5. A Closer Look at Methods

Contents

[1. Introduction 1](#_Toc5969)

[2. Passing Objects as Parameters 1](#_Toc32583)

[3. Changes to Objects Passed as Parameters 3](#_Toc17287)

[4. Overloading 4](#_Toc9358)

[5. Method Overloading Examples 4](#_Toc20882)

[6. Matching Method Calls to Overloads 5](#_Toc3562)

[7. Using Method Overloading in CalcEngine 6](#_Toc13221)

[8. Variable Number of Parameters 8](#_Toc860)

[9. Summary 9](#_Toc7662)

# 1. Introduction

=>slides: Pg. 1

Welcome to our next module, A Closer Look at Methods. In our previous course,we introduced methods, and as part of that discussion, we looked at how we can use parameters to pass data values into our methods.

=>slides: Pg. 2

Well, in this module, we're going to take a closer look at Java's capabilities for accepting values and parameters. We're going to take a look at some of the richer features that it provides. Now to start out, we'll take a look at passing objects as parameters. In other words, just what happens when you pass an object as a parameter to a method? Then, of course, the next question is, just what happens if that method starts making changes to an object that it receives as a parameter? Then from there we'll take a look at something known as method overloading, and method overloading allows us to have multiple versions of the same method. And if we can have multiple versions of the same method, there must be some set of rules for how Java decides which one to call. So we'll take a look at how Java goes about resolving which overloaded method to use. And then finally, we'll finish up with a look at variable number of parameters. In other words, how can we create a method that can accept different number of parameters at different times?

# Passing Objects as Parameters

=>slides: Pg. 3

Now as you recall from our previous course, when we talked about passing primitive types as parameters, we said that primitive types were passed by value, meaning that a copy of the value was passed into the parameter. Well, when it comes to passing objects as parameters, objects are passed by reference. What that means is, we don't pass a copy of the entire object into the method. Instead what we do is, we pass a copy of the reference to the object into the method. So that means that the method will have its own copy of the reference. Now since the method has its own copy of the reference, the method is free to make changes to that reference, and any changes it makes to the reference will be visible inside of the method. But again, because the method is working on its own copy of the reference, any changes it makes to the reference will not be visible outside of the method.

=>slides: Pg. 4

So now, to see what this looks like, we're again going to use our Flight class. As you recall, our Flight class has a number of fields. One of those fields is the flightNumber field. One of the constructors for our Flight class accepts a flightNumber, and of course, that constructor then sets our flightNumber field. So let's use our Flight class to see how object parameters behave.

=>slides: Pg. 5

So i have a method here, swapFlight, and it receives two parameters, i and j, both of type Flight. And the code inside of swapFlight wants to swap the references for i and j. So by the end of this method, i should point to the Flight instance that j originally pointed to, and j should point to the flight instance that i originally pointed to. So to call this code, let's go ahead and create two Flight instances and assign those to local variables outside of our swapFlight method. So when we do that, we'll have a val1 reference that points to an instance of our Flight class that has a flightNumber of 10, and our val2 reference will point to an instance of our Flight class that has a flightNumber of 20. So if we now call swapFlight, passing in val1 and val2. One side of swapFlight, the space for those two parameters needs to be allocated, and then the reference contained in val1 will be copied down to our parameter i. So the reference in i will now point to the same Flight instance that val1 points to. Same sort of thing for val2. The reference in val2 will be copied down to our j parameter, so our j parameter will now point to the Flight instance that val2 points to.

