SIENAcollege
Computer Science

Computer Science 210
Data Structures
Siena College
Fall 2020

Topic Notes: Java Classes

You have been writing programs that operate on data (*i.e.*, variables and parameters) that we can categorize in just two ways:

- primitive types, such as `int`, `double`, `char`
- object types, such as `String`, `Scanner`, `Random`, `DecimalFormat`

Here, we focus on these object types a bit more. In particular, we will think more carefully about introducing our own object types into our programs.

Objects and Classes

We already looked at one way to have a single entity in Java refer to multiple items: the array. Arrays are very convenient for many purposes, but they have some important restrictions:

1. all of the items in the array must be of the datatype declared in the array's declaration and construction
2. we can only refer to array elements by their subscript (or index), which must be a number from 0 to $n - 1$ for an array of length n

Among other benefits, *classes* allow us to overcome both of these restrictions.

Every object in a Java program is an entity that can contain both *fields*, or *instance variables* – which are in many ways like the local variables you've been using in your programs almost from the beginning, and *methods* (or *member methods*), which operate on the data in those fields.

The idea of an object is central to the *object-oriented programming* paradigm, which has been very popular since being introduced a few decades ago.

The idea is that we write program components, called classes, which represent templates for the *objects* we wish to represent in our program. For each object, we include fields that are used to represent the state of the object and methods that allow that state to be queried or modified.

A text I used previously includes an example of an alarm clock. They came up with a list of fields that can be used to describe the state of an alarm clock. It is similar to this list:

- the current hour (0-23)

- the current minute (0-59)
- the current second (0-59)
- the alarm hour (0-23)
- the alarm minute (0-59)
- the alarm status (on or off)

And some methods that can be used to modify the state of the alarm clock:

- set current time
- set alarm time
- disable alarm
- enable alarm
- stop currently sounding alarm

We might also consider some methods to query the current state of the alarm:

- get current time
- get alarm time
- get alarm status (on or off)

And then the alarm clock might have some other things it does "on its own" – its state changes as the time proceeds:

- increment time by 1 second
- start sounding alarm

In Java, the functionality of an object is described in a *class*. Beginning programmers in Java need to write classes as containers for our `main`, and in some cases, a few other methods. This is just a small fraction of what a Java class can be used to do.

We look at a mechanism to achieve this by a simpler example. Consider this example, which is written with all code in its `main` method.

`https://github.com/SienaCSISDataStructuresJDT/RatiosNoClass-example`

This program maintains information about ratios of integer values. We create two ratios, each represented by 2 `int` variables, and print them, modify them, and compute their decimal equivalents.

But with a class that represents a `Ratio` object, we can *encapsulate* the numbers (the numerator and denominator) into fields, and provide methods to construct (the *constructor(s)*), access (the *accessor methods*), and modify (the *mutator methods*) the fields.

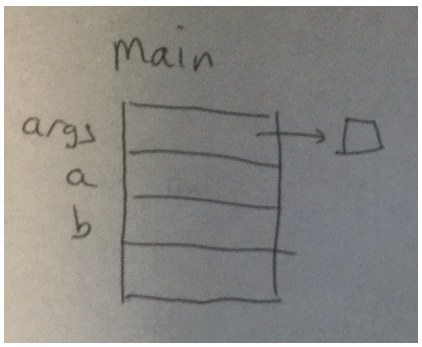
See this example and the extensive comments within for details. Start with the version in the "`basic`" directory.

`https://github.com/SienaCSISDataStructuresJDT/Ratios-example`

A Memory Diagram

It is important at this point to start thinking carefully about exactly how and where Java allocates memory for the variables in our programs. Throughout the semester, we will make *memory diagrams* of varying detail and complexity. We begin by making one for the example above.

When constructing a memory diagram for a Java application (*i.e.*, a Java program we launch by calling its `main` method), we start by allocating memory for `main`'s parameter and any local variables. We will think more carefully soon about the fact that this memory is allocated on the call stack, but for now, we'll just draw it as a box labeled with the names of any parameters and local variables for our method. In this case:



`args` is the one parameter to `main`, and that gets initialized with a reference to any command-line parameters we pass to our program. Since in this case, we aren't expecting any, we will just represent those as the empty box. The `main` method also has two local variables, `a` and `b`, each of which is a reference to a `Ratio` object. At this point in our diagram construction, these have not yet been given values. Java does not provide local variables with any default values, so we leave those boxes blank.

