7. More About Inheritance

Contents

[1. Introduction 1](#_Toc12630)

[2. Special Reference: super 1](#_Toc24834)

[3. Preventing Inheritance and Method Overriding 1](#_Toc4054)

[4. Requiring Inheritance and Method Overriding 2](#_Toc26753)

[5. Inheritance and Constructors 2](#_Toc4855)

[6. Implementing Abstract Classes and Constructor Inheritance 2](#_Toc2656)

[7. Using Constructor Inheritance in CalcEngine 2](#_Toc17206)

[8. Summary 2](#_Toc28976)

# 1. Introduction

=>slides: Pg. 1

Welcome to our next module, More About Inheritance. As you may have guessed by the title, in this module, we're going to continue our discussion of inheritance. We're going to look at some of the finer points of dealing with inheritance.

=>slides: Pg. 2

So the first thing we'll take a look at is the Java super reference. The super reference is sort of like the this reference, but has some specific capabilities as related to inheritance. And then from there, we're going to look at a few issues related to controlling how a class is used in inheritance. For example, we have the option to prevent class inheritance. We can declare a class that no other class is allowed to inherit from. That class can only be directly instantiated. We can also declare classes that have methods that are not allowed to be overridden. So although the class may be allowed to be inherited from, we have specific methods that cannot be overridden. Now we can also require class inheritance. So we can declare a class that's not directly creatable. Instead, the class only exists to serve as a base class. And then we can even have classes who have methods that must be overridden. So when we declare a class, we don't provide an implementation of the method, but we require classes that inherit from our class to provide their own implementation. And then we'll finish up. We'll look at constructors and inheritance. And as we'll see, constructors behave a bit differently than methods do when inheriting one class from another.

# Special Reference: super

=>slides: Pg. 3

So to get started, let's look at the Java super reference. Now the super reference is very similar to the Java this reference, and as you recall from our previous course we mentioned that this reference allowed us to refer to the current object and it turns out the super reference does that, too. Suppose this and super allows to refer to the current object, but the super reference has a key difference from this. When we use the super reference to refer to the current object, it treats the object as if it's an instance of the base class and the reason that that's important because there are scenarios where we may have overridden something in the derived class and we want to access a member from the base class. So using the super reference, we can reach back and get method implementations from the class that we inherited from, even if we've overridden that method.

=>slides: Pg. 4

So to help us understand that, let's start out with a quick look again at our Flight class. As you recall, our Flight class has a couple of fields in it that identify a particular flight, that's flight number and Flight class. And in our previous module, we overrode the equals method because as you recall that the full implementation of equals that we inherited from the Object class didn't look at the contents of a particular object, it simply compared to references to see if they were equal, and in our case, that wasn't what we wanted. We actually wanted to look at these flight number and Flight class fields to determine if two flights are equal. So we overrode this equals method, the first thing we did was check to see if o refers to an instance of the Flight class, if it doesn't, we're just going to return false, but if it does, we'll then cast that o reference from an object reference into a flight reference to store that into a local variable named flight, and then once we add our flight reference, we could go through and compare the flight number field in the current instance to the flight number field of the flight we received and do the same thing for the Flight class. And in the overwhelming majority of scenarios, this is the best way for us to implement the equals method.

=>slides: Pg. 5

But now, let's take a look at a special situation here. What if I go ahead and create an instance of the Flight class specifying a flight number of 175 and I assign that reference to a variable of type flight named F1. And then we assign F1 to another flight reference named F2. So now F1 and F2 point to the exact same object. So then our program would go off and do some other work, and then later we call F1 = F2. So now we happen to know that F2 and F1 point to the same object instance,

=>slides: Pg. 6

but when our code runs and we get over to our Flight class and we're inside the equals method, what's going to happen? Well, we're going to take that reference F2, which we've received into our o parameter. The first thing we're going to do is make sure that o refers to one instance of the Flight class, now, of course, in this scenario, we know it does. So then we do the work, the cast o to a flight reference and then we do the work to compare those fields and we've done all this work even though the reference we received points to the exact same instance of the class that were currently in. Now, in this case, we only did a few steps, so doing these steps is not that big a deal. But in some scenarios, you may do a great deal of work inside this implementation. And to do all that work unnecessarily would be unfortunate because remember that our base class object, if we recall it's equals method, it would do that very inexpensive check of simply looking at the references themselves and looking to see if they point to the same object instance. So by making a call to the base classes equals method, again, our object classes equals method, we could have very easily determined that these two flights were equal because both references refer to the same object instance, so we could have simply returned true without doing all that other work. But the problem here, if you look at the way we've called equals, we're not calling equals as it's implemented in our base class, we'd be calling equals as it's implemented in our class.