=>slides: Pg. 6

So now once we get down inside the code here in our swapFlight method, well, we'll allocate out a local variable named k, we'll initialize it to the value of i. So k will now reference the same object instance that i references. So then when we say i = j, what we want to do now is actually make a change to the i reference. So the reference in i will now point to the same flight instance that j points to. So we've successfully changed that reference. So then when we get to our next line of code, j = k, we're again going to operate on our reference j, and we're going to give the value of k. So j will point to the same Flight instance that k points to. So now at the end of this code, we've successfully swapped our references i and j. I now points with one that j originally pointed to, and j now points with one that i originally pointed to. But again, all this work was done on the local copies of the references that we had inside the method. And when this method exits, all that gets cleaned up with no changes to the original references of val1 and val2. So if we want to point out the flight numbers for those 2 Flight classes, if we follow the reference for val1, it points to the same Flight instance it always did, so it'd print out the flightNumber of 10. If we follow our val2 reference, it also points to the same flight instance it always did. So it'd still print out its original value of a flightNumber of 20. So even though the method was able to make changes to the references, it was only making changes to its own copies. Those changes were not visible outside of the method. But it turns out there are some changes that can be visible outside of the method, and we'll see what those are in our next section.

# Changes to Objects Passed as Parameters

=>slides: Pg. 7

As we mentioned, when we pass objects as parameters to a method, those objects are passed by reference, meaning that the method receives a copy of the reference back to the original object. Remember that objects have members, and since the method has a reference back to that original object, the method can make changes to those members. Now any changes the method makes to the members will of course be visible within the method. But it turns out those changes will also remain visible even outside the method. So those changes will be visible all the way back to the original calling code.

=>slides: Pg. 8

So let's see why this is the case. Let's have another method here, swapNumbers. SwapNumbers again receives two parameters, i and j, both of type Flight. Now here inside of swapNumbers, we're not going to try and swap the references themselves. Instead what we want to do is swap the flight numbers contained within those flights. So again we'll have our code here where we allocate out two instances of our Flight class. So val1 will point to an instance of Flight that has a flightNumber of 10, and val2 will point to an instance of Flight that has a flightNumber of 20. So we'll call swapNumbers, passing in val1 and val2. And of course swapNumbers will allocate out space for the parameters, so we have our parameters i and j, The reference in val1 will be copied down to our i parameter, so i will now reference the same object that val1 references. The reference in val2 will be copied down to j, so j will now reference the same object that val2 references.

=>slides: Pg. 9

So now when we get down here inside of swapNumbers, the first thing we do is allocate out a local variable named k. It's type is int. So we get the flightNumber from the Flight that i points to, and we assign that to k. So now k has that value 10. So now the next thing we want to do is set the flightNumber and the Flight reference by i to the flightNumber of the Flight reference by j. So now, in this case, we're not making a change directly to i. What we're going to do is follow the reference that's contained in i. We want to modify the object that i points to.

=>slides: Pg. 10

So what we're going to do then is take the Flight instance that's referenced by i and set its flightNumber to 20. Then the next thing we do is set the flightNumber of the Flight reference by j. So again, in this case, we're not trying to change j.

=>slides: Pg. 11

Instead, we're going to follow the reference that's contained in j, and when we get there, we're going to set its flightNumber to now have a value of 10. So now notice here we haven't tried to change the references; instead, we're changing members within the objects that these references point to.

=>slides: Pg. 12

So now once our method is complete, we know that everything that was allocated within that message gets cleaned up, so all of our references and local variables here are cleaned up. But now even though those references are cleaned up, the objects that they pointed to are still around because those are the same objects pointed to by val1 and val2.

=>slides: Pg. 13

So if we now go and print out the flightNumbers, we'll first follow the reference in val1, and it points to an object instance that has a flightNumber of 20. So even though that instance originally had a value of 10, if we print out its flightNumber now, it is 20. The change that was made inside of swapNumbers is still there. Same thing for val2. If we follow its reference, it points to a Flight instance that has a flightNumber of 10, even though it originally had a flightNumber of 20. So the key thing to take away from this is that because objects are passed by reference, the method has a reference back to the original object. So any changes it makes to that object are lasting changes and will still be in effect even after the method exits. All right, so now in our next section, let's take a look at something known as method overloading.