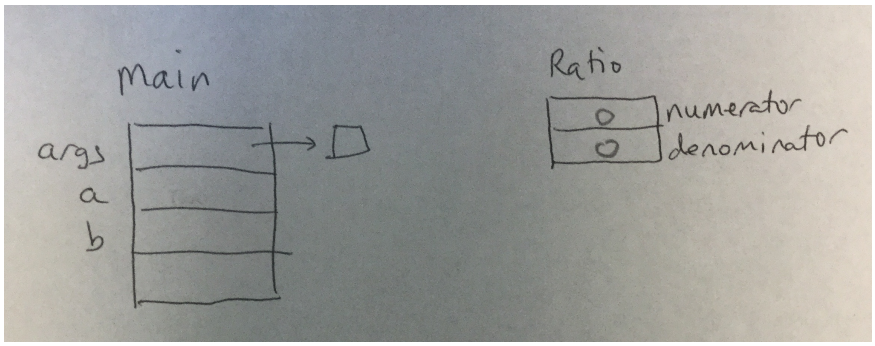
Now, we're ready to look at the actual execution of the `main` method. The first line:

```
Ratio a = new Ratio(4, 6);
```

causes many things to occur, and we will consider many of them in some detail, updating our memory diagram for each step that affects it.

That line is an object construction and an assignment of a reference to that new object to a local variable. Before anything can happen, Java needs to find the `Ratio` class, and locate the con-

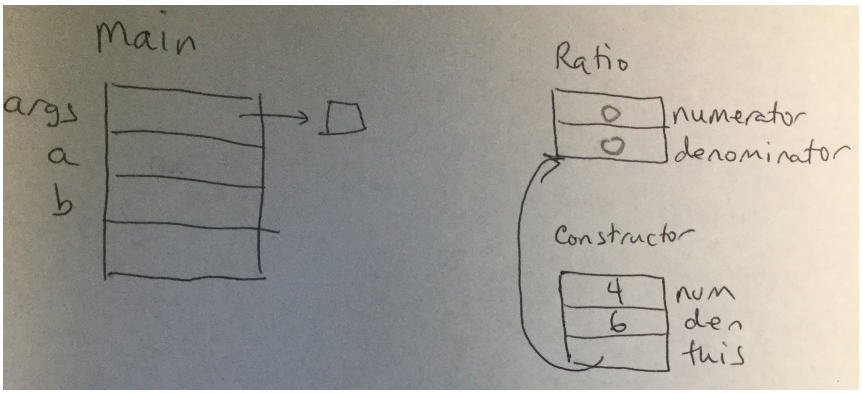
structor that takes two `int` parameters. Once it locates the class, it will allocate memory for the instance variables of the `Ratio` class from heap memory.



Going forward, we will separate the memory allocated into the stack (typically on the left) and the heap (typically on the right).

Our `Ratio` class has two instance variables of type `int`, named `numerator` and `denominator`. Java automatically initializes all instance variables with zero, `false`, or `null` values, as appropriate. So our `int` variables get 0.

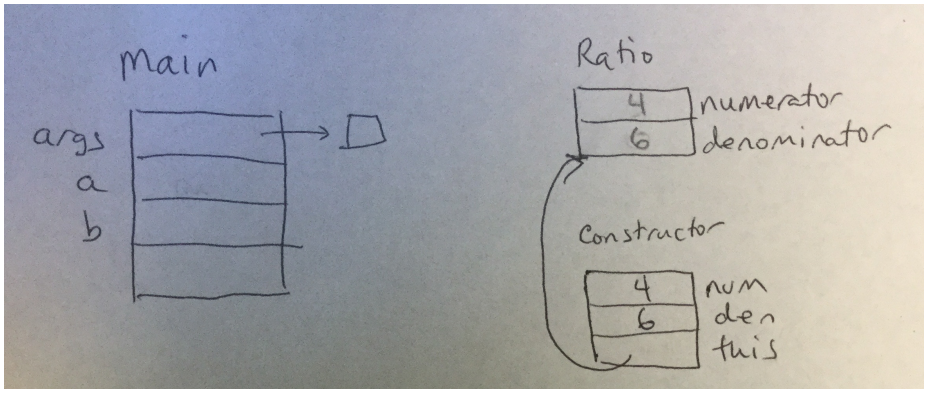
Next, Java sets up the call to the constructor, which acts much like a method. We get a chunk of memory (this time stack memory) large enough to hold the parameters to the constructor, any local variables in the constructor, and the special `this` reference that will tie this constructor call to its object. The formal parameters `num` and `den` have their values initialized using the actual parameter values from the construction (in this case, 4 and 6). The `this` reference is initialized to point to the instance variables we just created in the previous step.