=>slides: Pg. 7

So what that would mean, when we call equals here, we would simply call right back into ourselves. So if we call back into ourselves, the first thing we would do is again make that call to ourselves, calling back into ourselves, again, we'd make that call into ourselves, and again, coming back into ourselves, and we would actually keep doing this until our program finally crashed because our equals method is repeatedly and unendingly calling back into itself and that's not what we wanted to do. When we called equals here, what we wanted to do was call equals as it's implemented in our base class

=>slides: Pg. 8

and that's where the super reference comes in. If instead of simply saying equals, if we say super.equals, that's just that, yeah, I want the equals method, but not as I've implemented it, but as my base class has implemented it, so that allowed us to override our equals method providing the implementation that's right for our particular class while leveraging the work that was provided for us by our base class. Alright, so now in our next section, let's see how we can prevent inheritance and method overriding.

# Preventing Inheritance and Method Overriding

=>slides: Pg. 9

When we declare a class, by default, we have no control over how that class is used in inheritance, because by default, any class we declare can be extended, so other classes are free to derive from our class. In addition, if someone derives from our class, the deriving class is free to override any of our methods, and although these are generally good things, it's not always the behavior we want. So for that reason, we have the final keyword. The final keyword allows us to override this default behavior. So by using the final keyword, we can actually have a class that we don't allow other classes to extend or we may want to allow someone to extend the class, but there may be specific methods that are may be critical to our class that we don't want to allow anyone to override, and we can prevent method overriding by again using the final keyword.

=>slides: Pg. 10

So let's first look at preventing inheritance. Now remember, we have this Passenger class and the Passenger class was our very simple class, but it was very important to our Flight class because, remember, the Passenger class represents a passenger that's added to our Flight class. Now, by default, anyone can come along and extend this Passenger class so they could create their own custom versions of passengers. But one of things you want to keep in mind if you're going to allow someone to extend your class, you want to make sure that any of your code that depends on that class will behave correctly if someone extends it. So because we consider the Passenger class so critical to the behavior of our Flight class, we might make the decision to not allow other people to extend our Passenger class =>slides: Pg. 11

and we could enforce that by simply marking the class as final. So once the class is marked as final, no other classes can inherit from it. Now one thing to note, here we have final public. Those two keywords, final and public, can appear in either order.

=>slides: Pg. 12

So we could say public final, as we have here, or final public, as we had previously. Now, in some scenarios, you may be willing to allow others to extend your class, but there might be certain methods that are critical to the behavior of your class, so you don't want anyone who inherits from your class to override those methods.

=>slides: Pg. 13

So let's take a look here at our CargoFlight class, and with our CargoFlight class had one public method, which was add one package. Then we had a couple of private methods, hasCargoSpace and handleNoSpace. Now let's say for our CargoFlight class, we consider our add1Package method to be critical to the behavior of CargoFlight. So we don't want to allow anyone to override this method. Well, then all we need to do it simply mark that method as final, and then once we mark it as final, anyone could inherit from our CargoFlight class, but no one else can provide their own implementation of add1Package. Alright, so now in our next section, let's take a look at the other side of this issue, requiring inheritance and requiring method overriding.

# Requiring Inheritance and Method Overriding

=>slides: Pg. 14

Now if we look at the default way we use a class, well, once we declare the class, we're generally going to expect to directly create instances of that class. And then if someone does decide to extend our class, well, it's up to them which of the methods they've inherited they wish to override. And in general, these defaults are what we're looking for. We declare a class so we can create instances of it, and so when it inherits from our class, it's up to them what they want to override. But there are some scenarios where we don't want that default behavior, and that's where the abstract keyword comes in. With the abstract keyword, we can indicate that a particular class can't be directly created. It can only be extended. And, again, with the abstract keyword, we could indicate that there are certain methods that when you inherit from this class, you have to override those methods.