# Overloading

=>slides: Pg. 14

Java supports a really important concept known as overloading. What overloading allows us to do is, within a single class we can have multiple versions of a particular method or multiple versions of our constructor. And we've actually seen overloading in practice earlier in this course when we talked about constructors.

=>slides: Pg. 15

Remember our Passenger class. Our Passenger class had multiple constructors. Had a constructor that didn't accept any parameters, one that accepted the number of freeBags, one accepted the perBagFee, and one that accepted the number of freeBags and a number of checkedBags. So in this scenario, our Passenger class had an overloaded constructor.

=>slides: Pg. 16

Now a key concept in overloading is what's known as the signature. Every constructor and every method must have a unique signature, and there are multiple parts that make up the signature. Now one of those parts is the number of parameters.

=>slides: Pg. 17

So again, looking at our Passenger class, our first instructor had 0 parameters, we had another constructor with a single parameter, and then yet another constructor that had two parameters. So by having a different number of parameters, that allowed them to have unique signatures. But notice that number of parameters must not be enough because we have two constructors here that both have a single parameter. So there must be something more than just the number of parameters.

=>slides: Pg. 18

And it turns out another key part of the signature is the data type of each parameter, so developing that unique signature uses both the number of parameters and the type of each parameter.

=>slides: Pg. 19

So again, looking at our Passenger class, we have these two versions of the constructor that both have a single parameter. But notice that one's parameter type is int; the other's parameter type is double. So even though they have the same number of parameters, because the parameter types are different, the signatures are considered unique.

=>slides: Pg. 20

Now when we're talking about constructors, the number of parameters and the type of each parameter is enough to make them unique because constructors have to have a specific name. The constructor's name is always the same as the class name. But when it comes to methods, the method name is also a key part of the signature. So when we're talking about methods, it's that combination of the method name, how many parameters it has, and the type of each parameter that creates a unique signature. So in our next section, let's take a look at some examples of method overloading.

# Method Overloading Examples

=>slides: Pg. 21

As we look at some examples of method overloading, we're, again, going to use our Flight class. As you recall, our Flight class has fields for the number of passengers and the number of seats. We have a method, add1Passenger, that internally will check to see if there's enough seats to add the passenger. As long as there are, it will increment the number of passengers. Well, this code here that checks to see if there are enough seats available to add the passenger is something we might want to do more often. So let's go and add another method here, hasSeating, that contains the logic to do that check. So as long as the number of passengers is less than the number of seats, hasSeating will return true.

=>slides: Pg. 22

So let's go and update our add1Passenger method to use our hasSeating method. And while we're in here in our Flight class, let's add another field, totalCheckedBags, and that will allow us to keep track of how many checked bags we currently have on the flight, and we can do that as we add each passenger. =>slides: Pg. 23

And since we're adding a new feature, keeping track of the total number of checked bags, we might want to add some overloads for our add1Passenger method.

=>slides: Pg. 24

So we'll go and keep the original version of our add1Passenger method that takes care of incrementing the number of passengers, but we'll go and add an overload that accepts the number of bags. So in this case, we're adding an overload that provides additional functionality, accepting the number of checked bags. And in this overload, what we'll do is we'll call the original implementation of add1Passenger, and we'll increment the total number of checked bags. Now as this code is written, it will compile just fine, but it actually has a logic error in it. Because when we call our new overload of add1Passenger, the first thing we're going to do is call that original implementation of add1Passenger. When we transfer control to that implementation, the first thing it's going to do is check to make sure that we have seating available. Well, if there's no seating available, we're not going to increment the number of passengers. But as our new overload is written, we would still increment the total number of checked bags.