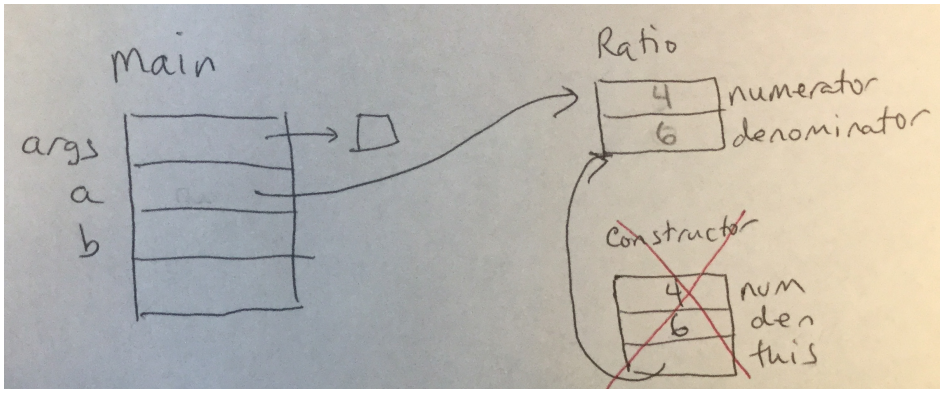
Now we are ready to execute the body of the constructor. This one is pretty straightforward, it's just two assignment statements. But even there, things are not trivial. There are four names involved in the two assignment statements, and Java needs to figure out which of the boxes in our diagram have the values we want to read or are the locations we want to write in each. The process is straightforward. It first looks in its list of parameters and local variables for a matching name. If none is found, it follows its `this` reference to look for a matching instance variable. In our

case, `num` and `den` are found in the parameters/locals list in stack memory, and `numerator` and `denominator` are found by following the `this` reference to the instance variable list.

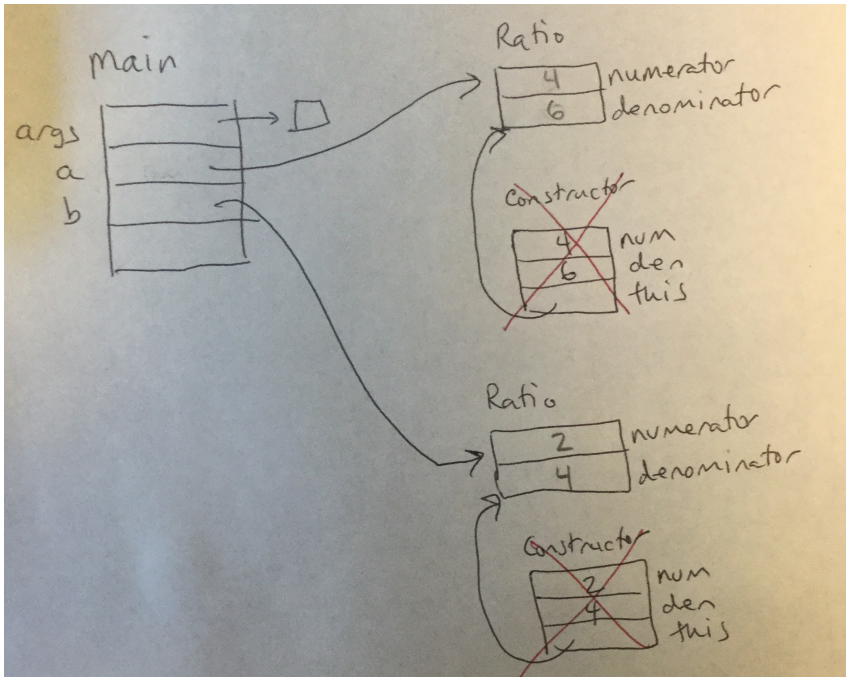
Once these two assignment statements are executed, our instance variables now have taken on the values of the parameters.



