1. Understanding Java Classes and Objects

Contents

[1. Introduction 1](#_Toc11424)

[2. Declaring Classes 1](#_Toc31906)

[3. Declaring the MathEquation Class 3](#_Toc17929)

[4. Using Classes 5](#_Toc22017)

[5. Creating an Array of Classes 7](#_Toc22592)

[6. Using the MathEquation Class 9](#_Toc5886)

[7. Encapsulation and Access Modifiers 11](#_Toc13949)

[8. Special References: this and null 12](#_Toc74)

[9. Field Accessors and Mutators 14](#_Toc14268)

[10. Summary 15](#_Toc27083)

# Introduction

=>slides: Pg. 1-2

So now in this module, we're going to focus on the fundamentals of working with classes and objects. So to do that, the first we'll look at is how we declare classes within our applications. We'll then see the different kinds of members a class can have and how we add those members to our classes. We'll then see how to work with objects, which is simply a case of creating instances of classes. We'll then see how we protect the details of our classes by understanding encapsulation and how access modifiers make that possible. And then we'll finish up by looking at how we protect the data within our classes by looking at field accessors and mutators.

# Declaring Classes

=>slides: Pg. 3

Java is an object‑oriented language. What objects allow us to do is encapsulate data and the operations on that data into a single entity, then expose semantics for working with that entity. And with objects, the storage and manipulation details can be hidden, and that's a really important concept because it allows the object to expose the operations that can be performed without burdening the user of the object with the details of how those operations are performed. And when used correctly, this can simplify building complex applications. Because all the complexity is packaged up in the object, the user of the object does not have to get involved in those complexities.

=>slides: Pg. 4

Now before we can have an object, we first have to have a class because a class is a template for creating objects. So we'll start out by declaring a class. We do that using the class keyword followed by the class name. So if I want to have a class named Flight, I'll use the keyword class followed by the word Flight. When we create class names, we follow the same basic rules is with variable names. We use just letters and numbers, but we always capitalize the name of a class. If a class is composed of multiple words, each word within that class name is also capitalized. Now the body of the class is contained within brackets. So we'll have an opening bracket and a closing bracket, and all members of the class will be contained within those brackets. Now when we create our class, we're of course going to put it into a source file, and that source file name is normally required to have the same name as the class. So I have a class named Flight. It's going going to a file named Flight.java.

=>slides: Pg. 5

Now classes are made up of both state and executable code. We use members to represent these. So one kind of member are fields. Fields store the object's state. So those are the data within the class.

=>slides: Pg. 6

So now if we look at our class Flight, some state it might have is maybe the number of passengers. So we have a field here named passengers whose type is int. We might also have state related to the number of seats. So we have another int field here, this one named seats. So together, passengers and seats represent the state of this class.

=>slides: Pg. 7

Now classes can also have methods. And as you recall from the previous course, methods are executable code. Now within a class, there are executable code that can manipulate the state of the object, as well as perform operations related to the class.

=>slides: Pg. 8

So our Flight class might have a method, add1Passenger, that's responsible to add one more passenger to the flight. So here inside of add1Passenger, the first thing we'll do is check to make sure that we haven't already filled the seats. So as long as the passengers are less than the number of seats, then we can go ahead and add one more passenger. So this add1Passenger method was able to view the state of the class, in other words it could see its fields, as well as manipulate the state, in other words it changed the value of the passengers field.

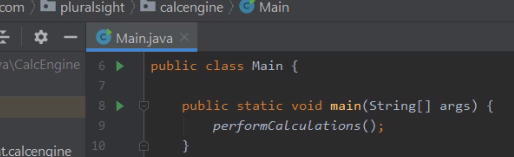
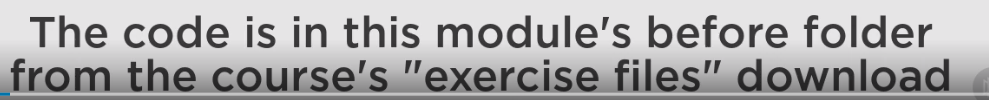
=>slides: Pg. 9

So now we have one more kind of member, which is a constructor. Constructors are also executable code, but they're executable code that automatically runs when an object is created. And what we normally use them for is setting the object's initial state.

