SIENAcollege
Computer Science

Computer Science 210
Data Structures
Siena College
Fall 2017

Topic Notes: Classes

So far, our example programs have operated 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`

We next focus on these object types a bit more. In particular, we will starting 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 we've been using in our programs, 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*. We have been writing classes so far this semester 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.

See Example: RatiosNoClass

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.

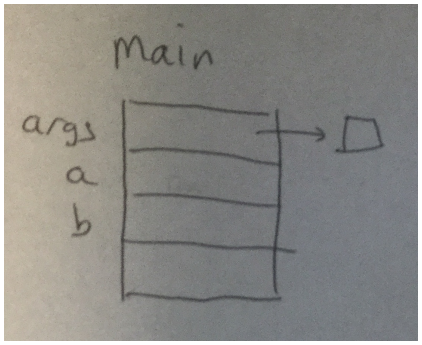
See this example and the extensive comments within for details.

See Example: Ratios

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:



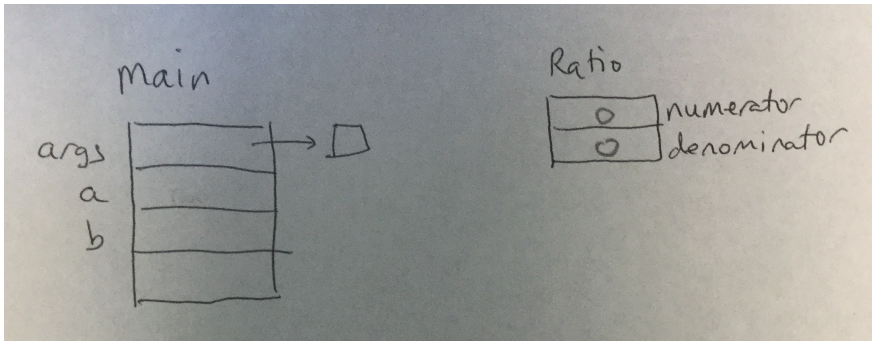
`args` is the one parameter to `main`, and that gets initialized with a reference to any command-line parameters we pass to our program. Since in this case, we aren't expecting any, we will just represent those as the empty box. The `main` method also has two local variables, `a` and `b`, each of which 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);
```