When the constructor returns, two things happen: its memory for parameters and local variables goes away, and it returns the reference to the new object's instance variables. In our case, we're storing that reference in `main`'s local variable `a`.



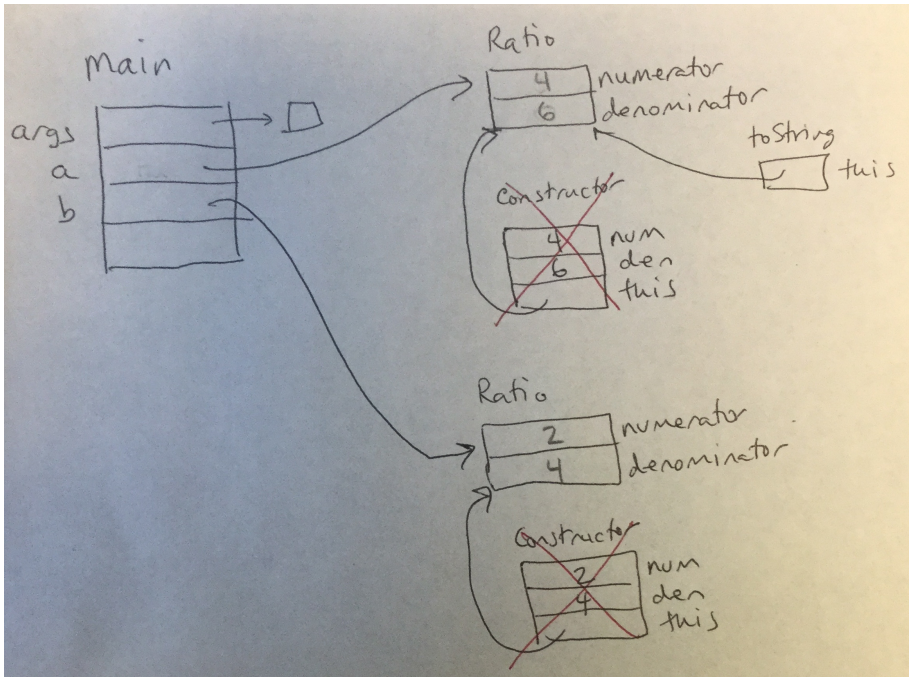
The same set of steps happens when we construct the second `Ratio` and store it in `b`.



Next up in the `main` method is

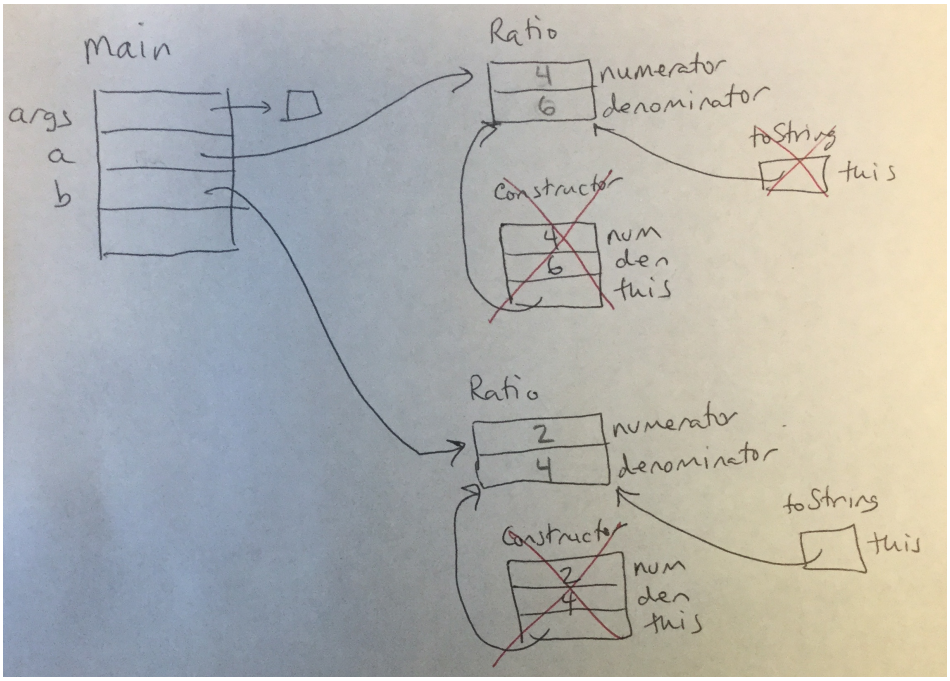
```
System.out.println("Ratio a is " + a);
```

As you might recall, this results in a call to `a`'s `toString` method. Any method call requires Java to allocate memory on the stack for its parameters, local variables, and in the case of non-`static` methods, the `this` reference to the object's instance variables. Since `Ratio`'s `toString` method has no parameters or local variables, it's just `this`.