=>slides: Pg. 10

So here in our Flight class, we'll add a constructor. Notice a constructor looks a lot like a method, but it doesn't have a return type, and the constructor name is the same name as the class. Now again, the code within this constructor will run automatically when an object is created and, again, will normally set initial state. So here within our Flight class, we'll start out with an initial state of the flight having 150 seats available and no passengers. Okay, so now that we understand how to declare a class, in our next section, we'll jump into some code, and we'll create a class from scratch.

# Declaring the MathEquation Class



So here we are now with some source code opened up in our Java IDE. The IDE we're using is STS. STS is actually one of the most popular of the Java IDEs. Now the project I have open here is a simplified version of the app that we created before , and this app currently does a fairly simple task. It actually works on two numeric values that we call a leftVal and a rightVal, then uses a one‑character code called an opCode to indicate an operation. And each of the opCodes are simply the first letter of the operation name. So the opCode d means divide, a means add, s means subtract, and m means multiply. So we have a leftVal of 100 and a rightVal of 50 with an opCode of d. That means divide 100 by 50, which would give us a result of 2. Now the way this application is currently built, it's using a very procedural approach, meaning that all the data and all the methods are really kind of independent, and it's up to the application to lace them all together.



So when our application first starts up, we go here into our main method.



Our main method calls the method performCalculations. PerformCalculations uses a bunch of parallel arrays. Parallel arrays simply means that each element in each array corresponds to the same element in the other array. So if we look here at our leftVals array, its first element 100 goes with the first element of the rightVals array, 50. That goes with the first element of the opCodes array d. So again, that would divide 100 by 50 and so with a result in the results array. And the same sort of thing is true for the second, third, and fourth elements of the array. Now the work of actually doing the operation is in this method here called execute.

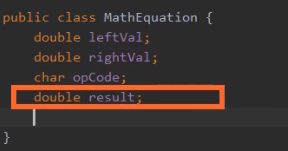


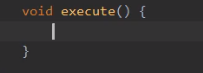
Let me scroll down so we can actually see that method. Now if we look here at the execute method, notice that it receives three parameters: the opCode, the leftVal, and the rightVal. Then it has a switch statement that operates on the opCode, does the appropriate operation, stores that into its local variable named result, and returns that back from our execute method. So what we want to do now is move away from this procedural approach to take advantage of classes that kind of tie all these pieces together. Because again, as this application is currently written, all the data values in the method really have nothing to do with each other unless the application specifically ties them together itself. So the first thing we'll need to do is create a new class.



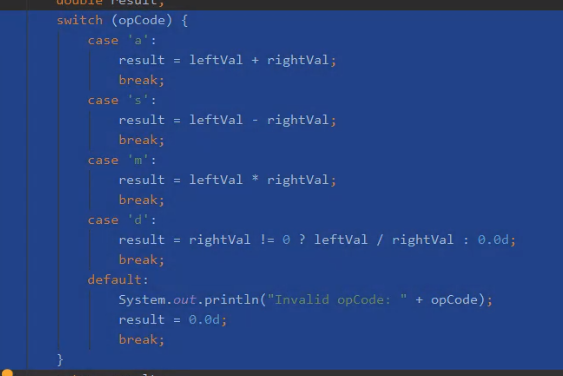


Well the easiest way to do that is to head over here to our project window. Within a project window, we'll go to our package name. We'll then right‑click on the package name. Go up to here where it says new. Then I'll choose Java Class. So now we need to give our class a name. Let's name the class MathEquation. To create the class, I'll go ahead and hit Enter. And once I do that, a couple of things happen. First, if we look here in the project window, notice we have a new source file, MathEquation. Then over here in our source code window, we have our class, MathEquation, stubbed out for us. So now we can start adding members to the class. So the first thing we'll do is add our fields.

  
And again, we'll need fields for each of the parts of the equation. For example, we'll need a field for the leftVal. Then once we have our leftVal, we can add fields for our rightVal, our opCode, and our result. So now our MathEquation class has four fields. We have our leftVal and our rightVal Fields, which are both double, our opCode field, which is a char, and then another field that's a double, which is a result. So those four fields represent the state of our MathEquation class.