=>slides: Pg. 15

Now these scenarios may seem a bit strange, but there are some cases where it's really, really useful. Let's look at an example. So let's declare a new class here named Pilot, and the pilot is someone who flies a plane. So we want to be able to associate a flight with this pilot, so we have this currentFlight field. And our Pilot class will expose a method named fly that accepts a reference to a flight, and that's how we indicate that we want a pilot to fly a particular flight. But in order for the pilot to fly the flight, they've got to determine if they're able to accept that flight. So we'll call a method named canAccept. As long as canAccept returns true, then we'll go ahead and associate that flight with the currentFlight field. If canAccept returns false, then we'll indicate that we can't accept that flight. Now implementation of handleCantAccept will be really simple. We'll just print out a message. But now when it comes to our canAccept method, what flights a pilot is allowed to accept is going to depend on what kind of pilot it is. So here in our Pilot class, we're not going to directly implement the canAccept method. We're going to leave that up to classes that inherit from Pilot. So what we'll do is as we're declaring our canAccept method, we'll start out by marking it as abstract. And then once we indicate that it's abstract, we'll provide its returned value, as well as the rest of its signature. Now because canAccept is marked as abstract, that means that here in the Pilot class, we're not going to provide an implementation. So rather than putting opening and closing brackets on the method, we're just going to put a semicolon. Now this might seem a bit odd. We're declaring a method here, but we're not providing a body for the method. So what's the reason for declaring the method at all? Well, because we've declared this method here in the Pilot class, the Pilot class is allowed to make calls to that method. So our fly method here is able to call the canAccept method, even though the Pilot class doesn't know the details of how the canAccept method is going to be implemented. But we can still have logic here that leverages that method. Now one thing to note, because canAccept is marked as abstract, that means the method must be overridden. So if we go up here to our Pilot class declaration, because it contains an abstract method, the Pilot class itself must also be marked as abstract, meaning that we cannot directly create an instance of the Pilot class. We can have references whose type are Pilot, and other classes can extend the Pilot class, but you can never directly create an instance of the Pilot class. So any time a class has an abstract method, the class itself must also be marked as abstract. Now, technically, you can have a class marked as abstract that has no abstract methods, but that would be extremely uncommon. In general, a class is marked as abstract because it contains one or more abstract methods. So now that we have our Pilot class, we can declare other classes that extend the Pilot class.

=>slides: Pg. 16

So let's start out here with a CargoOnlyPilot. CargoOnlyPilot extends the Pilot class. Because the Pilot class contains an abstract method, CargoOnlyPilot is required to override that method. We'll be sure to mark that method with the @Override annotation. Now the implementation of canAccept is up to this class. And what we're going to use the CargoOnlyPilot class for is pilots that are only allowed to carry cargo. They are not allowed to carry any passengers. So in this class, the implementation of canAccept is going to check the number of passengers on that flight. And as long as it equals to 0, we'll go ahead and return true. So now let's go and declare another class, the FullLicensePilot class. Now this also extends the Pilot class, and FullLicensePilot is kind of our super pilot. He's allowed to fly any flight he wants to. But now because he extends the Pilot class, he's required to override the canAccept method. Again, we'll be sure to mark it with our @Override annotation. And for this class, the implementation of canAccept is always going to return true. Because we would use the FullLicensePilot class, the representing pilot who has a license that allows him to fly any flight he'd like to. So what we have here now are two classes, CargoOnlyPilot and FullLicensePilot, that fully leverage the capabilities of our Pilot class, but each one has provided its own specialization of the canAccept method. So now in our next section, let's take a look at inheritance and constructors.

# Inheritance and Constructors

=>slides: Pg. 17

When one class inherits from another, one of the really neat things that happens is a derived class automatically exposes all the public methods of its base class. But in case of constructors, things work differently. Constructors are not inherited. In other words, the only constructors that are available in our class are ones that are directly tied to that class itself. And this has important implications about how we handle constructors when we derive one class from another.

=>slides: Pg. 18

So when we create an instance of a class that derives from another class, remember that that class is itself an instance of the base class. In other words, the derived class has the characteristics of its base class. So as part of constructing the derived class, a base class constructor is always called. So any construction work a base class expects has an opportunity to happen. Now by default, when we construct an instance of the derived class, it will call the no‑argument version of its base class constructor. So it does kind of the default construction on the base class. But we have the option to explicitly call one of the base class constructors, and we do that by using the super keyword. Remember, we could call our own class's constructor by using the this keyword. We can call our base class constructor by using the super keyword. And we're able to pass any necessary parameters to that base class instructor. But now one thing to keep in mind, if we're going to call the base class constructor explicitly, it has to be the first line of our constructor.