=>slides: Pg. 25

So in order for our new overload to work correctly, we need to wrap all this code in a call to hasSeating. So that way we only increment the number of passengers and the total of checked bags as long as there's seating available for the passenger that we're adding. So here in our new implementation of add1Passenger, notice that when we call the other overload of add1Passenger, we don't use any special syntax. We call it just like we were calling that add1Passenger method from anywhere else. So that's an important thing to understand. When one overload calls to another overload of the same method, there's no special syntax. You call that overload just like you would call it from anywhere else. Also notice that the call to the other overload does not have to be the first line in this implementation. Remember when we were talking about constructors, if one constructor called another version of the constructor, that call to the other constructor had to be the first line in the new constructor. No such limitation exists when one overload of a method calls another overload of that same method. So let's take a look at some other overloads of our add1Passenger method we might want to add. And something that's important to understand, just because you're adding an overload of a method, it doesn't have to be adding a brand‑new feature. Some overloads are provided just for convenience.

=>slides: Pg. 26

So let's say we have an add1Passenger overload here that accepts a reference to our Passenger class. Remember that one of the things the Passenger class can be asked is how many checked bags does that passenger have? So our overload that accepts the reference to the Passenger class is simply going to call getCheckedBags on that reference and then call the overload that we just added that accepts the number of bags. So by adding this new overload that accepts a reference to the passenger, we're not adding any new features. We're just making it easier for other code to use our Flight class. Now let's go and add another overload. This overload will actually accept the number of checked bags, as well as the number of carry‑ons. So in this overload, what we're going to do is check to make sure the number of carry‑ons is two or less, and as long as it is, we'll call the overload that accepts the number of bags. Let's add one more overload. This one accepts a reference to a passenger, as well as the number of carry‑ons. And all this overload is going to do is get the number of checked bags from the passenger and then call the overload we just created that accepts the number of bags and number of carry‑ons. So as you can see, as we add these overloads, some of them are adding new features. others are just being provided for convenience. So now in our next section, let's take a look at some code that uses these new add1Passenger overloads.

# Matching Method Calls to Overloads

=>slides: Pg. 27

As you recall, our Flight class now has five overloads of our add1Passenger method. So let's take a look at some code that uses these various overloads. So we'll start out by creating a new instance of our Flight class, and the first thing we'll do is call add1Passenger, passing in no parameters. So, of course, because we're passing no parameters, the compiler is going to use the implementation that accepts no parameters. So we're using the implementation to simply increment the number of passengers.

=>slides: Pg. 28

Then we'll call add1Passenger, passing in an integer. We're passing in a single integer, so we'll use the overload that accepts a single parameter of type integer. And then we know internally this implementation will increment the total number of checked bags and call the original implementation to increment the number of passengers.

=>slides: Pg. 29

So now let's go and create a new instance of our Passenger class. This newly‑created passenger has a single checked bag. We'll call add1Passenger, passing in a reference to that passenger, and then, of course, we'll call the overload that accepts a reference to our Passenger class. Internally, this method gets the number of checked bags for that passenger, calls the add1Passenger overload that accepts the number of checked bags, which internally will call the original add1Passenger implementation to increment the number of passengers. Alright, so let's take a look at a couple more examples of using our overloads.