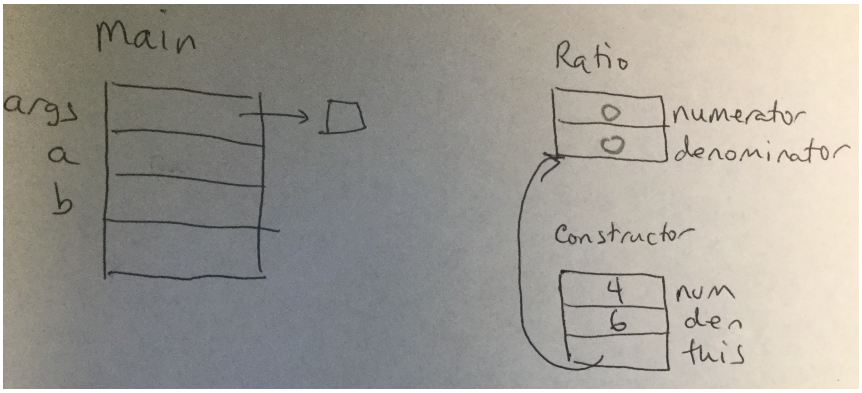
causes many things to occur, and we will consider many of them in some detail, updating our memory diagram for each step that affects it.

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 constructor 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.



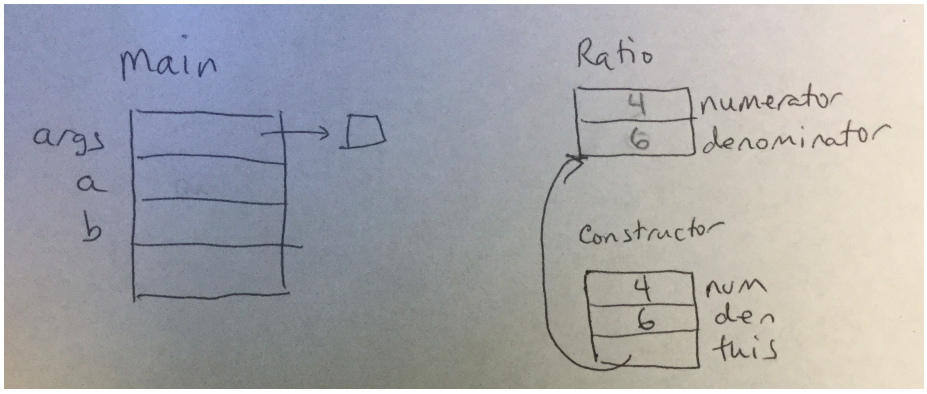
Our `Ratio` class has two instance variables of type `int`, named `numerator` and `denominator`. Java automatically initializes all instance variables with zero, `false`, or `null` values, as appropriate. So our `int` variables get 0.

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.