So now let's add a method that will execute the equation. We'll call the method execute, and we'll give it a return type of void. Now notice that our execute method within our MathEquation class doesn't take any parameters, and that's distinctly different from the execute method in our Main class. Now that's because here in our MathEquation class, all the values we need are actually part of the class itself. We have all the values in the opCode. So the execute method will use those fields rather than accepting parameters. So now our execute method will need the same switch statement we had over in the execute method for our Main class. So let's head back over to our Main class.



Let's highlight the switch statement. We'll go ahead and copy that. We'll head back to our MathEquation class, and then we'll paste in the switch statement. So now the execute method in our MathEquation class has the switch statement that we need to perform the appropriate operation for each individual opCode. Now let me scroll up here a bit so we can see more of our MathEquation class. So now our MathEquation class has everything we need to perform an equation. We have both our left and right values along with our opCode. We have our execute method that knows how to interpret the opCode and perform the operation. We have a result field that will store the result of the equation. So now we have our class declared. We're just about ready to start using it. So in our next section, let's see how we can use a class once it's declared.

# Using Classes

=>slides: Pg. 11

Once we have our class declared, we're ready to start using it. So to use our Flight class, we'll of course need to start out with a variable of type flight. Now something that's very important to understand, declaring a variable of type flight doesn't actually create a flight object. Instead it creates a variable that can hold what's then a reference to a flight object. To actually create the flight object, we have to use the new keyword. Now when we use the new keyword create our Flight class, we can refer to what is doing in a couple of different ways. We can say it's creating a new flight object, or another way to say the same thing is that we can say that we're creating a new instance of the Flight class. Both those terms mean exactly the same thing. And what we're doing is kicking off a process that has a few steps in it. First of all, we allocate out the memory to hold our Flight class. So this memory will have our passengers field, as well as our seats field within it. We'll also run the constructor code, and our constructor code sets passengers to 0 and seats to 150. And then, we'll return back a reference. That reference is then stored in the variable. So our variable, nycToLv, rather than directly holding an instance of the Flight class, instead holds a reference to an instance of the Flight class. Now as I had the code written here, we're doing all this work across two lines. We could, of course, do it in a single line. But the result is exactly the same. We have our variable, in this case it's slcToSf. We create our new instance. We run the constructor code. And we take the reference of that instance and store it in the variable.

=>slides: Pg. 12

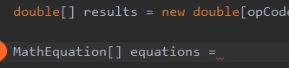
Now because classes are always accessed through references, they're known as reference types. And as reference types, there are some important implications on their behavior. So let's go ahead and create another instance of our Flight class. So again, we'll have our variable here. In this case, it's flight1. We'll create the new instance, run the constructor code, and then store the reference of that instance in our variable flight1. We'll go ahead and create a second instance that we'll assign to a variable, flight2. So again, we have our variable, the newly created instance, and a reference to that instance stored in flight2. Now if we want to interact with the members of one of these instances, we use dot notation. So it's the variable, dot, and the member name. So in this case, we're calling the add1Passenger method using the variable flight2. So what Java will do is it'll look at the reference stored in flight2, follow that to the object instance, and then call the method on that instance. So when we call add1Passenger, passengers is incremented from 0 to 1. So if we now go ahead and print out the value of flight2.passengers, that of course will print out the value 1.

=>slides: Pg. 13

But now what happens if we make an assignment from one of these variables to the other? So we're assigning flight1 to flight2. Well, if this was a primitive type, something like a double or an int, the entire value in flight1 will be copied over to flight2. But these are not primitive types. These are reference types. And working with reference types, the only thing that gets assigned is the reference. So what that means is our variable flight2 will have its reference changed to point to the same object instance that flight1 points to. So if we print out flight2.passengers now, we'll print out the value of passengers in that instance that is currently referencing, which is now 0. But now what happens if we start interacting with flight1? So we say flight1.add1Passenger. We're going to follow the reference stored in flight1. When we call add1Passenger, passengers will increment from 0 to 1. If we call flight1.add1passenger again, passengers in that instance will increment from 1 to 2. But now what happens when I print out flight2.passengers in this case? I haven't made any changes using flight2 at this point, but let's see what the system is going to do. It's going to look at the reference stored in flight2, follow it out to that object instance, and print out that value, and that value is 2. So you see what's happening here is that reference types allow us to have multiple variables that point to the same object instance. And what that means is changes made through one of those variables are still reflected in other variables that reference that same object instance. And that's a really powerful concept when it comes to working with classes. It's important that we understand it so we don't get unexpected side effects. All right, so now in our next section, let's jump back into our Java code, and let's start using our MathEquation class.

