4. Using Static Members

Contents

[1. Introduction 1](#_Toc18067)

[2. Static Members 1](#_Toc23864)

[3. Using Static Members 1](#_Toc26763)

[4. Enhancing MathEquation with Static Members 1](#_Toc29438)

[5. Static Imports 2](#_Toc16333)

[6. Static Initializers 2](#_Toc154)

[7. Summary 2](#_Toc17780)

# 1. Introduction

=>slides: Pg. 1-2

Welcome to our next module, Using Static Members.

=>slides: Pg. 2

There are times when we are designing our classes that we need to have values and actions that are associated with the class itself, and that's where static members come in. So in this module, we'll start out with a quick overview of static members. We'll then look at how we can have static fields, static methods, and we'll also look at something new with static imports, which allow us to use a shorthand when accessing static methods. And then we'll finish up with a look at how it can do type initialization with static initialization blocks.

=>slides: Pg. 3

Now when we associate members of our classes, most commonly those are instance members. Meaning that each time we create an instance of the class, that instance has its own copy of the values, and the methods on that instance interact with those copies. Static members are different. Static members are shared class‑wide. Meaning that static members are not associated with an individual instance. They're associated with the class itself. Now the way we declare static members is by using the static keyword. The declarations are very much like the way we declare instance members, we just annotate them with a static keyword. And because these members are associated with the class itself, we're going to access them using the class name rather than relying on an instance variable. Hurry to turn on our next section, let's start looking at the details of how we work with static members.

# Static Members

=>slides: Pg. 4

So now the first type of static member we'll look at are static fields. A static field is a value that's not associated with a specific instance of a class. This value is associated with a class itself, and what that means is each instance of the class accesses the same value. There's not a separate copy for each instance, there's just one value for the whole class.

=>slides: Pg. 5

So let's look again at our Flight class. Now we know our Flight class already has a couple fields, passengers and seats. Now although we normally call these just fields, in reality they're instance fields, again, meaning that each instance of the Flight class will have its own copy of these fields. But now if I declare another field here, allPassengers, and indicate that it's static, that means that there's one value for the whole class. Each instance of the class will access that exact same value. So let's see how this would impact our add1Passenger method. Now as you recall, the way this method is implemented, so we use our two instance fields, passengers and seats, and as long as there are still seats available, we go and increment passengers. So that would increment the number of passengers associated with a specific instance of flight. But now let's go ahead and add 1 to allPassengers as well. Now remember that allPassengers is static, meaning that it's not tied to a specific instance. So what allPassengers would allow us to do is count the total number of passengers across multiple instances of the Flight class.

=>slides: Pg. 6

Now in addition to static fields, we also have static methods, and a static method performs an action that's not tied to a specific instance. In other words, it's an action that's tied to the class. And static methods can only access other static members. Static methods cannot access instance members.

=>slides: Pg. 7

So again, looking at our Flight class, we have our instance fields, passengers, and seats. We have our static field, allPassengers. Well allPassengers is private, so it's not accessible outside the class. So if we want to make its value available, we'll need to wrap it in a getter. So notice our getAllPassengers method is also marked as static, and so it can return that value for allPassengers. But now if we're using allPassengers to keep track of the number of passengers across multiple flights, we might periodically want to reset it back to 0. So we have another static method here, we set allPassengers, that will set allPassengers = 0. So now with that, our Flight class has a static field, allPassengers, and two static methods, getAllPassengers and resetAllPassengers. So now in our next section, let's take a look at how we'll use these static members.

# Using Static Members

=>slides: Pg. 8

So let's see what it's like to work with our Flight class now we have some static members. So let's start out by creating an instance of our Flight class, and we'll assign it to a variable, laxToSlc. So as we know, that'll give us that variable. It'll hold a reference to an instance of our Flight class. And note that that instance has its own copy of the fields, passengers, and seats. And then if we go off and create another instance of our Flight class, we'll assign that to a variable dfwToNyc. Again, that creates our variable and it holds a reference to a separate instance of the Flight class. And this separate instance has its own copy of the fields, passengers, and seats. Now remember that our Flight class now has a static member, allPassengers. And as a static member, that field is associated with the class itself being that it's separate from any instance of the class. Now just like instance members, a static integer field will default to zero.