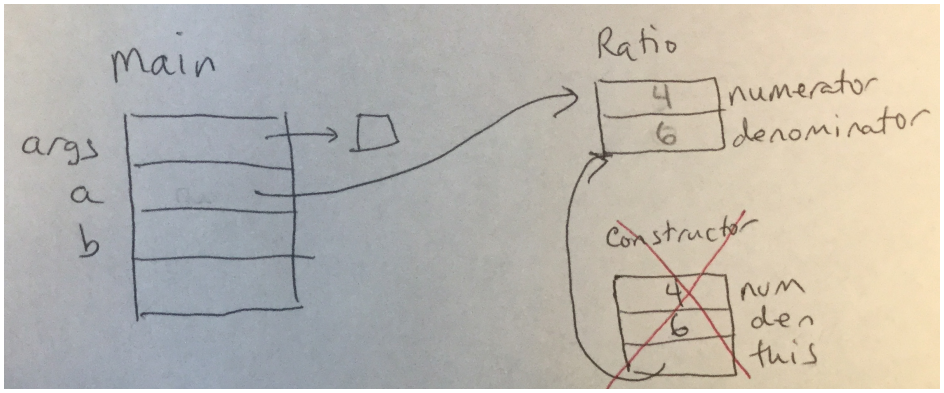


Now we are ready to execute the body of the constructor. This one is pretty straightforward, it's just two assignment statements. But even there, things are not trivial. There are four names involved in the two assignment statements, and Java needs to figure out which of the boxes in our diagram have 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.

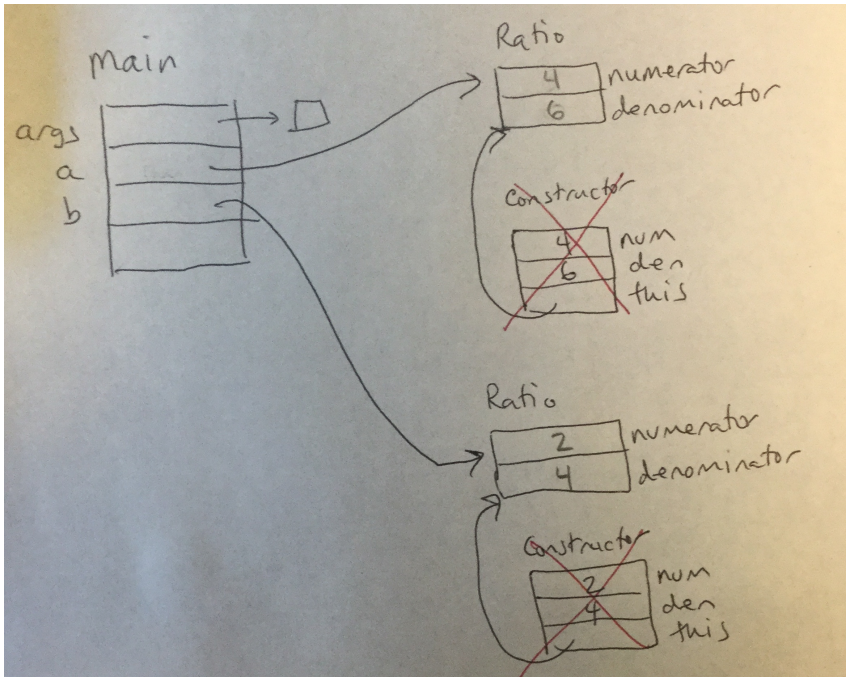
Once these two assignment statements are executed, our instance variables now have taken on the values of the parameters.