# Creating an Array of Classes

Here we are back in STS, and we're looking at our CalcEngine project. This is the project we we're working on when we added our MathEquation class earlier in this module. And what we want to do now is update the application to use our MathEquation class. So to do that, we'll head over to our Main class.   
And here in our Main class is where we have this performCalculations method that we looked at earlier. And as you recall, this method does its work by using a series of parallel arrays. And what we want to do is change the application so we no longer need these individual parallel arrays. Instead we'll have just one array, and that'll be an array of our MathEquation class.



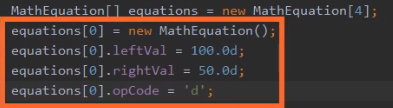
So the first thing we'll need to do is declare an array of type MathEquation, and let's call it equations. Now you'll notice that when we declare the array, it's very much like declaring an array of a primitive type. We, of course, give the variable a name. And when we specify its type, we identify the type itself, which is MathEquation. But then we indicate that it's an array of putting the square brackets after the type name.



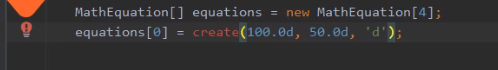
So then we'll go and create a new MathEquation array of size 4. Now notice here when we create this array, we're creating an array of MathEquation references. So we're not creating four instances of the MathEquation class. Instead, we're creating four references of type MathEquation. Each of the elements within this array will need to explicitly create an instance of the MathEquation class.



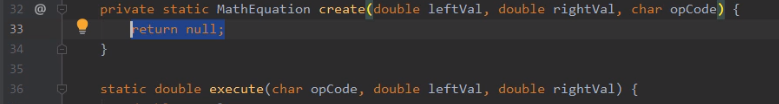
So we'll start out by setting the zeroth element to a new instance of the MathEquation class. So now that zeroth element of our equations array references a new instance of our MathEquation class. So that means we can now set that instance's field values. Now if we look up here at our parallel arrays, notice that the first calculation we were doing was a left value of 100, a right value of 50, and an opCode of d.

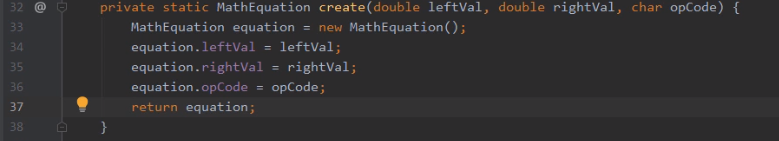


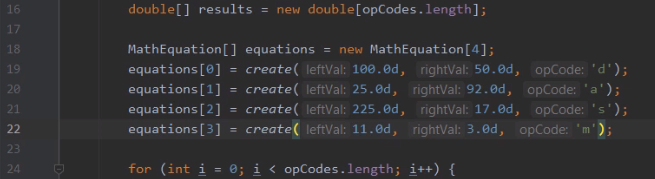
So let's set each of those fields on equation subzero to the appropriate values. So now with that code in place, each of the fields within our new MathEquation instance have the appropriate values. Now, of course, that just takes care of one member of a four‑element array. We're going to have to repeat all this work for each of the other elements. So rather doing all that code explicitly right here, maybe what we should do instead is use a method to create the MathEquation instance and set each of the fields. So let's go ahead and remove this code here where I set the individual fields, as well as create the new instance of the MathEquation class.



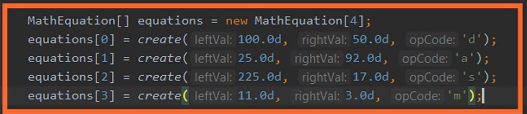
So now with that code removed, I'll call a method we'll create named create. Then I'll pass on the values we want for each of the fields. So now that we have a call to the method passing in the appropriate values, we now need to create the method. But it turns out I don't have to create the method manually. You'll notice that when I place my cursor here on the method name that STS places this light bulb off to the left. That tells me the STS has some ideas for how it can help. So to access those options here on Windows, I'll press Alt+Enter. And you'll notice when I do that, one of the options is to create the method. So I'll go ahead and choose that option.

