3. Implementing Class Constructors and Initializers

Contents

[1. Introduction 1](#_Toc14194)

[2. Class Initial State 1](#_Toc6309)

[3. Field Initializers 1](#_Toc3318)

[4. Constructors 1](#_Toc16183)

[5. Constructor Chaining 2](#_Toc13829)

[6. Constructor Visibility 2](#_Toc14169)

[7. Adding Constructors to MathEquation 2](#_Toc5843)

[8. Initialization Blocks 2](#_Toc20463)

[9. Summary 2](#_Toc22578)

1. Introduction  
   =>slides: Pg. 1

Welcome to our next module, Implementing Class Constructors and Initializers. Throughout this module we'll be looking at how we can control the initial state of a class when it's first created.

=>slides: Pg. 2

Now, in order to do that, the first thing we need to understand is, what is the default initial state of the fields within a class when that class this first created? From there, we'll look at some of those field initializers, which allows us to assign our own values to a field. We'll then look at class constructors. Now we touched on constructors quickly in the previous module, and in this module we'll going to look at it much more closely. And one of the interesting things about constructors, is a class can have multiple constructors and even has the option of chaining constructors together. So that when one constructor is used, that constructor automatically calls another constructor and runs the code in that constructor as well. We'll take a look at constructor visibility. We'll talk about why, in certain cases, you may want to limit the visibility of some of the constructors within your class. We'll then look at initialization blocks. Initialization blocks allow us to have a code that automatically runs no matter which constructor is used. And then, finally, with so many options for how we establish the initial state of a class, we want to be sure that we understand how those different options relate to one another, and in what order each of the options are performed.

1. Class Initial State  
   =>slides: Pg. 3

Whenever we create a new instance of a class, that newly created object is expected to be in some useful state. So to help us with that, Java will give you to the fields within the class the default value. But these default values are often not enough. In many cases, we need to take specific action to properly establish the initial state of the fields within our class. Now there's a couple ways we can do that. One option is to simply set the field values directly. Another option is to have code that executes automatically as part of creating the class instance.

=>slides: Pg. 4

And as you recall, the state of a class is represented by its fields. So when a new class instance is created, each of those fields will be given a default value. Now the specific value will depend on the type of the field itself, but in general, each field is set to whatever its zero value is. Now in the case of numeric field types, things like byte and short, int and long, as well as float and double, they are literally set to zero. But remember that all types are not numeric. For example, we have the char type, where the char type contains a character value. So the default value for any field that's a char type is the character that's represented by all zeros. Then for our boolean fields, boolean fields default to a value of false. But oftentimes our classes are going to continue with a number of reference type fields as well. In the case of reference types, each field is set to a null value by default. Which of course means that field is not set to reference an actual object instance.

=>slides: Pg. 5

Now, as we've mentioned, we're not limited to accepting the default state that Java provides. We can actually establish our own starting state of our classes. And in general there are three ways to do that. One way is to use what's known as a field initializer, and a field initializer allows us to give a field a value as part of the field declaration itself. Now, in addition to that, we have constructors and we also have initialization blocks. And both of these give us the option of having code that runs automatically as part of the object instance creation. Now, throughout this module, we'll look at all these options in more detail. So in our next section, let's start out with a closer look at field initializers.

# Field Initializers

=>slides: Pg. 6

Now oftentimes, we want our fields to have a value other than the default value, and that's where field initializers come in, because field initializers allow us to specify a field's initial value as part of the field's declaration itself. So for example, I have a class here named Earth, and I want to have a field circumferenceInMiles. Now at this point, circumferenceInMiles has a default value of 0, and I'd like to set it to the appropriate value for circumferenceInMiles. And I can do that right here where I declare the field by simply assigning the value to the field. So now when a new instance of our class is created, circumferenceInMiles will have that value of 24,901. But now when we set our field values, we're not limited to simple assignments. You can also include equations. So if I want to have a field here circumferenceInKms, well, I know I can convert miles to kilometers by multiplying by 1.6. So here I'm taking the literal value of 24,901 and multiplying that times 1.6. Now although that'll give me the correct answer, it's not really clear what I'm doing here. So it would be nice if I could reference that other field, and I can indeed do that. When I set the value of one field, I can use other fields in determining its value. =>slides: Pg. 7

So here where we set circumferenceInKms, rather using that 24,901 literal, we'll use our field circumferenceInMiles. Now when we do this calculation and multiply circumferenceInMiles by 1.6, the result will be a double, which will include a fractional portion. But now currently, we're simply casting it to a long, which would drop any fractional portion. It would be nice if we could round the result.