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`.



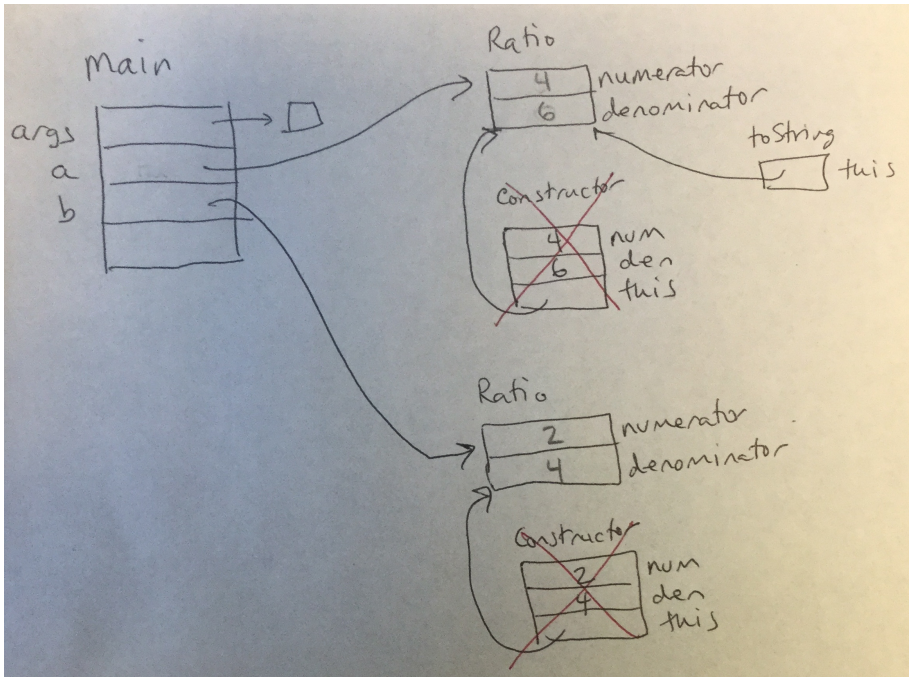
The same set of steps happens when we construct the second `Ratio` and store it in `b`.



Next up in the `main` method is

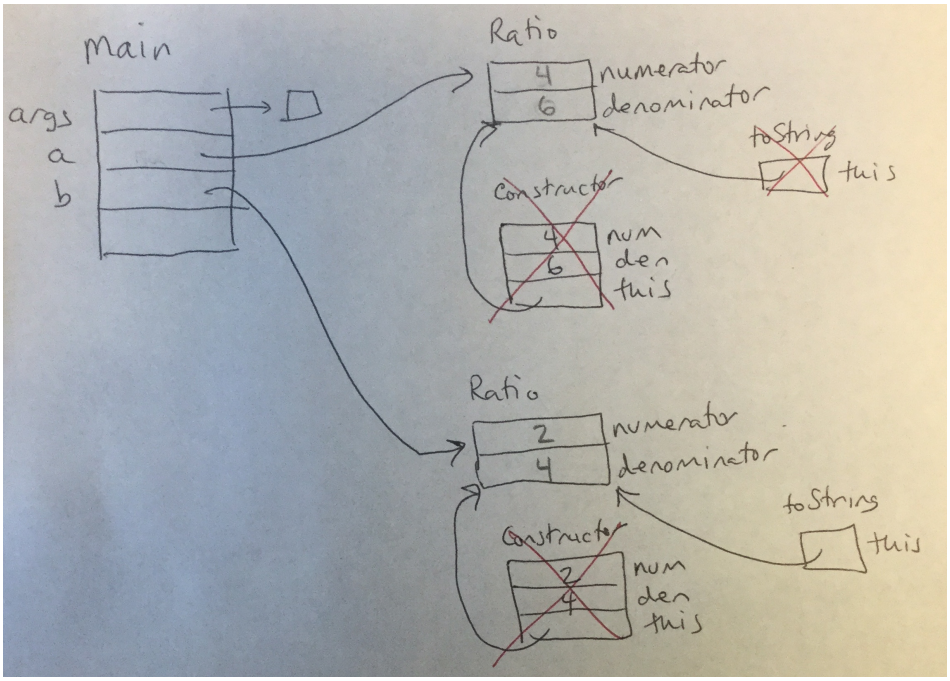
```
System.out.println("Ratio a is " + a);
```

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`.

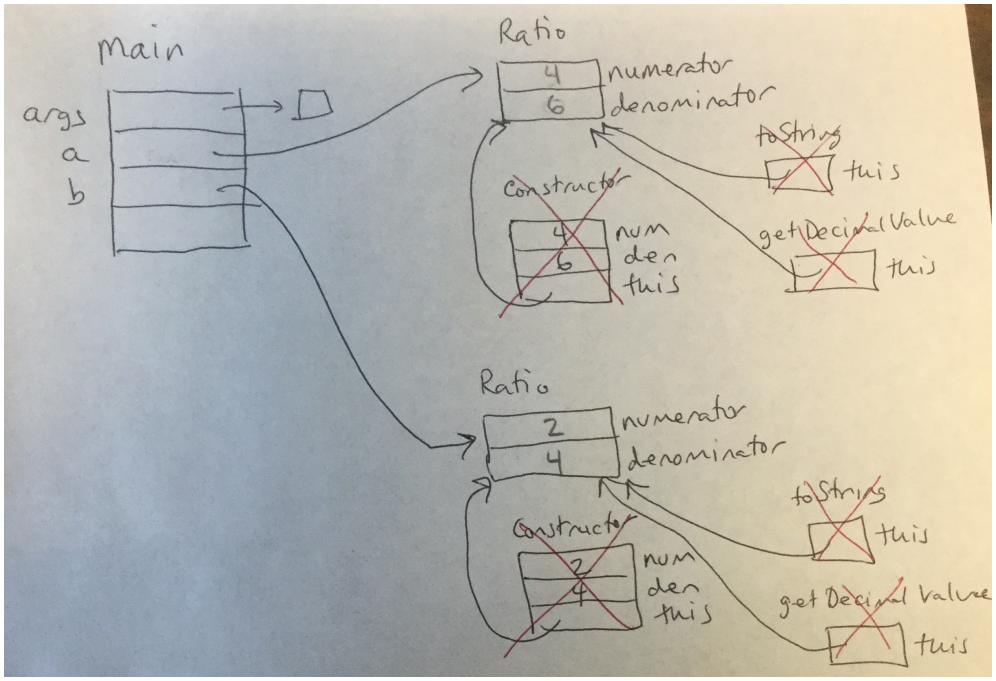


An important thing to notice here is that `toString`'s `this` reference is initialized to the object reference, in this case, `a`.

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.



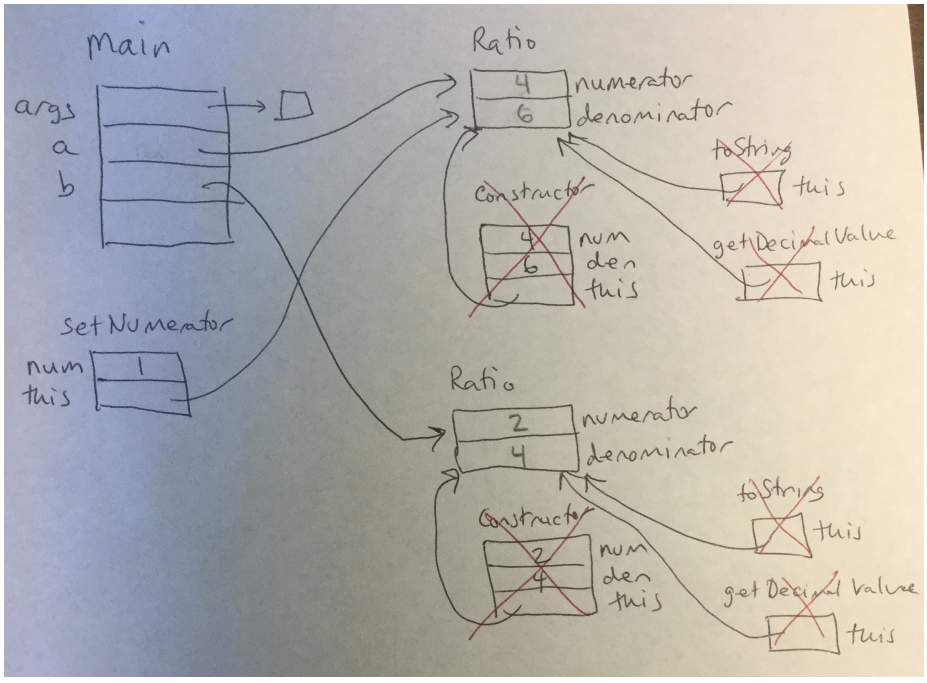