 You'll notice that it stubbed out this create method. It indicates it has a return type of MathEquation, which is what I want. It also provides a list of parameters and has given those parameters some default names. But we'll want to change those parameter names. So to get over to those parameters, I'll press the Enter key a few times. For the first parameter, I'll change its name to leftVal. I'll press Enter to accept it and move over to my next parameter. I'll make this one rightVal. Using the Enter key, I'll go over to our last parameter, and I'll make this one opCode. Then I'll go ahead and accept that parameter. So now that we've got our method stubbed out, we can start adding the code that we actually want.

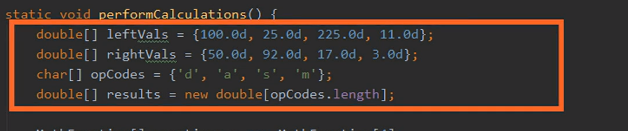
So what we'll do is create a new instance of our MathEquation class and assign it to a local variable named equation of type MathEquation. So that gives me a new instance of the MathEquation class. So now I can use each of the parameters to set the appropriate fields on that new instance using our equation variable. So now once we've set the fields, I can simply return equation, which will return back the reference to our newly created instance of MathEquation. So now with that, our create method takes care of the details of creating a new instance of our MathEquation class and then returning back a reference to that newly created MathEquation instance. So that means that we can now use its create method to initialize each of the elements in our equations array. So let me scroll back up to where we set our equations array. So now we're back up here in our performCalculations method.

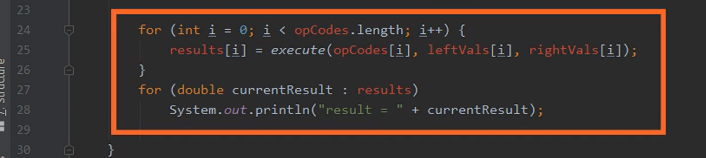
 Ao we'll go ahead and use that create method to set elements 1, 2, and 3 of our equations array. So now with that, we have our equations array set up. Each of the elements in the array are referencing new instance of the MathEquation class. And each of the fields within those instances will have the values that we want to use to perform our calculations. So now we're ready to start using this array. We'll see how to do that in our next section.

# Using the MathEquation Class

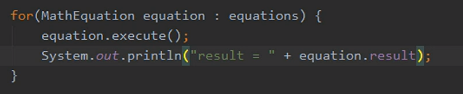


Here we are back in STS, and what want to do now is continue the work of migrating our application to using our MathEquation class rather than having to manage a bunch of individual arrays directly. So now we already have the code in place that sets up our MathEquation array. So that means that we can take away all the code related that are working with those individual arrays that the application had to explicitly manage on its own.

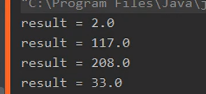
So that means first of all, we can get rid of the arrays themselves, so I'll remove those. So now with those arrays gone,

we can also get rid of these loops here at the bottom. And notice that one of these loops here at the bottom actually calls the execute method that accepts a number of parameters, and these parameters were all values that came from those arrays. So that means first of all, we can go ahead and get rid of these loops. So now once those loops are gone, we can also get rid of the execute method that's located here in our Main class that relied on receiving each of those values as parameters. So let me go ahead and scroll down a bit.

 So now we're down here in our Main class's execute method. We don't need this anymore, so let me go ahead and get rid of this. And now with that method gone, let's scroll back up to where we're doing the work with our MathEquation class. So now we're back up here in our performCalculations method. So what we want to do now is loop through this equations array and do the work contained in each of the elements.



So let's start out with a for loop, and what we'll do is use the variation of the for loop and do a foreach loop. Remember that the foreach loop can iterate to the elements in the array automatically. So to do that, we'll declare a variable named equation of type MathEquation. Then we'll place a colon in the array we want to iterate through, which is our equations array. So what this for loop will do is walk through each element in the equations array and each pass through the loop the equation variable will represent the current element of the array. So first it'll represent the zeroth element, than the 1 element, and then all the way through to the end of the array. So we can simply use that equation variable to perform the work we want to do on each of those elements. Well the first thing we'll want to do is execute the work that's contained in the MathEquation instance. So we'll call equation.execute. Now remember, the way we wrote the execute method is that we'll look at the leftVal, the rightVal, and the opCode fields, do the appropriate work, and then store that result in the result field. So that means we can simply display equation.result. And with that, our application is all set. Rather than the application having to manage a bunch of values individually, manage how all the work is done and the details of how that work is done, instead the application simply relies on our MathEquation class. So it sets up an array of type MathEquation, it sets all the appropriate values for each of those equations, and then it can simply loop through and say to the MathEquation class, hey, go do an execute. You handle the details of how that's done, and just tell me what the result is. So what we've done now is we've got a good division of labor. The MathEquation class focuses on how to manage an equation. The application simply takes advantage of the capabilities provided by our MathEquation class. In addition, if we need to do any other math equations anywhere else in the application, we can now leverage that existing MathEquation class. So now just to confirm that everything works, I'll go ahead and run the application.