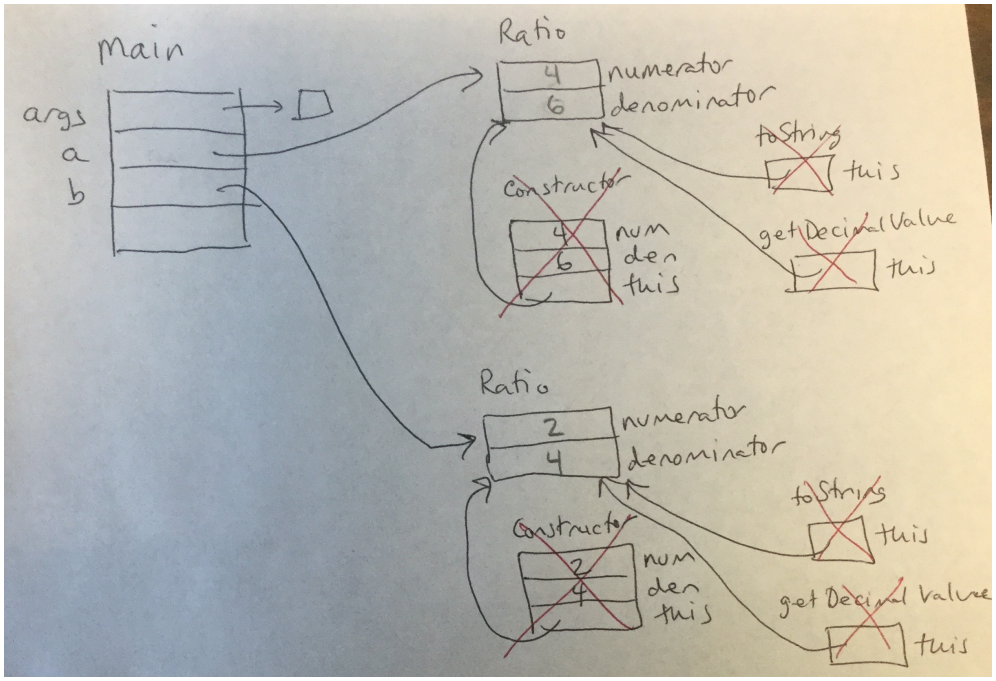


An important thing to notice here is that `toString`'s `this` reference is initialized to the object reference, in this case, `a`.

When the line printing `a` using its `toString` method implicitly completes, its chunk of memory on the stack is deallocated, and we do the same things for the explicit call to `b`'s `toString` method.



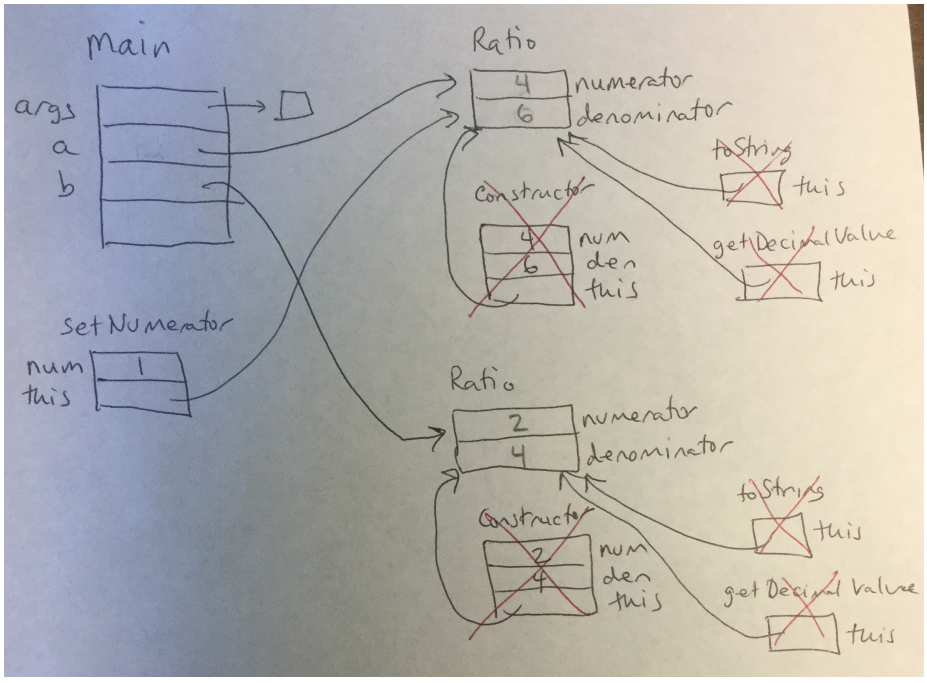
Moving things along a little more quickly, since it's the same idea as what we just saw, the calls to `getDecimalValue` on each of `a` and `b` would also result in chunks of memory on the stack for each of their calls.