=>slides: Pg. 9

But let's go ahead and call resetAllPassengers anyway because it's possible that some code earlier has already been affecting the value of allPassengers. So if we want to keep track of the total passengers on just these two flights, we'll call resetAllPassengers just to make sure allPassengers is set to zero. Now notice here when we call resetAllPassengers, we're calling it using the class name because, again, we setAllPassengers as a static method so it's accessed using the class name.

=>slides: Pg. 10

So let's go ahead and call add1Passenger on our laxToSlc variable. Now of course, that will follow that reference to the appropriate instance. When we call add1Passenger, it'll take our passengers instance field and increment that from 0 to 1. Remember that we also now increment allPassengers. So that'll also increment from 0 to 1.

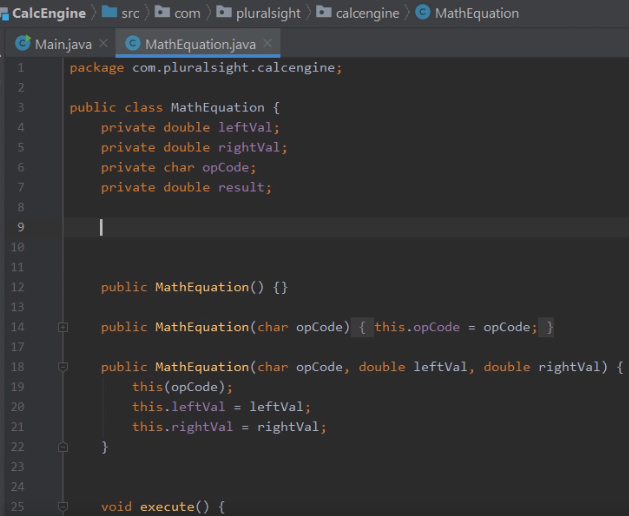
=>slides: Pg. 11

We'll call add1Passenger again on laxToSlc. So our instance field passengers goes from 1 to 2, and our static field allPassengers also goes from 1 to 2.

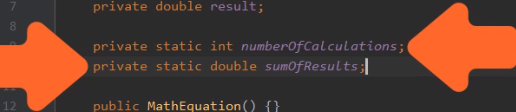
=>slides: Pg. 12

Now let's go ahead and call add1Passenger on dfwToNyc. So again, we'll follow the reference to the appropriate instance, and that will increment the copy of passengers contained in that instance. But again, we're also going increment allPassengers. AllPassengers is a static field again that's shared across all instances. So allPassengers will increment from 2 to 3. So if we print out the value for getPassengers on laxToSlc, that would, of course, return 2. Do the same thing for dfwToNyc, that, of course, will return 1. And then if we print out the value for flight.getAllPassengers, that would print out the value 3. So notice that each instance is tracking its own values because it has its own copy of the instance values. But all of the instances affect a single static value. All right, so to help us get a better understanding of all this, in our next section, let's jump back into our CalcEngine project. We'll start adding static members to our MathEquation class.