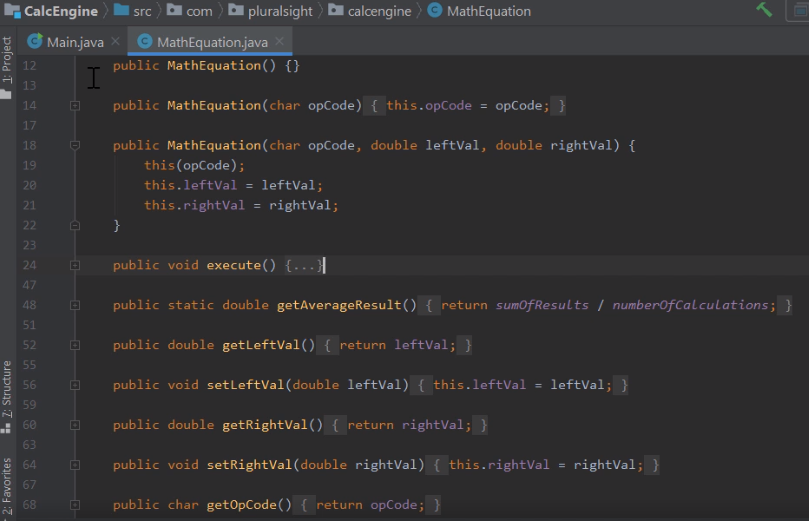
=>slides: Pg. 30

So let's go ahead and create another reference to a passenger. This passenger has two checked bags, so we'll call add1Passenger, passing in the reference to the passenger and an integer. So that will call our overload that accepts the passenger reference and an integer for the number of carry‑ons. Again, internally, we call getCheckedBags on the passenger reference, call the overload that accepts the number of checked bags, as well as the number of carry‑ons. Then internally, that will call the overload that accepts the number of checked bags, then it finally calls our original implementation to increment the number of passengers. So now let's take a look at one example that's a little bit less obvious.

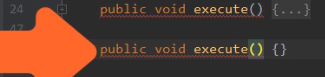
=>slides: Pg. 31

Let's declare a variable here of type short named threeBags with a value of 3. Then we call add1Passenger, passing in threeBags and the literal integer 2. So now, of course, what the compiler's going to look for at first is an overload of add1Passenger that accepts a short and an int. Of course, the problem is there is no overload that accepts a short and an int. So when the compiler can't find an exact match, what it will do is start looking for opportunities to do an automatic conversion. As you recall from our previous course, Getting Started with Programming in Java, there are a number of scenarios where the compiler can do automatic type conversions. One of those automatic conversions is that a short can be automatically converted into an int. So the compiler will look for an opportunity to take advantage of that. So we'll look for an overload that accepts two ints, and we have one of those overloads, the overload that accepts the number of checked bags and the number of carry‑ons. So the compiler will take care of converting that short to an int and making the call for us. Once it makes that call, everything works just as it does in any other scenario. We know internally this implementation will call the overload that accepts the number of bags, which in turn calls our original implementation and increment the number of passengers. Alright, so to get a better understanding of all this, in our next section, let's jump into our CalcEngine project, and we'll start adding some overloads to our MathEquation class.

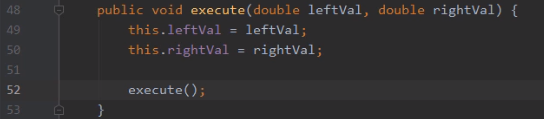
# Using Method Overloading in CalcEngine