=>slides: Pg. 8

Well, it turns out we can do that because we're also allowed to include method calls when we make the assignment. So here when I set my circumferenceInKms, rather than doing the cast to long, I'm going to call the Math function round, which will then round the results of that equation. So as you can see, field initializers give us an easy way to set the initial values of our fields. But in some cases they're not enough. In some cases, we need to actually execute code. So in our next section, let's take a look at constructors.

# Constructors

=>slides: Pg. 9

Now as you recall, we talked about constructors briefly in our previous module. A constructor contains code that runs whenever we create a new instance of our class. Now remember the constructors have to have the same name as the class that contains them, and constructors don't have a return type because a constructor doesn't return a value. It simply contains code that runs when we create a new instance of our class.

=>slides: Pg. 10

So let's take a look at our Flight class that we worked on in a previous module where the Flight class contains two fields, passengers and seats, and we also gave it a constructor. Now the constructor has set values for those two fields. But now we've actually learned a lot since we originally wrote this code. =>slides: Pg. 11

For example, here we set passengers equal to 0. We know that that code is not necessary because an int field automatically has its value set to 0. And then here we set our seats to 150, but we don't need that code either because we know a field initializer will allow us to assign the value to the field as part of the field declaration itself. So it turns out that we didn't really need to do any of the work in that constructor that we had been doing.

=>slides: Pg. 12

Now it turns out that each class does need to have at least one constructor, but we don't have to explicitly write that constructor. If we don't provide a constructor of our own, Java will provide one for us. The one Java provides will simply be an empty constructor.

=>slides: Pg. 13

So let's take a look at another class here. Let's say we have our passenger class, and the passenger class represents a passenger on a flight. Now our passenger class will have a checkedBags field to keep track of the number of bags that passenger has checked. And then we'll also include a freeBags field because some passengers will be allowed to bring some bags with them for free. We'll assume we have getters and setters for both of those fields. Then we'll also another field, our perBagFee field, which will indicate the cost for each bag the passenger brings beyond the number of freeBags. Now as our passenger class is currently written, it does not explicitly contain a constructor.

=>slides: Pg. 14

But we can still create instances of our passenger class, and the reason that we can do that is that Java will automatically inject a constructor for us. Again, that constructor won't do anything, but it does fulfill that requirement of having at least one constructor. And then, of course, once our passenger instance is created, we're free to interact with that instance so we can do things like set the number of checked bags.

=>slides: Pg. 15

Now although a class has to have at least one constructor, it's not limited to having just a single constructor. It can have as many constructors as it needs. The thing is is each constructor has to have a unique parameter list. So that means the constructors either need to differ in the number of parameters they contain or they need to differ in the data types of those parameters. Now this information about the number of parameters and their data types is often what we call the signature. So we commonly will say that you need to have constructors with differing signatures. And when we say that, all we mean is that each constructor has to have its own unique parameter list.

=>slides: Pg. 16

So now back here in our passenger class. Now again, our passenger class is relying on that constructor that Java has provided for us automatically. So now let's say we want to add another constructor that allows us to specify the number of freeBags this passenger is allowed to have as part of the object instance creation. And inside this constructor, we're simply going to set our freeBags field to contain the value we pass in to this constructor. So now with this constructor in place, when we create a new passenger, we can pass in a value. So in this case, we're passing in the number 2. So when we create this new instance of the passenger class, the freeBags field in that instance will be set to 2. Now once we create this passenger instance, we can interact with it like any other instance of our passenger class. =>slides: Pg. 17

But adding this constructor will actually break our existing code. We'd no longer be able to create an instance of our passenger class, passing in no parameters at all. And that's because the code that now fails relies on this constructor that Java generated for us automatically. Java only automatically creates a constructor if your class contains no constructors at all. Once we provide one constructor,

=>slides: Pg. 18

we're responsible to provide all of the constructors. Now keep in mind we're not required to replace that constructor that Java was generating for us automatically. If we want to require that whenever a passenger class instance is created, we have to pass in the number of freeBags, we're allowed to do that. We can keep just this one constructor we have here.