=>slides: Pg. 19

Now let's again take a quick look at our Flight class. Remember that one of the fields in our Flight class was flightNumber. When our Flight class exposed a number of constructors, it had its default constructor, which accepts no parameters, and it also had a version of its constructor that accepted the flightNumber that it used to set its flightNumber field.

=>slides: Pg. 20

Remember, the Flight class serves as the base class for our CargoFlight class. So our CargoFlight classic extends the Flight class. But, remember, the CargoFlight class has no explicit constructors of its own. So how does that affect things when we try to create instances of these classes? Now, of course, in the case of our Flight class, it explicitly exposes a constructor that accepts the flightNumber. So here we can create an instance of Flight, setting its flightNumber to 175. Now, remember, in the case of our CargoFlight, because we have no explicit constructors, Java will automatically provide the no parameter constructor for us. So we can create an instance of CargoFlight that specifies no parameters because we're using that default constructor. But now what happens if we try to create an instance of CargoFlight specifying its flightNumber? Its base class Flight has a constructor that accepts a flightNumber, but that does not mean that CargoFlight can use that same kind of constructor. If CargoFlight does not explicitly expose a constructor that accepts a flightNumber, we'll get an error when we try to use a constructor that accepts a flightNumber.

=>slides: Pg. 21

So let's see how we can improve our CargoFlight class. Well, if we want to create a CargoFlight instance that accepts a flightNumber, we'll need to explicitly declare a constructor that accepts that flightNumber. Now as part of constructing this instance, we don't want to duplicate the work that's already in the Flight class, so we'll simply use a super keyword to execute the Flight class's constructor that accepts the flightNumber. So this way we're creating a very simple constructor that says that our CargoFlight class accepts a flightNumber, but we're not duplicating the work that's in the Flight class. We're just explicitly calling the constructor provided by the Flight class. Now our CargoFlight class has some fields of its own. One of those fields is the maxCargoSpace, and it has the default value of 1000. But in some cases, we might want to create a CargoFlight that has a different amount of cargo space. So we might expose a constructor here that accepts both a flightNumber and the amount of cargo space. So in this constructor, we'll go ahead and set the maxCargoSpace field, but we still need to do the work of dealing with the flightNumber. Now remember that any calls we make to another constructor have to be the first line in this constructor. So we could again use the super keyword to call the Flight class's constructor that accepts a flightNumber. And that would certainly be fine. But in our case, our CargoFlight class already has a constructor that accepts the flightNumber.

=>slides: Pg. 22

So it would be better that rather calling the Flight class's constructor that accepts a flightNumber, we should use the this keyword to call the CargoFlight class's constructor that accepts the flightNumber. That way the work of constructing an instance of CargoFlight that accepts a flightNumber is centralized in one constructor.

=>slides: Pg. 23

Now our CargoFlight class should probably have some additional constructors as well. So let's go ahead and expose a constructor that accepts no parameters. Now this constructor has an empty body. But that doesn't mean this constructor does nothing. Because the CargoFlight class inherits from the Flight class, if we do not explicitly call one of the Flight class's constructors, then the CargoFlight class's constructor will automatically call the Flight class's constructor that accepts no parameters. Now let's go ahead and add one more constructor. Let's add a constructor here that accepts just the maxCargoSpace. And inside this constructor, we'll go ahead and set that maxCargoSpace field. So, again, we make no explicit call to the base class's constructors. So, again, we're going to automatically call the Flight class's constructor that accepts no parameters.

=>slides: Pg. 24

So now with all this in mind, our CargoFlight class has four constructors, one that accepts the flightNumber, one that accepts flightNumber and maxCargoSpace, one that accepts no parameters, and one that accepts just maxCargoSpace. So let's go ahead and create some instances of the CargoFlight class. So now we'll go ahead and create a CargoFlight instance that sets the flightNumber to 294. When we do that, we're calling into our CargoFlight class's constructor that accepts the flightNumber. And remember that this constructor makes an explicit call to the Flight class's constructor that accepts the flightNumber. Now let's go ahead and create another instance of CargoFlight. This one specifies both the flightNumber and a maxCargoSpace, so we'll call the constructor that accepts those two values. Remember that this constructor sets our maxCargoSpace field. But before it does that, it calls the CargoFlight class's constructor that accepts flightNumber. And then that in turn calls a Flight class's constructor that accepts flightNumber.