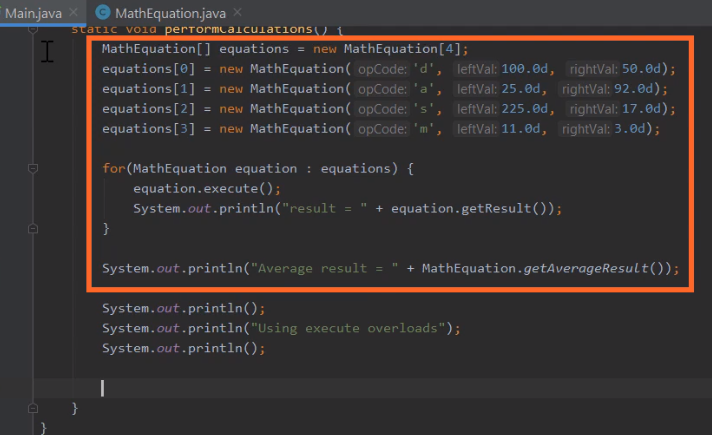


Here we are back in STS, looking at our CalcEngine, and what we want to do now is see how we can use method overloads to improve the usability of our MathEquation class. Now as you recall,' earlier in the course, we added some additional constructors to our MathEquation class. Now we still have our default constructor, but in addition to our to our default constructor, we have a constructor that allows us to specify the opCode, as well as the leftVal and the rightVal; and this constructor works well for scenarios where you know the operation in both values right at the time we're creating our MathEquation instance. And then we added another version of the constructor that just accepts the opCode, and this constructor works well for scenarios where you know the operation you want to perform, but you don't yet know the values. But in situations where we want to use this version of the constructor, the class still has a bit of a challenge, because when it comes time to actually perform the operation, it takes a number of steps to be able to do so. The first thing we have to do is call setLeftVal to set the left value, then we've got to call setRightVal to set the right value, and then finally, we get to call execute. It would be nice to be able to simply provide the left and right values as part of calling execute. So let's add an overload of our execute method that does that.

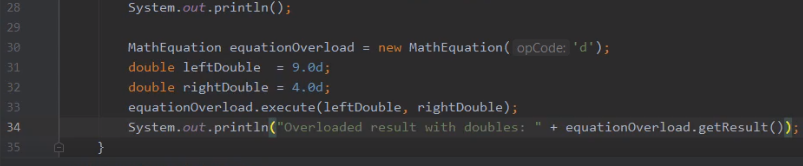


So we'll start out by declaring it as public void execute,

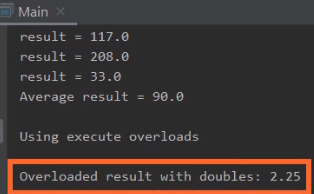


and this overload of execute will accept two parameters of type double, we'll name them leftVal and rightVal, and then once we have our parameters, we can start implementing the overload. So now the first thing we'll do is assign the leftVal parameter to the leftVal field. Then we'll do the same thing for the rightVal, and then once we do that, we'll go ahead and call our original implementation of execute; and now that easily, we have an overload of our execute method that accepts the two values. So let's head back over to our main class. 

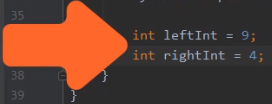
So now over here in our main class, we still have all the code we added earlier in the course. So I add the code to use our new overload after that existing code.



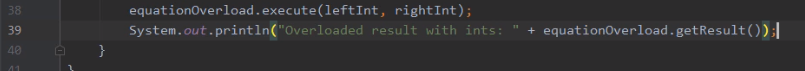
So now the first thing we'll do here is declare a reference of type MathEquation. Let's equation over load. They will create a new instance of MathEquation initialized with an opCode of d. So that gives us a MathEquation instance that knows how to perform division. So after that, let's go ahead and declare two doubles. So now we have leftDouble with a value of 9.0, and rightDouble a value of 4.0. So let's go ahead and call our new execute overload, passing in those values, and then we can go ahead and print out the result. So now with that, we should be all set.



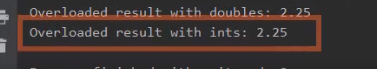
Let's go ahead and run our code and verify that it behaves as we expect; and you'll notice that it does exactly what we wanted it to. When we divide 9.0 by 4.0, we get this result of 2.25. So our overload is working perfectly. So let's go ahead and collapse the window here at the bottom, and now let's try something else. Let's go and declare a couple of integer values.



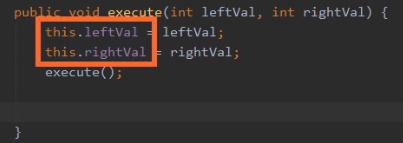
So now we have leftInt with a value of nine, and righInt with a value of four.



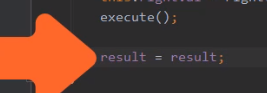
So let's try calling our new execute overload, passing in leftInt and rightInt. So now we're calling our execute overload, passing in ints rather than doubles. Of course the question is, is this going to work? Because our overload expects doubles, it doesn't expect ints. So let's go ahead and run it, and let's see what happens, and you'll notice it does indeed work.