=>slides: Pg. 19

But if we still would allow instances of our class to be created that pass in no arguments at all, then we're responsible to add that constructor as well. So now we've explicitly written a constructor that doesn't accept any arguments. Now the constructor happens to be empty. It doesn't have to be, but it's allowed to be. But that would then allow the creation of our passenger class without passing any arguments. And just as a matter of terminology, when we have a constructor like this that doesn't accept any arguments, we call that constructor the default constructor. Okay, so now in our next section, let's continue our discussion of constructors, and we'll take a look at how we can chain constructors together.

# Constructor Chaining

=>slides: Pg. 20

With classes able to have multiple constructors, it makes sense that one constructor might want to leverage the code that's contained in another constructor, and we can do that. One constructor is able to call another constructor. And oftentimes we call this process of one constructor calling another chaining the constructors together. Now when one constructor calls another constructor, the call to the other constructor must be the first line of the new constructor. And the way we make that call is by using our this keyword followed by a parameter list, which means that one constructor can actually pass in values to the other constructor.

=>slides: Pg. 21

So let's again look at our passenger class. As you recall, one of the constructors in our passenger class accepts a value for freeBags. And in that constructor, we set our freeBags field. So let's add another constructor here that accepts two integer values, one for freeBags and one for checkedBags. So now inside this constructor, we're going to want to set the appropriate fields. So we'll set our freeBags field, and we'll set our checkedBags field. Now although this code is perfectly valid, the first line here actually duplicates the code that we already have in the constructor that accepts only the value for freeBags.

=>slides: Pg. 22

So rather than duplicating that code, we can simply call that other constructor. So now what'll happen if the user creates an instance of the passenger class, passing in two integer values, those values will be received by the constructor as parameters for freeBags and checkedBags, the first thing this constructor does is call the constructor that accepts the value for freeBags. We pass in the value for freeBags. We run the code inside that constructor, which then sets our freeBags field. And then when that code completes, we return it back to the original constructor, running the code inside there to set our checkedBags field. So by chaining the constructors together, it allows each constructor to focus on doing a particular job well. Also, this idea of keeping the code centralized in each individual constructor makes it easier to evolve our code over time.

=>slides: Pg. 23

So again, looking at our passenger class, let's look just at the constructors that we have here. So we have one constructor again that accepts a number of freeBags. The other constructor accepts freeBags and checkedBags. And let's add one more constructor that accepts a value for the perBagFee. And the perBagFee is the amount of money the customer has to pay for any bags they check beyond the number of free bags that they're allowed. So let's say as part of designing our passenger class, we're given business rules to tell us that the perBagFee is dependent upon the status of the passenger. So elite passengers pay a lower bag fee than standard passengers do. And a passenger's status is indicated by the number of free bags that they're allowed. So what we can do is go up to our constructor that receives a value just for freeBags, and let's add that appropriate business logic there. So now, inside this constructor, we have code that checks the number of free bags. And a passenger is considered an elite passenger if they're allowed more than one free bag. So those customers pay a discounted bag fee. So if a customer is allowed more than one free bag, their perBagFee is only $25, but all other customers have to pay a perBagFee of $50. So what our code here will do is run that logic to determine the appropriate perBagFee and then pass that value into the constructor that accepts the perBagFee. So now if we create a new instance of our passenger class, passing in two integer values, we'll again call the constructor that receives parameters for freeBags and checkedBags. We then call the constructor that accepts just the value for freeBags. We pass in the value for freeBags. That constructor then does the business logic to determine the perBagFee, then calls the constructor that receives a value for perBagFee. Then inside that constructor, we set our field for perBagFee. When that constructor completes, we run the code inside the constructor that received the value for freeBags that sets our freeBags field. And when that code completes, we go back to our original constructor and set our checkedBags field. So as you can see, by chaining the constructors together, it allows us to easily evolve the code related to creating the instance of our passenger class because each constructor was focused on a certain part of the work that had to be done based on the information we had about the passenger instance we were creating. All right, so now in our next section, let's take a look at the issue of constructor visibility.