A bit more interesting is the call to

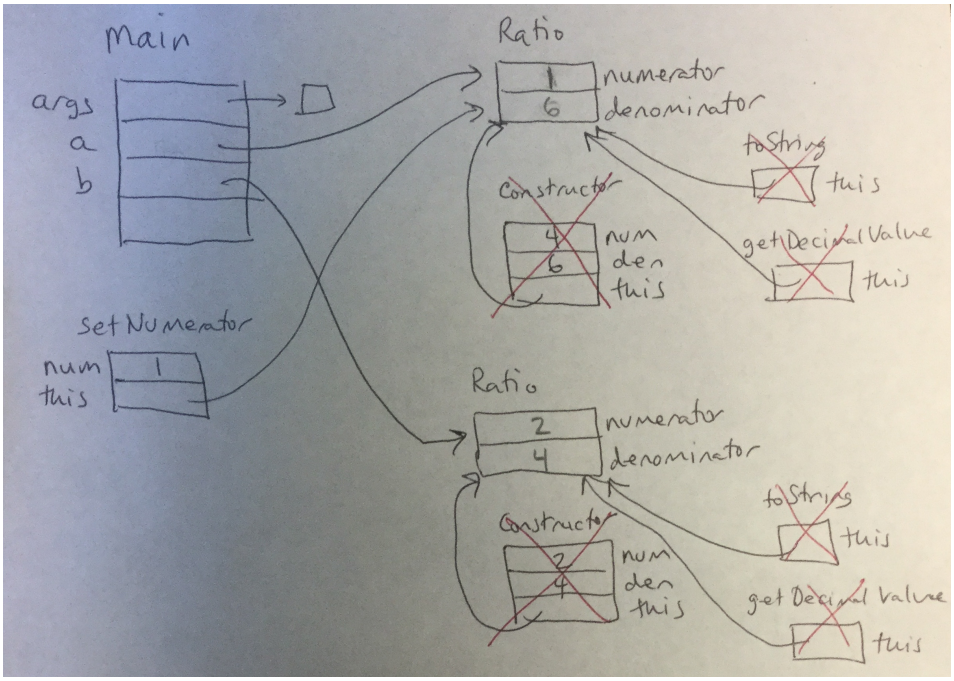
```
a.setNumerator(1);
```

Again, it's a method call, but this one does have a parameter. To set up the method call:



We have a slot for the formal parameter `num`, initialized to the value of the actual parameter, 1. Then the `this` reference, initialized to the object, which is the thing that comes before the `.` in the call.