It prints out the value, 2.25, and that's because Java was able to automatically convert our ints to doubles. Remember, as we talked about in our previous course, Java can automatically perform what are known as widening conversions, and converting an int to a double is considered a widening conversion. So Java took care of converting those int values into double values. Then once it had those as double values, was able to call our existing execute overload. You know, something interesting to note, although the code actually ran and printed out the value 2.25, that's not necessarily the best answer. Remember, as we talked about in the previous course, integer division works differently than floating point division. When you perform integer division, there's no fractional portion. So the right answer for dividing the integer 9 by the integer 4 is 2, not 2.25. But because our MathEquation class is always using doubles, it performed the operation as a double. So let's see if we can improve the behavior of our MathEquation class when working with ints. So let me again collapse this window here at the bottom and I'll switch over to our MathEquation class. So now we're over here in our MathEquation class, just below the execute overload that we just added.



So let's go ahead and declare another overload of execute that accepts two ints, and then inside this method we're going to do pretty much what we did in the other overload. So we'll assign each of the parameters to their appropriate fields, and we'll call our execute method. So now with that code in place, we will perform the operation, but the result wouldn't be any different, because remember, our leftVal and rightVal fields are still doubles, so the actual operation will be performed using doubles. So somehow we need to convert the value that's inside a result to the appropriate value for an integer.



So let's start out by assigning a result field back to itself. Now of course, assigning result back to itself isn't really going to change anything.



But here on the right hand side of the equal sign, let's cast that result to an int, and by doing that, we can actually get rid of the fractional portion. Remember that result as a double, we'll start out with a fractional portion, when we cast a result from a double to an int, the value gets converted to an int, which would then drop the fractional portion. We then take that int value and assign it back to result, but now will no longer have the fractional portion. So adding this one line of code will allow us to display the appropriate result for dealing with integers. So let's go ahead and run our code again, and let's see what happens.



So you'll notice now, when we run our code with ints, we get the appropriate answer, which is two, but the code that uses doubles still shows the right result for doubles, which is 2.25, and that shows us that our code is using the right overloading each of those situations. Now, in general, the way Java goes about choosing the appropriate overload is pretty intuitive. It's generally going to choose the overload that you expect, but the underlying logic to make that selection is actually fairly complex.



If you'd like to know the details of how Java goes about making that selection, I have a URL on the screen right now that will point you to more information about the underlying logic that Java follows to select the appropriate overload. All right, so now in our next section, let's take a look at variable length parameter lists.

# Variable Number of Parameters

=>slides: Pg. 32

Let's look, again, at our Flight class. Now as you recall, up until now, any time you want to add passengers to our Flight class, we have to add them one at a time because the only way we have to add a passenger is with our add1Passenger method. It might be nice to add another method, addPassengers, that allows us to add multiple passengers at one time. So we'll declare this addPassengers method to accept an array of type Passenger. Now, of course, if we're going to add more than one passenger at a time, we need to have another hasSeating method that accepts the number of passengers that we want to add. What our hasSeating method will do will take the current number of passengers, add in that count of passengers you want to add, and as long as we have enough seats for all those passengers, we'll go ahead and return true. So in our addPassengers method, we'll call hasSeating, passing in list.length. That's the size of our array. As long as that returns true, we'll increase the number of passengers by the length of the array. And as part of adding these passengers, we want to add in the appropriate number of checked bags. So we can loop through the array one by one, and for each passenger, we'll call getCheckedBags, adding that to the totalCheckedBags. So now we have this addPassengers method that will allow us to add multiple passengers at one time.