# Constructor Visibility

=>slides: Pg. 24

As we add constructors to our classes, we should be thinking about what visibility each constructor should have, because some constructors should probably not be public. And the reason we make some constructors nonpublic is because we want to limit what code can have access to certain phases of the instance creation process.

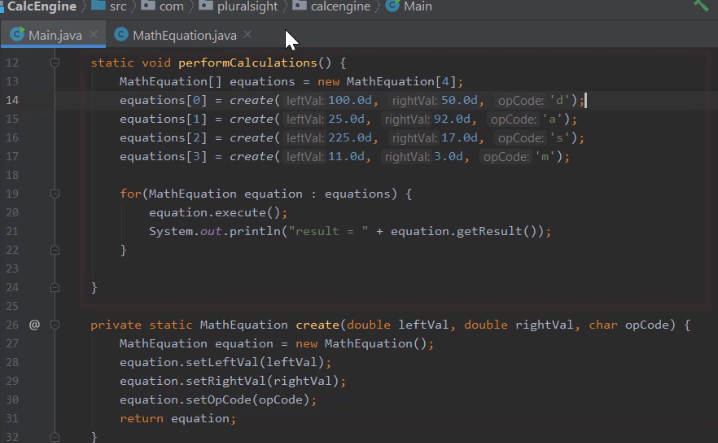
=>slides: Pg. 25

So let's again look at our Passenger class, and let's look at the list of constructors that we now have. We have a constructor that accepts no parameters, we call that our default constructor, we have a constructor that just accepts a value for freeBags, a constructor that accepts a value for freeBags and checkedBags, and a constructor that accepts a value for the perBagFee. Now remember that our perBagFee is meant to be part of the business logic for our Passenger class, so we probably don't want code that we don't control being able to use this constructor, because by having this constructor as public, we run the risk of code that we don't control, creating an instance of a passenger, we'll call him cheapJoe, and this code determines that the perBagFee should just be 1 penny. And it's unlikely that that's consistent with the business logic we want to use with our Passenger class. So we really want to restrict what code can determine the perBagFee for a passenger. So rather than making this constructor public, let's go ahead and make it private.