  
Once the application runs, we can see the results in our run window. So if we look up here in our code, the first equation was a d opCode, which means division. So we divide 100 by 50. That gives us the result of 2. Our next equation is an add operation, so we'll add 25 to 92. That gives us 117. Then we have a subtract operation, so we subtract 17 from 225. That gives us 208. And then our last operation is a multiply. So wel multiply 11 times 3, and that gives us 33. All right, so now in our next section, let's take a look at this idea of encapsulation and how we use something known as access modifiers to achieve it.

# Encapsulation and Access Modifiers

=>slides: Pg. 14

Now as we've mentioned. as we declare our classes, in general, we went the details of how that class is implemented to be hidden. This idea of hiding certain details about the class is what's known as encapsulation. So you're encapsulating the details within the class. Now in order to achieve encapsulation, Java gives us access modifiers. What access modifiers allow us to do is control the visibility of classes and their members.

=>slides: Pg. 15

So let's take a look now at three of the basic access modifiers. Now one option for access modifiers is to not specify any access modifier at all. And in this case, the item would be visible only in the package where the class is declared. And for that reason, this is sometimes known as package private. Now classes themselves can be package private, as well as the members of a class can also be package private. Then we have the public access modifier. If something is public, that means it's visible everywhere. And classes can be public and the members of classes can also be public. And then at the opposite end of the spectrum, we have private. If something is private, it's only visible within the class where it's declared. Now, in general, classes cannot be private. Now there is an exception to that. There's something in those nested classes that we'll talk about later on in this course, and nested classes can be private. But in general, classes cannot be marked as private. But now the members of a class can be marked as private.

=>slides: Pg. 16

So let's again look at our Flight class and see how access modifiers affect the uses of that class. So we have our class itself, as well as our two fields, passengers and seats, we have a constructor, and we have our add1Passenger method. Now there are a number of things we need to consider about the usability of this class. One of the key things we need to consider is where can variables of this type be declared? So if I mark the class itself as public, that means I can declare variables of type flight anywhere I want to. But interestingly, marking the class as public does not mean I can necessarily create instances of the class. If I want to be sure I can create class instances from anywhere, I need to also mark the constructor as public. So by marking the constructor as public, I can declare variables of type flight and create new instances of that type flight from anywhere. So now when it comes to our fields, let's go ahead and mark those as private. That means that there are only accessible from within the class itself. So if I have code that tries to use that flight reference and print out the number of passengers, I'm actually going to get a compile error because passengers is private, meaning it's not accessible from outside the class. But now for our add1Passenger method, let's go ahead and mark that as public, and because it's public, it can be accessed from anywhere. So what that means is the add1Passenger method gives us a controlled way for user's to increment the number of passengers without giving them access to our passengers field itself.

=>slides: Pg. 17

So again looking at our Flight class, let's take a closer look at our add1Passenger method. Now currently the way this method is implemented, we check to make sure there are still seats available. As long as there are seats available, we increment the number of passengers by 1. But notice if there are not seats available, we silently fail. We don't increment the number of passengers, and we give no indication that there's no room left. So we should really improve this method. Now this idea of trying to add passengers and there being no room left is something that might come up fairly often. So let's go ahead and add a method that will handle when we try to add too many passengers. And for simplicity, we'll just have it print out the fact that there are too many passengers. Now notice that our handleTooMany method is private. And that means that we don't want anyone from outside the class to access this method. But of course, methods within this class can access it. So in our add1Passenger method, if there aren't any seats left, then in that case we'll go ahead and call our handleTooMany method.