Moving things along a little more quickly, since it's the same idea as what we just saw, the calls to `getDecimalValue` on each of `a` and `b` would also result in chunks of memory on the stack for each of their calls.



A bit more interesting is the call to

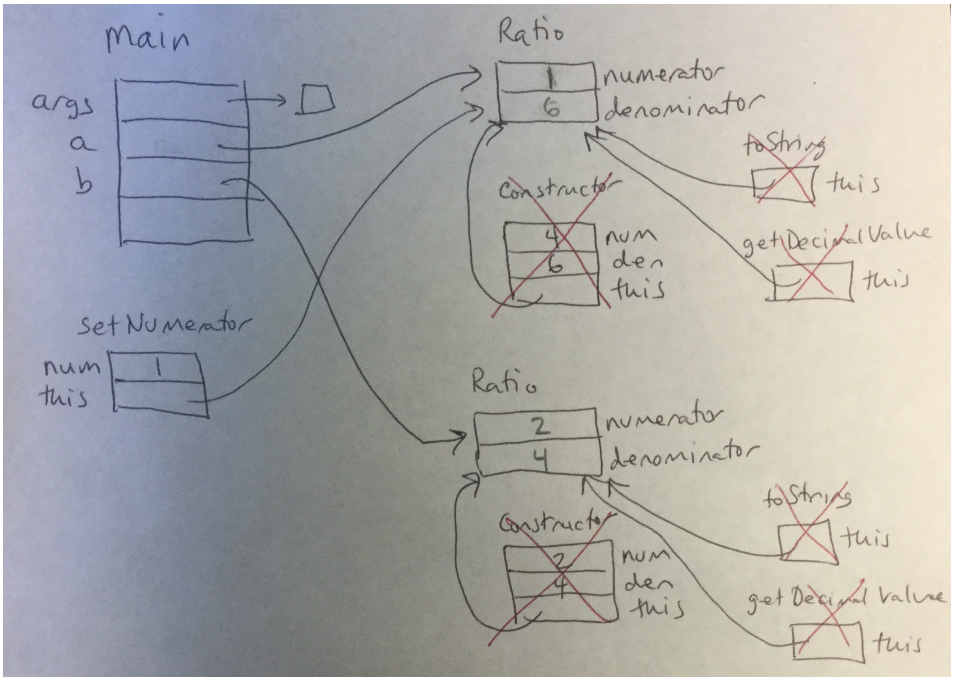
```
a.setNumerator(1);
```

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.

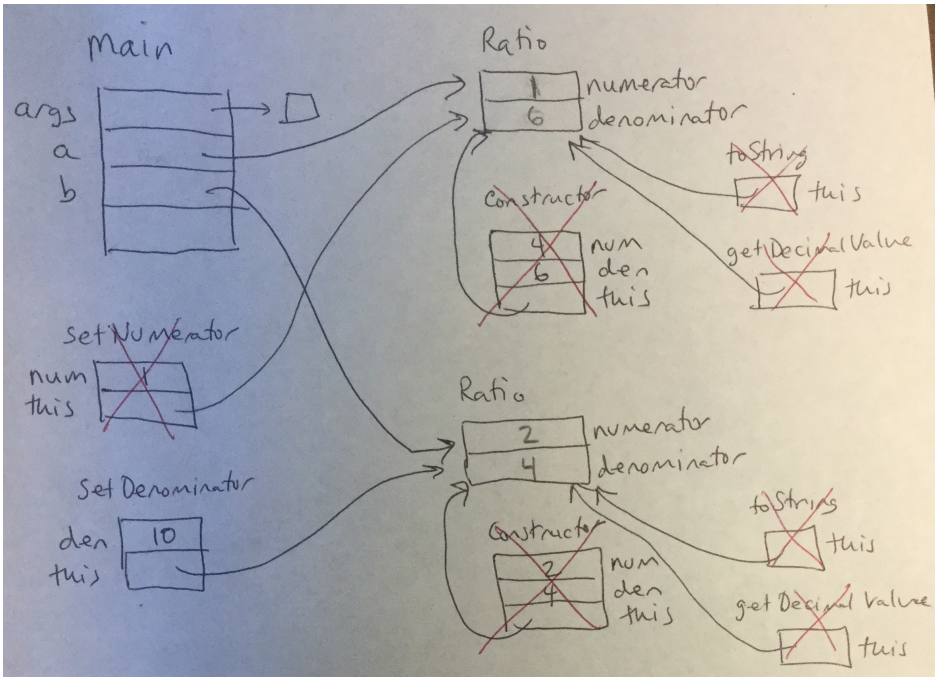
Now the setNumerator method is ready to execute, and it changes the value of `a`'s `numerator`.



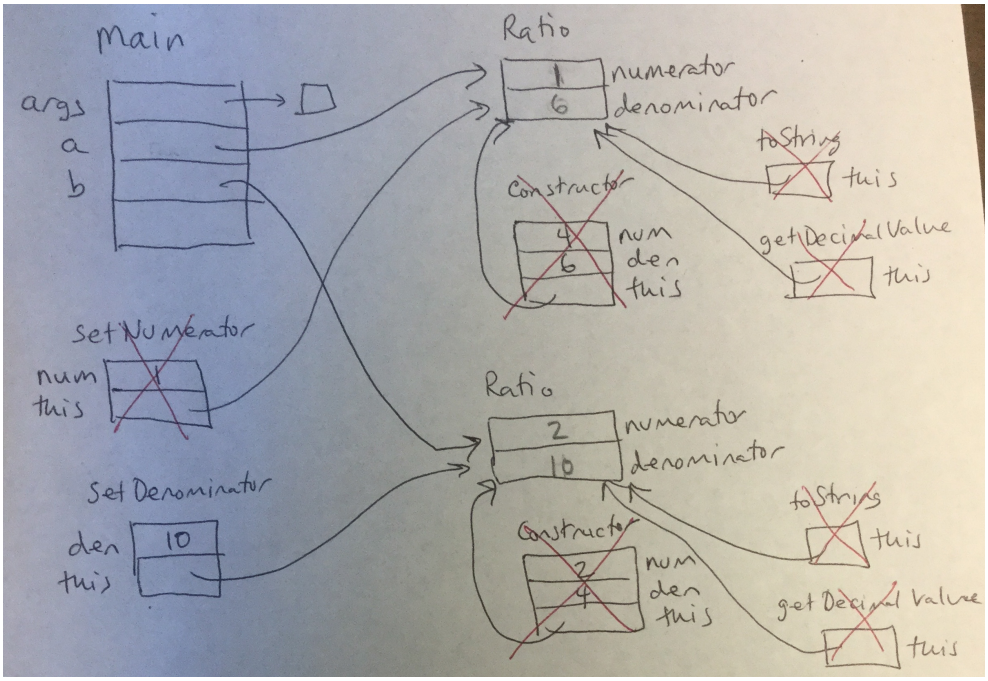
The same process applies to the next call.