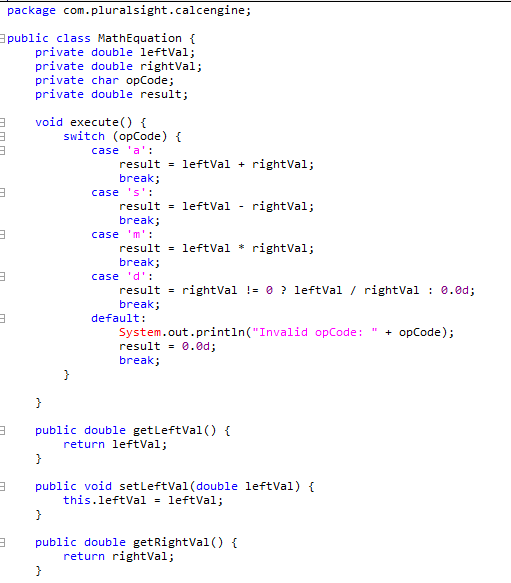
=>slides: Pg. 26

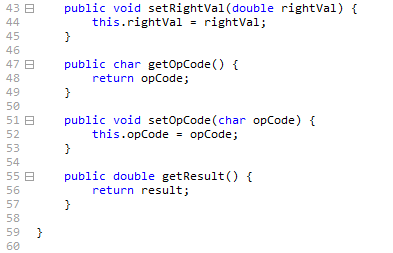
So now by making that constructor private, only code within the Passenger class can access it. So the attempt to create an instance of passenger from outside the Passenger class using this constructor will no longer compile because that constructor is not accessible outside of the class. But now the constructors within the class can still access it, so we want to create another passenger instance here, geetha, and we pass in a single integer value, which means we'll create this passenger instance with a constructor that accepts the value for freeBags. Remember that this constructor has the logic that determines the perBagFee based on the number of freeBags, and this constructor can still chain to that constructor that accepts the perBagFee. Now, let's look at another use case. We'll create another Passenger here, santiago, and this uses a constructor that accepts two integer values, which means we're passing in a value for freeBags and a value for checkedBags. This constructor calls the constructor that accepts the value for freeBags, and again, this constructor has the logic for determining the perBagFee and calls the constructor that receives the perBagFee. So as you can see, making certain constructors nonpublic limits what code has access to specific parts of our instance creation process. All right, so to help us get a better understanding of what it's like to work with constructors, in our next section, let's jump back into our code, we'll add some constructors to our MathEquation class.

# Adding Constructors to MathEquation

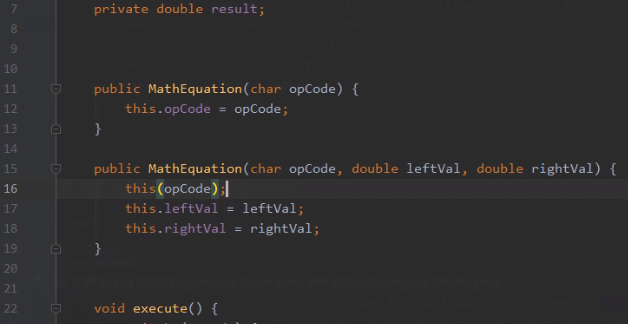


Here we are back in STS, and what we want to do now is add the appropriate constructors to the MathEquation class that we added to this project in the previous module. As you recall, the way the code works is we have this method here, performCalculations. It creates an array of type MathEquation and then goes through and creates a series of MathEquation instances that we can then run and display the results. Now remember the way we currently create each instance is down here in this create method within the Main class. So the Main class is responsible to go ahead and create an instance of the MathEquation class. And each time it creates an instance, it's using the default constructor that Java provided for us. Then we go through and set the leftVal, rightVal, and opCode fields and then return back that reference to the new MathEquation instance. Now one quick thing to note, I have made one small change to this code since our previous module. As we initially wrote this code, we allowed the Main class to directly access each of the fields within the class. And as we've talked about, that's not really a good best practice.





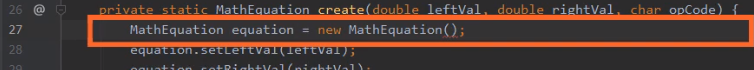
So I've actually updated the MathEquation class to have the appropriate getters and setters for each of our fields. You can see here where I'm using the setters to set each field value. All right, so how can we use constructors to improve this code? Well right now, the entire burden of setting up the MathEquation instance to be in an appropriate usable state is being placed on the code that uses our MathEquation class. But with the appropriate constructors in place, we can put that work inside the MathEquation class itself.



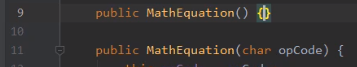
So let's head over here to our MathEquation class. And then here in a MathEquation class, we want to think about what are the appropriate constructors to add? Well, I think one appropriate constructor would accept just the opCode. That would be useful in a situation where we know what operation we want to perform, an add, subtract, multiply, or divide, but don't yet have the values. So let's go ahead and add a public constructor that accepts the opCode as a parameter. Then here inside this constructor, we'll go ahead and set our opCode field. So now we're all set with this constructor. And we also know we need another constructor for our current use case. We need a constructor that accepts the opCode, leftVal, and rightVal. So now in this constructor, we'll start by setting are leftVal and rightVal fields. And then for our opCode field, rather than setting the value here, let's chain to the constructor that accepts the opCode, passing in our opCode. And now with that, we can create an instance of our MathEquation class, passing in the opCode, leftVal, and the rightVal or just passing in the opCode. Now before we do anything else, let's go ahead and run our code and verify that everything still works.



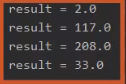
And you notice that when we do that, we get an error message, and it says that no suitable constructor is found, and the code it's showing us is our create method over here in our Main class.