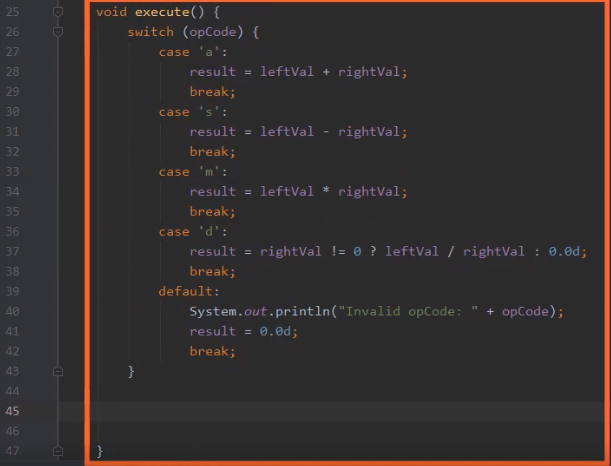
# Enhancing MathEquation with Static Members



Here we are back in STS looking at our CalcEngine project, and what we want to do now is see how we can use static members to add a new feature to our MathEquation class. And the feature we want to add is to provide the ability to get back the average result for a series of calculations. And we want to calculate this average result in a way that's not tied to any single instance of the MathEquation class. We want an average result across all the instances of MathEquation. So to do that, we'll need to use static members. Now in order to determine the average, one of the things we'll need to know is how many times we've calculated our equations.



So let's go ahead and add a static field here named numberOfCalculations of type int. Now once we have that field, we'll also need a field to hold the sum of all the results of these calculations. So let's add another static field named sumOfResults of type double. So now with these two fields in place, we have everything we need to start calculating that average. Now remember that because these fields are static, there's only one copy of these fields for the whole class. It doesn't matter how many instances of the class we create. Also, keep in mind that static fields, just like instance fields, automatically have default values set. So both of these fields will start out as zero, which is exactly what we want. All right, so now that we have these fields declared, let's head down to our MathEquation class's execute method.



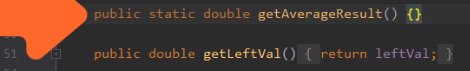
So now we're down here in our execute method. And as you recall, the way our MathEquation class works is that we provide an opCode and two values, and then we call execute to actually determine the result. So inside this method is a great place for us to keep count of the number of calculations we had to execute.



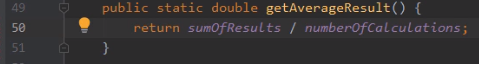
So here after the switch statement where we determine the results for the equation, let's go ahead and increment the number of calculations by 1.



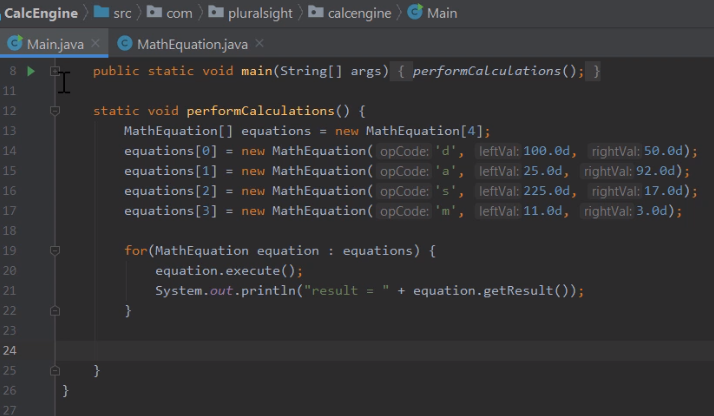
And then once we've done that, we can take the result for this MathEquation instance and add that result to our field someOfResults. So now with those two statements in place, that gives us everything we need to calculate our average result across all the different instances of our MathEquation class. So all we need to do now is provide a method to get back that value. So let me just scroll down here a little bit. So here we are now just after the execute method.



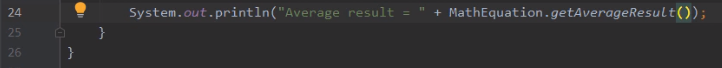
So let's go ahead and add a static method named getAverageResult of type double. And then inside this method, we can go ahead and calculate the average result and return that value back.