=>slides: Pg. 33

So let's see what it's like to use our addPassengers method. So we'll create a new instance of our Flight class, and then let's say we want to add two passengers, Luisa and John. Luisa has one checked bag, John has two. So we want to call addPassengers on our Flight class, and what we want to pass in is the references, luisa and john. But we can't simply pass these references in because addPassengers is expecting an array. So in order to pass those two values in, we have to explicitly wrap them in a new instance of a passenger array. So then if we want to add three more passengers, Harish, Julie, and Ashanti, we're going to have to do that same sort of thing, call addPassengers on our Flight class. We can't pass in just harish. julie, and ashanti. We, again, have to explicitly wrap them in a new instance of our passenger array. So although it's nice that our addPassengers method allows to add multiple passengers at one time, it's a bit of a burden to use the method because we're always having to create these arrays to add the passengers.

=>slides: Pg. 34

Now if we think about what makes our addPassengers method hard to use, it's the way we've declared the parameter. We've indicated that we always have to pass the list of passengers as an array. It'd be nice if we could simply declare this method to accept a variable number of passenger references, and it turns out that we can do that. The way we do it is by indicating the type of the parameter followed by an ellipse. So where we declare the type of our parameter, rather than explicitly saying it's a passenger array,

=>slides: Pg. 35

we're going to say Passenger..., and that indicates that our method accepts a variable number of passenger references. It will accept 0 or more passenger references. Now any time you want to indicate that the parameter accepts a variable number of values, it has to be the last parameter for the method. The method is allowed to have other parameters, but only the last parameter could be variable length. Now one of the things that's really cool about variable length parameter lists is the method itself receives that list as an array.

=>slides: Pg. 36

So what that means is we can implement our addPassengers method just like we did previously because our parameter named list is still an array of type Passenger. So we can call hasSeating, passing in list.length. We can still increase the number of passengers by list.length, and we can still loop through that list to get the number of checked bags for each passenger. Where the convenience of a variable‑linked parameter list comes in is for the callers of the method.

=>slides: Pg. 37

So if we look at our code here where we call addPassengers, here we call addPassengers, passing in luisa and john,

=>slides: Pg. 38

we no longer need to create an array in order to call the method. In the same way, when we call addPassengers for Harish, Julie, and Ashanti, again, we no longer need to create that array.

=>slides: Pg. 39

We can simply provide the list of values we want to pass into the method. So variable‑linked parameter lists are a great solution for situations like this. As implementers of the method, we have the convenience of working with the values as an array. But for the caller of the method, all they have to do is pass in the individual values themselves, and the compiler takes care of the details of turning those values into an array.

# Summary

=>slides: Pg. 40  
To wrap up, here are some of the key things you'll want to remember from this module. Remember that when we pass objects as parameters those objects are passed by‑reference. And what that means is the entire object is not copied into the method. Instead, the method receives a copy of the reference, which points back to the original object. And this has some important implications about changes within a method. Remember that if a method tries to make a change to the reference itself, in other words make that reference point to a different object, although those changes are visible inside the method such changes would not be visible once we exit the method. But that's okay because in most cases if a method's going to make modifications it's going to make modifications to the referenced object. And if the method makes changes to the object itself that's fine because those changes will remain visible even when we exit the method because remember the method is still operating on the original object instance.

=>slides: Pg. 41  
Then we looked at overloading. Remember, overloading allows a class to have multiple versions of a method or a constructor. Now remember that the way those individual versions are identified is by having a signature, and each one must have a unique signature. Remember that a method's signature is made up of three parts. It's the name of the method, along with the number of parameters it has, as well as the data type of each parameter. So as long as that combination of name, number of parameters, and type of each parameter's unique the method is considered to have a unique signature. =>slides: Pg. 42  
Then we finished up. We looked at variable length parameter lists. Remember that variable length parameter lists allows the caller of the method to pass in 0 or more values. The way we indicate that the parameter can accept multiple values is by placing an ellipse after the parameter type. Remember that only the last parameter of a method is allowed to be variable length. Now remember that all the values that are passed are received into an array. So within the method itself all we have to do to process the values is process the contents of that array. Alright, that wraps up this module. In our next module we'll start looking at class inheritance.

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