Well remember that this method relies on our default constructor, and that default constructor is being provided for us by Java because previously our class didn't have any constructors. But once we add one constructor, we're responsible to provide all the constructors.



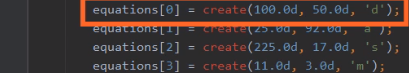
So let's head back to our MathEquation class. Let's go ahead and add default constructor that doesn't do anything. So now with that, our class explicitly contains a default constructor. So let's go ahead and run the code again.



And you'll notice that now it runs properly. So with that code all working, let me go ahead and collapse the run window, and we'll head back over to our Main class. So now we're back over here in our Main class.



Let's take a look at this create method. If you think about it, we don't really need this create method anymore because all the work the create method is doing is now encapsulated in one of the constructors for our MathEquation class. So I'll go ahead and highlight this method, and then I'll go ahead and delete it. So now up here where were previously calling the create method, we'll instead use our constructor.



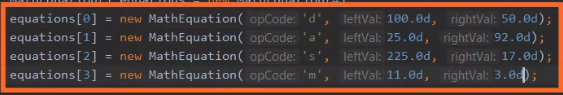
Let's start out with the zeroth element of our array where we divide 100 by 50.



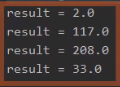
Rather than calling create, we'll simply say new MathEquation, passing in the parameter values. So now as we're creating this new instance of MathEquation, we want to be sure we have the parameters in the right order. The Main class's create method accepted the opCode d as the last parameter. In our constructor, that should be the first parameter.



So let me go ahead and move our d value to the appropriate position. So now we're properly creating an instance of our MathEquation class,



so let's do the same thing for each of the other three instances. And so with that, we should be all set. We create each instance of our MathEquation class, passing in the appropriate opCode, leftVal, and rightVal. So when we go ahead and run this, we'll verify that everything works as it should.



And as you can see, we now display the appropriate results. And the key thing we've done here is encapsulate the details of establishing the initial state of the class within the class itself. We're no longer burdening the code that uses our class with those kind of details. All right, so now in our next section, let's take a look at something known as initialization blocks.

# Initialization Blocks

=>slides: Pg. 27  
Now another option for incorporating code that automatically runs whenever a class instance is created is known as initialization blocks. Initialization blocks allow us to have code that's shared across all constructors. Basically it's a block of code the automatically runs no matter what constructor is used to create a new instance of the class. Now initialization blocks do not accept any parameters. They are simply a block of code. And the way we specify them is having opening and closing brackets containing the code outside of any method or constructor. Now a class can have zero or more initialization blocks. But understand, if you provide more than one initialization block, all of the blocks will always run. Basically, when we create in the instance of the class, the block that's closest to the top of the source code file runs first, the one below that runs next, and so on until all of the initialization blocks have been run.

=>slides: Pg. 28  
So let's take a look at where initialization blocks can be helpful. So let's again look at our Flight class. As you recall, our Flight class has a passengers field. It also has a seats field that we've initialized to be 150. So let's look at adding a couple more fields to our Flight class that we can use to identify an individual flight. So we'll have a flightNumber field, which we'll use in cases where a flight has a very specific flight number, flight 100, flight 450, that sort of thing. Other flights we'll identify with a more general value known as the flightClass. Is this a class A flight, is this a class B flight, that sort of thing. Now in addition to identifying the flights more specifically, we also want to keep track whether a particular seat on that flight is available. So what we'll do here is have a boolean array, isSeatAvailable, that will tell us if a particular seat is still available. Now remember that a field that's a boolean type defaults to false. So when we declare this array isSeatAvailable, all the elements in it are going to start out as false, and that's not what we want. We want to start out with the value as true, indicating that all the seats are available. Now one way we could initialize all the elements of the array to start out with a true value would be to go ahead and provide our default constructor and, inside that default constructor, loop through and set each element of the array to have a value of true. And that would certainly work well enough. We would have important implications on other constructors we add to the class.