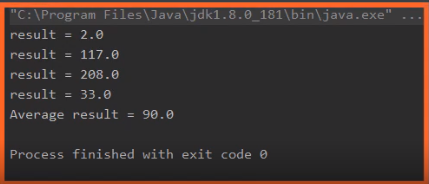


So to calculate the average, we'll take our sumOfResults and divide it by our number of calculations. And that easily we now have the ability to determine the average result across all our MathEquation instances. So now what we need to do is display that value out. So to do that, let's head back over here to our Main class.



Now remember here in our Main class, we have this method performCalculations. PerformCalculations has an array of MathEquation instances. We then loop through that array, calling execute on each of those instances and to display the corresponding result. So that means that once this loop completes, we should be able to callGetAverage result and display out the average for all those calculations.



So just after our for loop, we'll call System.out.println. And then to display our average, we'll call getAverageResult. And remember, because it's a static method, we're going to access it through the MathEquation class name. And that's all there is to it. So let's go ahead and run our application and verify that it behaves as we expect. So now once the program runs, let's take a look here at our output. 

You can see the results of each of the MathEquation instances. And if we were add each of those results up, it would give us a total of 360. And we have four results there, so if we divide 360 by 4, that gives us an average result of 90, and that's exactly what we displayed. So our code is doing exactly what we want it to do. Each MathEquation instance took care of the details of that individual equation, and then the addition of our static fields allowed us the track information across those instances. All right, so now in our next section, let's take a look at something known as static imports.

# Static Imports

=>slides: Pg. 13

Let's take a look now at static import statements. As you recall in our previous course, Getting Started with Programming in Java, we talked about standard import statements. As you recall, an import statement allows us to have a shorthand for accessing a type because in Java type names are fully qualified by the package that contained the type. And having to type the full package qualified name of every type out every time would be really cumbersome. So import statements allow us to specify at the top of the source file, the package qualified name of the type. And then from that point forward, we can simply use the simple name of the type, which in general is just the class name. Well static import statements take this idea one step farther. Static import statements allow us to access static methods with a simplified notation. What static import statements allow us to do is specify the fully qualified name of the method at the top of the source file. And then from that point forward, we can use the method named by itself. We no longer have to qualify it by the class.

=>slides: Pg. 14

So let's look at some code again that uses our Flight class. As you recall, our Flight class has a static method, resetAllPassengers. Since it's a static method, we access it through the class name, so it's flight.resetAllPassengers. Then we have some code that creates an instance of our Flight class, call add1Passenger on that instance twice. We create another instance of our Flight class, call add1Passenger on that instance. And then we want to get the total number of passengers across those two flights, so we use our static method, getAllPassengers, and again we qualify that method by the class name. Now let's see how using a static import changes this code. Now as you recall, the way a standard import statement works, so we simply say import and the package qualified name of the type. Well, in the case of a static import, we're going add the word static after the word import. And in the case of a static import, we're referencing a static member. So we're going to give the name of the method that we want to use. So now we have a static import for our Flight class's resetAllPassengers method.

=>slides: Pg. 15

So if we look at this code here where we currently say Flight.resetAllPassengers, once we have the static import in place, this line can be simplified to just resetAllPassengers. We can use the method name without having to qualify it by the class.

=>slides: Pg. 16

Now we can do the same sort of thing where we call Flight.getAllPassengers. We can add a static import for getAllPassengers, and this line of code now can be simplified again simply using the method name. And static imports are useful in scenarios where you have some static methods you just have to access very frequently. Now in scenarios where a class has a large number of static methods that you want to use rather than having to list them individually as we've done here, we can actually switch to a star notation.

=>slides: Pg. 17

So we can boil this down to simply Flight.\*. So basically, we're saying we want to import all the static members of the Flight class. The code to use those members is just as it was before. We simply used the member names without having to qualify them by the class. The value of the star notation is that we only need one static import to have access to all those static members. All right, so now in our next section, let's take a look at static initialization blocks.

