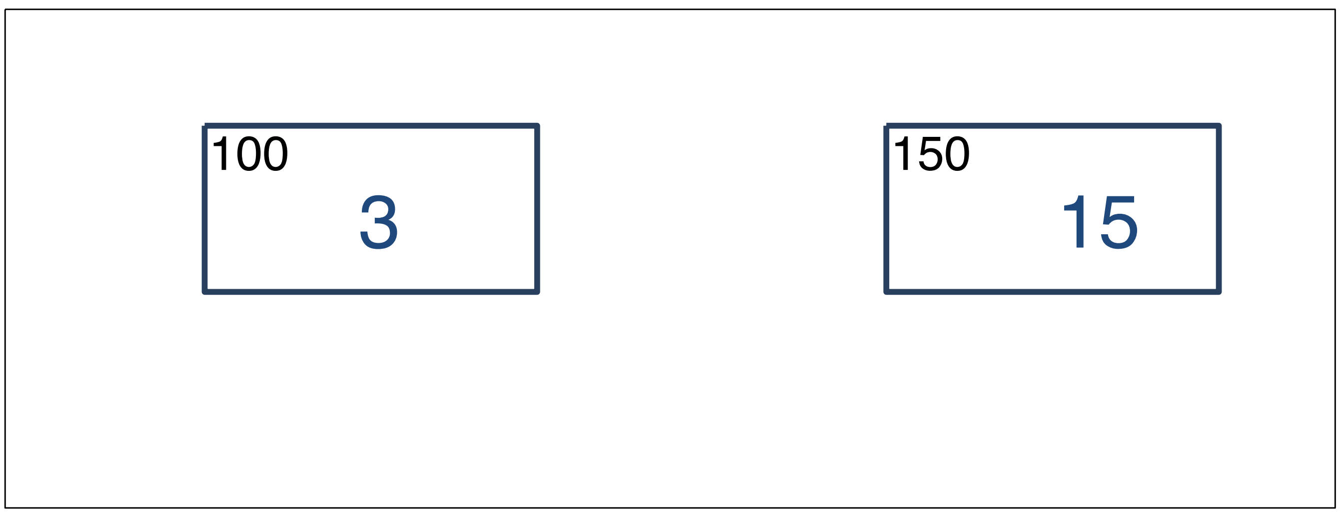
a and b are assigned each a block of 4 bytes because in Java an int has a size of 4 bytes. The first byte allocated to a is in address 100, the second one in address 101, and so on.
Java keeps track of where the variables are stored in memory in a table called the *symbol table*.

Allocating Memory to Variables

If now the following code is executed:

```
a = 3;
b = 15;
```

The computer's memory will look like this:



Allocating Memory to Variables

Non-primitive variables are handled in a different manner.

Consider the following Java class representing a rectangle:

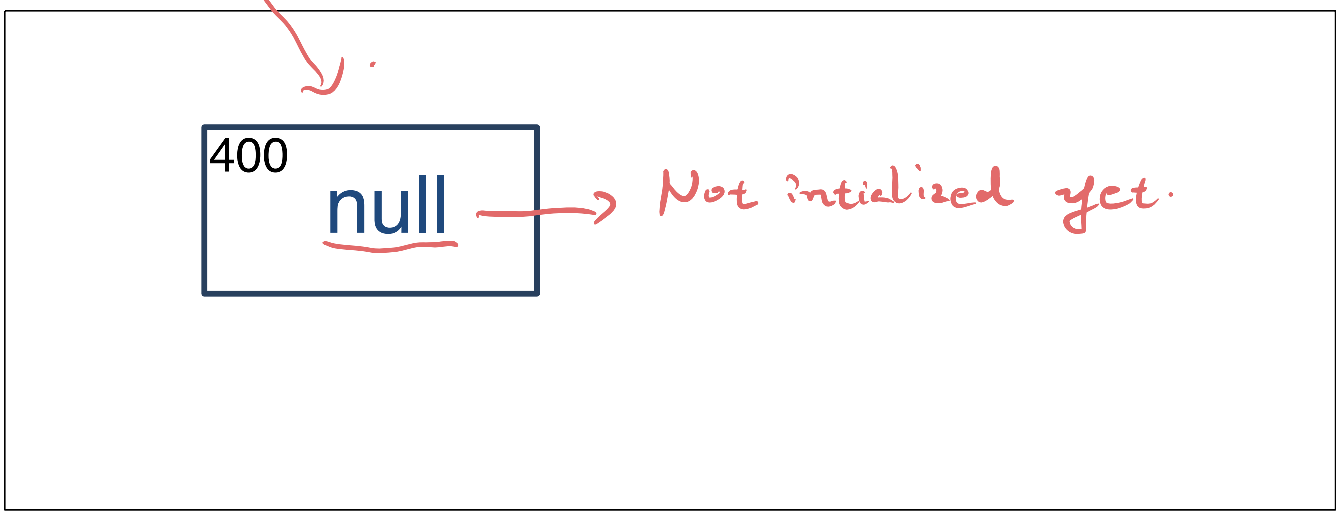
```
public class Rectangle {
        private int width, height;

        public Rectangle (int w, int h) {
                width = w;
                height = h;
        }

        public int getArea () {
                return width * height;
        }
}
```

Allocating Memory to Variables

Consider the following Java code:

```
Rectangle r;
r = new Rectangle (10,5);
```

When the declaration of r is processed (statement Rectangle r;), a block of memory is allocated to r, say starting at address 400 and large enough to store a reference to an object of class Rectangle:



Allocating Memory to Variables

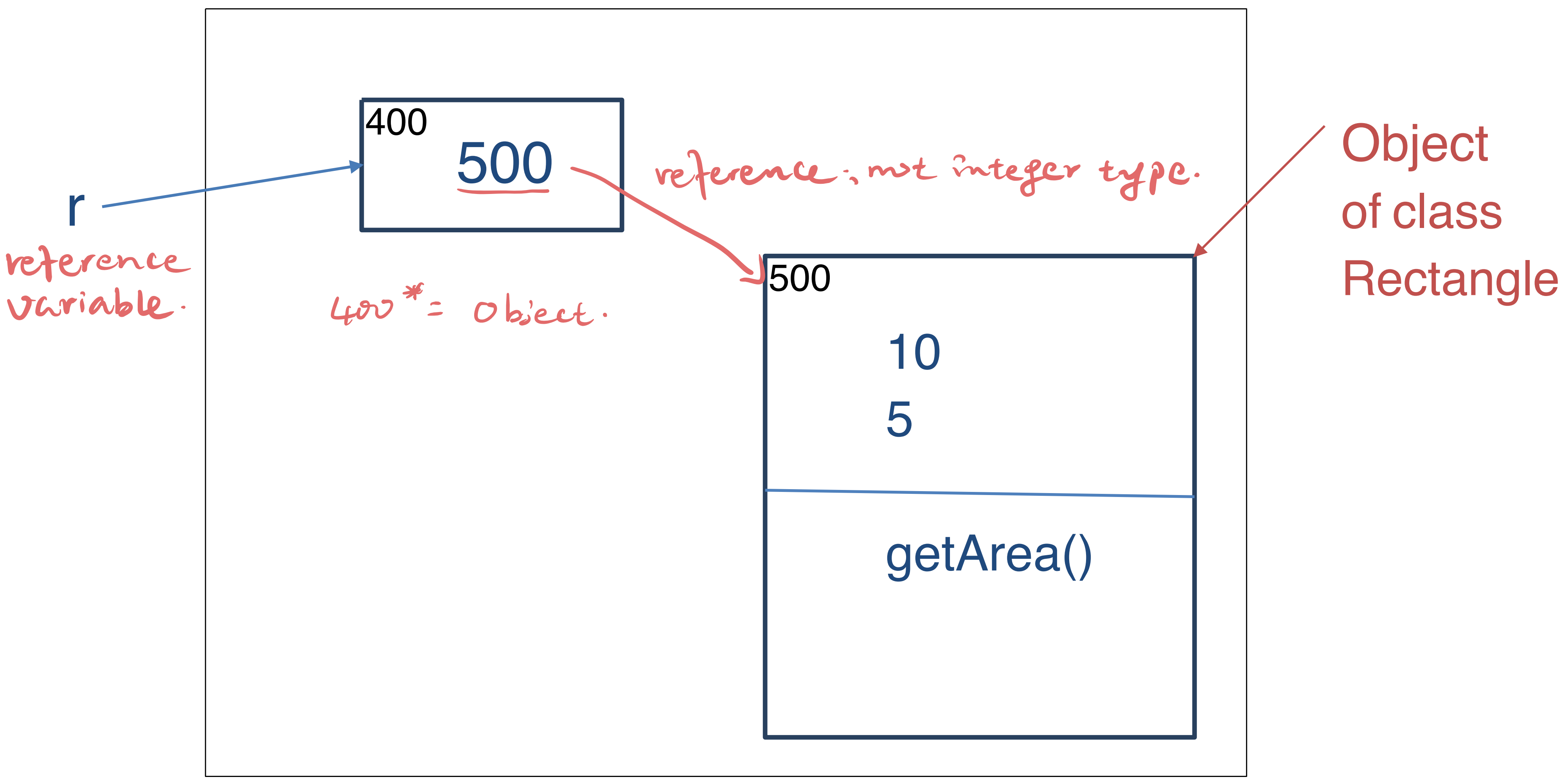
By default Java stores the value null in each non-primitive variable when it is declared. When the object is created:

```
r = new Rectangle (10,5);
```

a block of free memory large enough to store the above object of the class Rectangle (large enough to store the int values for width and height and the methods of the class Rectangle) is allocated to this object and the values 10 and 5 are stored in it. Let this block of memory start at address 500.

Note that the object is not stored in address 400, which was allocated to r. Instead in address 400 the computer stores the address 500 of the above object. The computer's memory will now look like this:

Allocating Memory to Variables



Allocating Memory to Variables

Variable r is called a *reference variable*, as it does not store an object, but a reference or an address of an object. To access the content of the object referenced by r in Java we use the dereferencing operator ".".

private private.

So, for example r.width has the value 10 and r.height has the value 5.

public.

Invoking the method r.getArea() will return the value 50.

Static variables

The term "static" in Java means class-based rather than object-based. To clarify, static variables or methods are accessed directly from a class, not from an object.

MyClass obj = new MyClass(21);

obj.getNum();

Not. static.

MyClass.add(9, 5);

obj. add (9,5).

public static class add (int, int)

instance.

getNum() is invoked on an object of MyClass (obj)

add() is invoked directly from MyClass – no objects are used

Static variables

Example: Math

Math.PI // static variable PI

Math.abs(-3) // static method abs

Math.cos(0) // static method cos

Notice that we never had to create an object of Math (i.e. Math mathObj = new Math();) to use these variables and methods.

Static variables

This is useful for data that is independent and not derived from an individual object.

Using static variables/methods reduces the amount of memory allocated than if they were non-static.

only need to allocate memory once.

Not-static: allocate memory everytime it is initialized.