=>slides: Pg. 29  
So let's look at some of the other constructors we might want to add to this class. Well, remember that when we create some flights, those flights are going to be identified by a flight number. So we'll have a constructor that receives a flight number. And inside that constructor, we'll set our flight number field. But at this point, the code in the default constructor will not run. The only way to run the code into the default constructor is to actually chain to that default constructor. So that means we want to add another constructor that allows us to create a flight identifying its flightClass. Inside that constructor, we'll of course have to set the flightClass field, but we'll again have to chain to the default constructor. We'd have to do the same sort of thing for any other constructor we add to the Flight class. If we want to be sure that the isSeatAvailable array is properly initialized, we're always going to have to explicitly chain to that default constructor.

=>slides: Pg. 30  
So let's see how initialization blocks can help us with that. Let's head back here to the top of our Flight class and down here where we have our default constructor. Let's get rid of that default constructor, and instead we'll have an initialization block. And you'll notice the initialization block has that same code that we had previously. We're simply looping through the isSeatAvailable array, setting all its members to true. But notice our initialization block doesn't have a name or anything like that. It simply has the code contained within opening and closing brackets. And it's this syntax that creates the initialization block. And the code in the initialization block will now run automatically no matter what constructor we use to create an instance of our class.

=>slides: Pg. 31  
So if we take a look at adding those other constructors again, we have the one constructor that accepts a flightNumber. The only thing we have to do in here now is set that flightNumber field. The code contained within the initialization block will automatically run. Same thing for the constructor that accepts a flightClass. We simply do the core work, we just set our flightClass field, the code in the initialization block will again automatically run. Now something to keep in mind. Initialization blocks do not replace the default constructor. If we want to allow instances of our class to be created without passing either a flightNumber or flightClass, then we need to go ahead and include a default constructor. And then if we have code that creates an instance of our Flight class using this default constructor, the code contained with the initialization block will still run just like it does in the other constructors.

=>slides: Pg. 32  
Now throughout this module, we've learned a number of mechanisms for establishing the initial state of our classes. So let's just take a quick look at what order that initialization work actually occurs. So now when we create a new instance of a class, the first thing that occurs is the field initializers. So basically, all the fields that specify their values using a field initializer have their values set before any of our other work occurs. Once the field initializers are done, then the code in the initialization blocks run. So that code is all run before any constructor code is run. Only after the initialization blocks are complete do we run the code in the specified constructor. And this order is followed each time we create a new instance of a class. So this gives us an orderly and predictable way to establish the initial state of our class instances.

# Summary

=>slides: Pg. 33  
To wrap up, here are some of the key things we want to remember from this module. Remember that whenever we create a new instance of a class, that class instance is going to be in some initial state. That initial state is expected to be useful. We wouldn't want a new instance of our class to be loaded up with garbage values. So to help us with that, Java does provide default values for our fields. For example, numeric fields start out with a default value of 0, whereas fields that are reference types start out with a default value of null. But we can also provide specific values for our fields using field initializers, and that allows us to assign a value to the field as part of the field's declaration. Now it can be a simple assignment of our literal value, or it can be something more involved. Field initializers can include equations, other fields, even method calls.

=>slides: Pg. 34  
Then, after field initializers we looked at constructors, and constructors give us code that can run during object creation. And constructors can accept 0 or more parameters. Remember, a constructor that accepts no parameters is called the default constructor. But keep in mind a class can have multiple constructors, and the specific constructor that's used is based on the parameters that are passed when the new class instance is created. Now remember that if a class has multiple constructors, one constructor can call another constructor. Now if we want to do that, the call to the other constructor has to be the first line of the new constructor, and when we make that call we can pass parameters, which means that one constructor can pass data values to another constructor. And remember that constructors don't all have to be public, our classes can have nonpublic constructors, and the reason we do that is to limit which code can perform specific types of instance creation. Some phases of creating an instance of our class should not be accessible outside of the class itself.

=>slides: Pg. 35  
And then we finished up with initialization blocks. Initialization blocks also give us code that's run during object instance creation. So remember that initialization blocks are not tied to any specific constructor. The code within initialization blocks runs no matter which constructor is used to create the class instance. And also keep in mind, the initialization blocks cannot receive parameters. There's simply a block of code that runs when you create a new instance of the class. Alright, that wraps up this module. In our next module, let's take a look at something known as static class members.

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