# Static Initializers

=>slides: Pg. 18

Static initialization blocks allow us to perform one‑time type initialization. The key is the code in a static initialization block is executed before a type's first use. So it gives us a way to centralize the work we want to do to prepare a type for use. Now static initialization blocks are limited to accessing only static members. Now the way we create static initialization blocks is very similar to the way we created instance initialization blocks. It's going to be code enclosed in brackets to mark it as a static initialization block, the brackets are preceded by the word static, and this code is going to be outside of any method or constructor.

=>slides: Pg. 19

So to see how to use a static initialization block, let's again look at our Flight class. Now recall our Flight class has these two instance fields, passengers and seats. We also have a static field, allPassengers. Well let's go ahead and another static field, maxPassengersPerFlight. And what we use this field for is that there are some scenarios where the flight administration authority may be putting limits on how many passengers are allowed on a given flight. And in some cases, that limit might be lower than the number of seats a flight actually has. And this limit is going to be the same for all flight instances. But in order to find out the limits, we need to go out and connect up to a service and retrieve the value back from that service. So we'd like to do that lookup work once before we start using our Flight class. So this is a great chance for us to use a static initialization block. So here within these brackets preceded by the word static is our static initialization block. So inside of our initialization block, we'll go ahead and create an instance of the admin service and then connect up to the service. Once we connect to the service, we're going to go ahead and check and see are there any restrictions in effect? Now if there are restrictions in effect, we're going to go out to the admin service and call its getMaxFlightPassengers method, and that will tell us the limit that they've set for the number of passengers on a given flight. But if there aren't any restrictions in effect, we're using the constant Integer.MAX\_VALUE. And that constant contains the largest possible integer. In other words, that constant gives us a value that indicates there's no administrative limit on the number of passengers. Now once we're done with the administrative service, we'll go ahead and close it. Now again the code within this block will run once before the first time we use the class. So that means the work to initialize our field maxPassengersPerFlight is done before we start using our Flight class. So let's see how we can use that field in the rest of our code.

=>slides: Pg. 20

As you recall, our Flight class has an add1Passenger method. Now currently the way this method is implemented is we simply check the number of passengers against the number of seats. As long as there are seats available, we'll go ahead and increment the number of passengers by 1, as well as increment are static field allPassengers by 1. So now we need to evolve this code because seats aren't the only possible limit.

=>slides: Pg. 21

So what we'll do is in addition to checking against the number of seats, we'll also make sure the passengers is less than the maxPassengersPerFlight. So now with this code in place, we only add another passenger if there are still seats available and we haven't exceeded any administrative limits. And because we set our maxPassengersPerFlight field as part of our static initialization block, the work of retrieving that value only had to be done one time, but was available to all instances of our Flight class.

# Summary

=>slides: Pg. 22  
To wrap up, here are some of the key things you want to remember from this module. Remember throughout this module, we've been looking at static members. Now static members are members of our class that are shared class‑wide. Now the way we indicate that the member is static is simply by including the keyword, static, as part of that member's declaration.

=>slides: Pg. 23  
Now the first kind of static member we looked at were static fields. Now the static field is a value that's not associated with any instance. In other words, all instances of the class access the same value, so it's a value that shared across all class instances. Now we also have static methods. Static methods allow us to perform an action that's not tied to an instance. In other words, the action is tied to the class itself. Remember that static methods can only access static members. So static methods can only access other static methods or static fields.

=>slides: Pg. 24  
Then we looked at the static import statement. Remember, the static import statement gave us a shorthand for accessing static methods. Normally our static methods have to be qualified by the type name. By using a static import, we're able to use that method by using the method name by itself, not having to qualify it by the class name. And then we finished up with static initialization blocks. Remember, static initialization blocks allow us to perform one‑time type initialization. Because the code we put inside these blocks executes before a type's first use. So stack initialization blocks give us a great way to do the work in our type, before we start using that type. All right, that wraps up this module. In our next module, we're going to take a closer look at methods and take a look at some of the richer capabilities that methods provide.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*