```
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`.



Object Orientation

The key to successful object-oriented programming is to identify and design the objects in the problem. Looking first at the ratios example, it is essential to notice that the `Ratio` class defines

the data and operations related to a single ratio. It does not know or care how or why the ratios are being used. It simply defines a ratio and methods that operate on one. It is up to other code – in this case the `main` method in the `Ratios` class – to determine when and why `Ratio` objects are instantiated and how they are used.

One popular metaphor for a class definition is to think of it as a cookie cutter. I prefer another: a file cabinet of forms and instructions. Think of the class as a master copy of a form that contains information about an object, how to create one, and instructions for how the object works. Each time we need a new instance of the object, we make a copy of that form and use it to keep track of the details of that object.

As a further example, let's consider a program intended to keep track of a series of items purchased in a store. Each item has a name, a unit price, and a quantity purchased by a customer. Our program will read in a series of these items and track the most expensive, least expensive, the largest quantity purchased, and the largest total cost (unit price times quantity). For simplicity, we will break any ties by the first encountered. So if the most expensive item cost $9.50 but there are two different items each priced at $9.50, we will keep the first and ignore the second.

We will make use of a class that represents one of these items. We will then be able to create an instance of this class for each such item. We can add accessor methods to retrieve the information we need to implement our program.

See Example: PurchaseTracker

One additional item here is the use of a *class variable* in the `PurchasedItem` class. The `DecimalFormat` object we declare and construct with the `static` qualifier is shared among all instances of the class, no matter how many we create.

General Purpose Classes

The few custom classes we have seen so far have been developed for very specific situations. The `Ratio` class stores a ratio, but isn't useful for much else. The `PurchasedItem` class in the purchase tracker example contains a `String`, a `double`, and an `int`, all used for very specific purposes.

Part of a good object-oriented design is to find places where we can write some software (in our case, a Java class) that might be useful in situations beyond the one at hand. It turns out there are many simple and no-so-simple structures that arise in many contexts, and rather than developing a new Java class each time we need one, we strive to re-use ones that already exist. Or if one does not exist, develop one that will satisfy needs of future programmers as well.

Let's consider one common *data structure*: the *pair*. To begin, let's assume that we want a pair of floating-point values. These might be used to represent a coordinate in two dimensions, a latitude and longitude, or perhaps even a pair of corresponding data values from a science experiment such as an object's volume at a given temperature.

Without knowing the reason someone might want to use our pair of `double` values, we can write a class that encapsulates them:

See Example: DoublePair

Notice that we have only very general-purpose code here. While it might make sense, for example, to add up the two numbers in the pair in some cases, we don't add that to our general purpose class.

Before we move on, we notice a couple of other items of interest.

- We provide a method `equals` that returns whether this `DoublePair` is the same as another `DoublePair`. `equals` is another of those methods (like `toString`) that are provided by Java for any object type. But like `toString`, we normally will want to provide our own `equals` method that determines the equality or equivalence of two objects of the class we are defining in some meaningful way. Here, we define `equals` so that two `DoublePair` objects are equal only if each of the numbers this one contains are equal to the corresponding numbers in the other.

  Note also that the method signature of the `equals` method includes a formal parameter of type `Object`, even though we know that for this to make any sense, it should be passed as an object of type `DoublePair`. It is necessary to define it with `Object`, however, as `equals` methods must have this same method signature for all classes to conform to Java's rules.

  The main complication of this fact is that we need to tell Java that we expect this `Object` to be a `DoublePair`, so we can make the meaningful comparison. The first line of the body of our `equals` method is a *cast*, which tells Java that the object we passed in as `o` should subsequently be treated as a `DoublePair` called `other`. The main drawback of this is that if we pass in some other type of object, the program will crash when the cast is attempted, with a `ClassCastException`. Uncomment the last line of the provided `main` method to see this happen.

- We provide a `main` method that tests our class. This method would not be used by a "user" of this class – it is provided only as a convenient way to test the class. It would work exactly the same way if we placed the `main` into a separate class.