=>slides: Pg. 18

So now let's look at how this affects the usability of our class. So we have our Flight class, which is public. It has our two private fields. It has a public instructor and our public add1Passenger method. And now it also has a private handleTooMany method. So we can, of course, still use the class itself. But if we try to call handleTooMany, we're actually going to get an error because handleTooMany is private. It's not accessible from outside the class. But now what about add1Passenger? Well add1Passenger is public, so we can still call add1Passenger. The fact that add1Passenger calls a private method does not impact the accessibility of add1Passenger itself. And this is a key part of how we implement our classes. We have a finite number of members that we make public and that controls the interaction points with our class. And then we deal with the details of our work using private members, which hides those members away from being used outside the class itself. All right, so now in our next section, let's take a look at some of the special references provided by Java.

# Special References: this and null

=>slides: Pg. 19

So let's take a look now at some of the special references that Java provides. Now one of the special references is what we call the this reference. The this reference is a reference to the current object instance. One of the things we often use this for is reducing ambiguity, making it clear that we're referring to a member of the current object instance. Also, this is useful for allowing an object to pass a reference to itself as a parameter to a method. Then we have the special reference null. Null represents an uncreated object. So it allows us to have a reference that, rather than referring to an object instance, doesn't refer to any instance at all. And null can be assigned to any reference variable.

=>slides: Pg. 20

So let's first take a closer look at this. So we again have our Flight class. Remember, our Flight class has the two fields, passengers and seats. Now let's say we want to add a method here, hasRoom. What hasRoom does is return back true or false, indicating whether there's room on this flight to combine the passengers from another flight. So the first thing hasRoom does is figure out the total number of passengers between the two flights, then returns back a true or false value, indicating whether there are enough seats to hold all those passengers. Now let's look here where we calculate the total. Our passengers field actually shows up twice. We first have f2.passengers. So this refers to the number of passengers on the flight that was passed in as a parameter. But our other use of the passenger field is unqualified. And in this scenario, the passengers field not being qualified actually refers to the current object instance. In other words, this code is doing exactly we want it to do. But there may be scenarios where we want to make that fact more clear. So what we can do is, rather than just saying passengers here, we can say this.passengers, making it clear that we're referring to the current object instance. Now in this scenario, the use of this is not required, although it's helpful for reducing ambiguity. But as we'll see a little bit later in this module, there are scenarios where the this reference is required in order for the code to do what we want it to do.

=>slides: Pg. 21

So now let's take a closer look at null. So let's say we have some code that creates two instances of the Flight class and assigns them to the variables lax1 and lax2. Then we'll have some logic that goes off and adds some number of passengers to each of those flights. And then we'll have another variable here, lax3. But we don't want lax3 to be initially set to an instance of the Flight class. So what we can do is actually initialize it to the value null. So that means that lax3 doesn't actually refer to an instance of the Flight class. It doesn't actually refer to any instance of any class at all. Lax3 is simply null. So let's add some code here that determines whether there's room to combine the passengers that are on lax1 and lax2. So it uses our hasRoom method. And if there is, room we'll call some method called createNewWithBoth. And what we'll say this method does is it creates a brand new instance of the Flight class and combines the passengers from lax1 and lax2 and returns back a reference so that new instance of the Flight class. So if there was room lax3 now references that new instance of the Flight class. So then we'll have our code go off and do some other work. When they work completes, we need to determine whether lax3 was set to an instance the Flight class or not. Well what we can do is check to see if lax3 is not equal to null. And if lax3 is not equal to null, that tells us that lax3 now references an the instance of the Flight class, which means those flights must have been combined. So as you can see, null is useful for initializing variables to refer to no object instance at all, and we can use it to test whether a particular variable refers to an object instance or not. All right, so now in our next section, let's take a look at field encapsulation.

# Field Accessors and Mutators

=>slides: Pg. 22

As we've talked about, as we design our classes, we want to keep the details of how that class is implemented hidden from the users of the class, and our fields are generally considered to be an implementation detail because the fields are details of how we actually manage the state of our classes. So in most cases, we do not want our fields to be directly accessible from outside the class. So instead of allowing users of our class to directly access to fields, we instead want to control access to the fields through methods. This will allow us to evolve the implementation of our class and possibly even change the way we store the state within our class without breaking any code that uses our class.