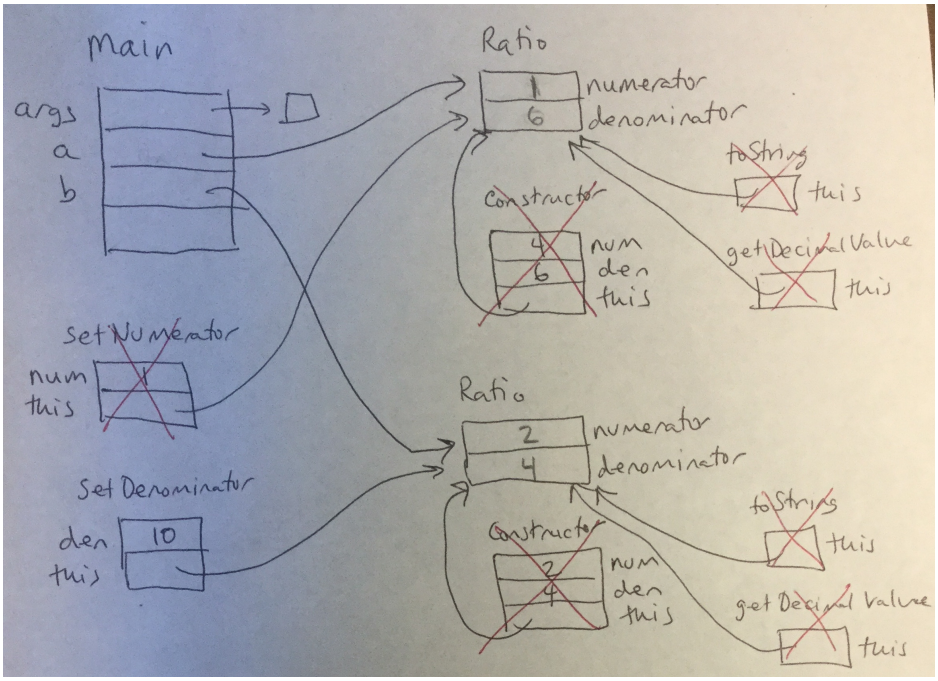
Now the setNumerator method is ready to execute, and it changes the value of `a`'s `numerator`.



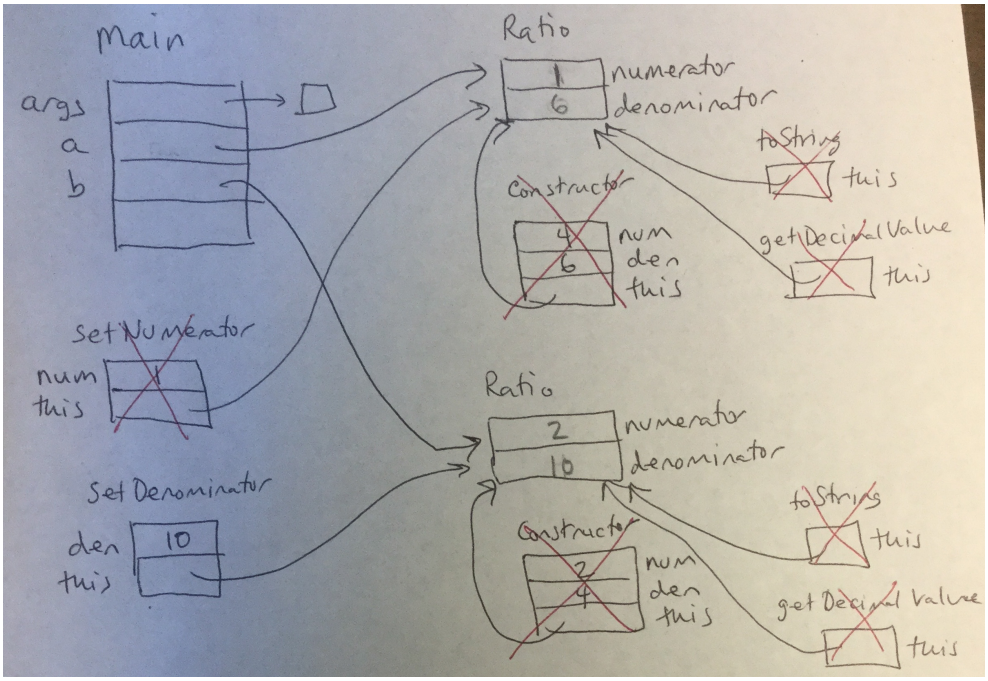
The same process applies to the next call.

```
b.setDenominator(10);
```

The setup:



And when the method executes, `b`'s `denominator` is updated. Shown below is the state of memory just before `setDenominator`.



Adding a `static` Variable

In the "`counted`" directory of the example repository, there is a modified version of the `Ratio`

class that includes a *class variable* – one with the `static` qualifier, that counts the number of `Ratio` objects created.

No matter how many `Ratio` objects are created, there will always be exactly one copy of the `countRatiosUsed` class variable. It is accessible in methods of all instances of the class, and would also be accessible in any `static` methods of the `Ratio` class, if it had any.

In class, we will see how to represent class variables in a memory diagram using the specific example of the `main` method in `CountedRatios`.