Making it More General Purpose

While it's nice to have a pair of `double` values, what if we need a pair of `int`s, or `String`s, or `PurchasedItems`? Or maybe a pair where the "first" is a `String` and the "second" is a `double`?

Java's `Object` class, which we just saw as a parameter to the `equals` method, provides a mechanism we can use for this.

See Example: ObjectPair

The code is nearly identical, except now our "first" and "second" can be any object type we wish.

In the `main` method, we actually pass primitive types as well, but that is being facilitated by Java's *autoboxing* functionality, supported by Java versions 1.5 and up.

Since primitive types, like `int`, `double`, and `boolean` are not objects, they do no qualify as a valid item to be stored in an `Object` variable. But, since we often do want to treat such values as objects, Java provides a set of classes, including `Integer`, `Double`, and `Boolean`, that are objects whose sole purpose is to include one `int`, `double`, or `boolean`, respectively. In old Java versions, we would have had to create such objects explicity (*e.g.*, "`new Double(9.1)`") but the Java compiler will now insert code to do this for us automatically.

Making it Generic

Another relatively feature of Java, also introduced in Java 1.5, is to allow class definitions to include *generic*, or *parameterized data types*. This means that we can write a definition of the structure using data types that are unspecified (much like the value of a method parameter is unspecified) until we create an instance of the class. But once we "bind" to a type, we have to stick with that type (unlike how our `ObjectPair`'s "first" was initially a `String` and later a `Double`).

Here is the generic version of our pair class:

See Example: GenericPair

The important feature here is the use of *type parameters* to specify the actual data types we want to use for the first and second entries of our pair. These are specified throughout in this example as `U` and `V`. We assume (and in fact, require) that all places we refer to `first` are of type `U` and of `second` of type `V`. This includes declarations of instance variables, formal parameters, and return types.

Only when we construct an actual `GenericPair` object do we specify the types we wish to use. See in the provided `main` method how this is done.

As an example of a place we could use the generic pair, let's look at a program that reads in a list of *Harry Potter* spells into a pair of parallel arrays and allows the user to look up the spell actions by name.

See Example: SpellsArrays

Rather than having two separate arrays of `String` values, we could group them in `GenericPair` objects. Each will have the spell incantation in its `first` and the spell's action in its `second`.

See Example: SpellsArrayGenericPair

Associations

Let's consider a very simple example of a data structure that we'll call an *association*.

As the name suggests, an association is a way to associate pairs of objects, one of which is the *key* and one of which is the *value*. Unlike the "pair" structures we just considered, once created, the key of an association cannot be changed, only the value can.

This is another "general purpose" structure, but one with the above restriction.

This is the first of many situations where we will study and use a data structure that is a restricted

version of one we already have. This structure does everything a pair does, except it does not allow the key to be modified. We will later that we can use this restriction to our advantage.

When developing any such structure, there are some questions to be answered first.

- What should such a structure look like?
- What instance variables will it need?
- What constructors should be provided?
- What methods will it need?

As we have seen, we have two main options when developing a generic class. We can develop our structures to hold references to `Objects` and then use them to store instances of any Java class, or use type parameters. The latter is the usual approach in modern Java programs.

An implementation of this structure is in a modified version of our spells example:

See Example: SpellsArrayAssociation

This is also our first opportunity to look at an implementation of a structure in the Bailey text's "structure package".

The `Object`-based implementation:

See Structure Source:
`structure/Association.java`

The generic implementation:

See Structure Source:
`structure5/Association.java`

This class is defined as part of `package structure`, meaning it can access `protected` entries of other classes in the structure, and those classes can access `class Association`'s `protected` items.

The actual implementation of the `Association` class is pretty straightforward. A couple of quick notes:

- We require a `key` to construct a new `Association`, but the `value` is optional. If not provided, the `value` part defaults to `null`.
- Two `Associations` are considered equal (by the `equals` method) if their `keys` are the same, regardless of their `values`.
- We have an accessor for the `key` (`getKey`) but no mutator. Once created, the `key` of an `Association` may not be modified.
- For the `value`, we have both an accessor (`getValue`) and a mutator (`setValue`).