=>slides: Pg. 23

Now a common pattern we use for protecting our fields is what's called the accessor/mutator pattern. And in this pattern, we use a pair of methods to control access to the fields. So we have one method, which is known as the accessor. The accessor method retrieves the value of the field. This is also commonly known as a getter method. When we create the method, we'll normally name it get followed by the field name. And then we also have a mutator method. The mutator method modifies the field value. This is also commonly known as a setter. When we create these methods, we'll normally name it set, followed by the field name.

=>slides: Pg. 24

So again, looking at our Flight class, let's focus in on our seats field. So now we've marked this field as private because we don't want users of the class directly accessing the field. So we'll use the methods to control interaction with it. So we'll first have our getter or what's called our accessor. So since our field is named seats, we've named the method getSeats. Notice the method's return type is the same type as the field itself, and all we're going to do in our getter here is return back the value of seats. Now we'll also need our mutator or what we call our setter. So we've named this method setSeats. Its return type is void, meaning it doesn't return any value. But notice is accepts a value as a parameter. In this case, we've named that parameter seats. So what we want to do is have the value that's passed in to setSeats assigned to our seats field. But now it turns out, as this code is currently written, there's a bit of a problem. Now looking here at seats on the right side of the equal sign, this refers to our seats parameter, and that's what we want. We want to take the value that's passed into our setSeats method and then assign it to our seats field. The problem is on the left‑hand side of the equals sign, as this code is written, that seats also refers to the parameter, and that's not what we want. We want to assign the value to the seats field itself. And the issue here is that when a parameter name within a method has the same name as a field within a class, the compiler assumes that we want to use the parameter. So we need to be more explicit about the fact we want to use the field, and we know how to take care of that. Remember we have our this reference. Our this reference allows us to refer to something within the current class instance. So by using this, we can now take the value that's passed in as a parameter and assign that value to our seats field.

=>slides: Pg. 25

So to use our methods, it's really straightforward. We'll go ahead and create an instance of our class. We want to set its value. We'll simply call setSeats, passing in the value. So this will set this flight to have 200 seats. If we want to get the value back, we call our get method, which will return back the value, which will display 200.

# Source File Declaration Rules in Java

=>slides: Pg. 26

Lets see rules associated with declaring classes, import statements, and package statements in a source file. We can call them as declaration rules in Java.

# Summary

=>slides: Pg. 27

To wrap up, here are some of the key things you want to remember from this module. Remember we started out by looking at classes. We said that a class is a template for creating objects. Then an object is simply an instance of a class. So it's a class that describes the type that we're creating. The object is what we get when we actually create an instance of that type. Now remember that classes are reference types. That means that when you declare a variable whose type is a class, it doesn't actually create an instance of the class. The variable can simply hold a reference to the class. To actually create an instance of the class, we used the new keyword. So when we call new, we create that class instance, and then we assign that reference into a variable. Now remember that multiple variables can reference the same instance, and this ability for multiple variables to reference the same instance of an object is really powerful, but it's also really important that we understand that's the case so we can work effectively with our classes.

=>slides: Pg. 28

Now as we saw, classes have three types of members. Classes have fields, and fields store the object state. Then we saw that classes have methods, and methods are executable code, and remember that class methods have access to the class's state, so they can actually modify and manipulate the state within the class. We can also use methods to simply perform operations that are related to that class itself. And then remember that classes can also have constructors, and constructors are executable code that is executable code that runs automatically during the creation of a class instance. And generally what we use constructors for is to set that class's initial state.

=>slides: Pg. 29

Then we saw that classes have some special references. We have the this reference, which allows us to refer to the current object instance. And then we have the null reference, and the null reference represented an uncreated object. As we saw with null, we can use it to initialize a variable to indicate that it doesn't reference any object instance, and we can also use null to test to see whether a variable refers to an object instance.

=>slides: Pg. 30

And then we looked at the access modifiers. What access modifiers do is control visibility. They can control class visibility. They can also control class member visibility. And access modifiers enable us to have encapsulation. So with access modifiers, we can have portions of our class that are accessible from outside the class and other parts of our class that are not accessible from outside the class. And as we saw, we normally don't want our fields to be directly accessible. So instead, we use methods to provide access to our field values. So we have the accessor methods, what we often call the getter methods. We use these to retrieve the field values. Then we also have mutator methods, what we sometimes call setter methods, that we use to modify the field values. All right, that wraps up this module. In our next module, we're going to take a closer look at constructors. We're also going to look at something known as an initializer.

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