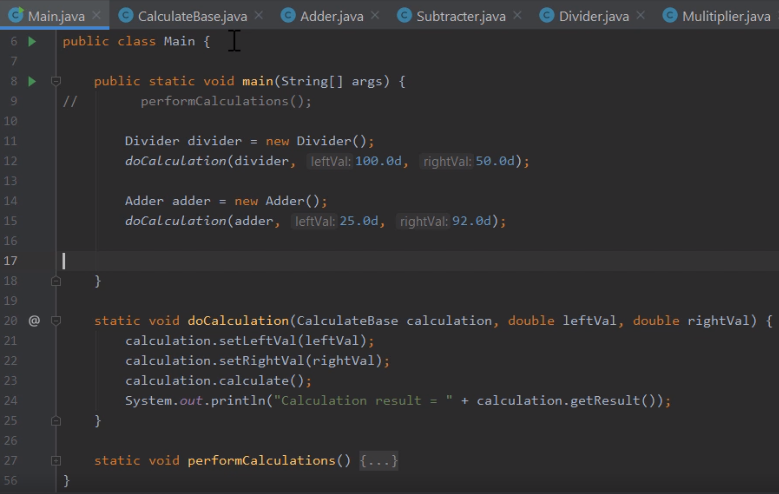
=>slides: Pg. 25

Now let's go ahead and create an instance of CargoFlight that specifies no parameters to the constructor. So that calls the CargoFlight class's default constructor, which will automatically chain to the Flight class's default constructor.

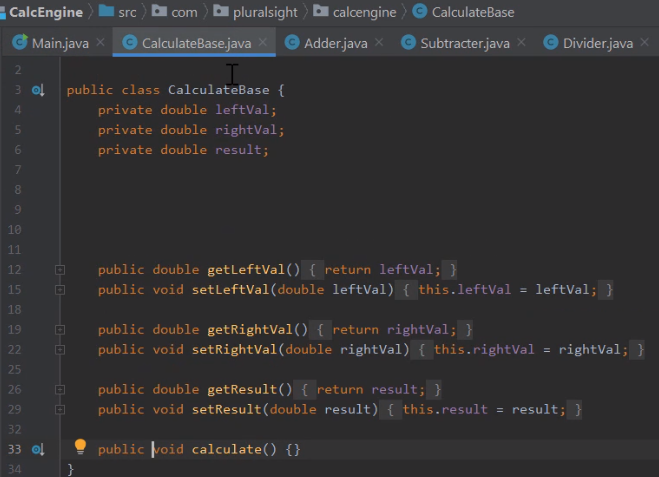
=>slides: Pg. 26

And let's create one more instance of CargoFlight. This one specifies only the cargo space. So that will call into our constructor that accepts maxCargoSpace. Now this constructor sets our maxCargoSpace field. But before that happens, the implicit call to the Flight class's default constructor will still occur. So as you can see, any time we create an instance of a class that derives from another class, that base class constructor always gets called. We can do it explicitly, or it will happen implicitly.

# Implementing Abstract Classes and Constructor Inheritance



Here we are in STS, and what we want to do now is take a look at how we can use some of the techniques we've learned in this module to improve the use of inheritance in our CalcEngine project. As you recall in a previous module, we added some classes that were specific to the individual math operations. For example, we have our class here, Divider. We also have a class, Adder. In addition to those, we created classes to handle subtraction and multiplication. As you recall, as we added the code to our Main class to leverage these math classes, we added this method here, doCalculation. And in this doCalculation method, what we did was receive a reference to the individual class, as well as the left and right values. Once we had those, we could go through and set the left and right values, perform the calculation, and display the result. And the reason we were able to do that is because each of the classes that derive from CalculateBase override the CalculateBase class' calculate method. But now an interesting question here, what would happen if someone created an instance of CalculateBase rather than one of the derived classes and called doCalculation using that?



Well, let's take a look at our CalculateBase class, and let's see what we have there. So now we're here looking at our CalculateBase class, and you can see here where we have the calculate method. Now notice the calculate method has an empty body, and the reason it has an empty body is because we expect each class it inherits from CalculateBase to actually override that method. We never really expect anyone to directly create an instance of CalculateBase. But as this class is currently written, they could. So what we want to do now is take advantage of the abstract modifier. So here where we have our calculate method, we expect everyone who inherits from CalculateBase to override it,



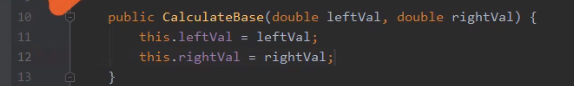
so let's go ahead and mark this calculate method as abstract. Now by marking this method as abstract, that means we don't expect it to actually have a body here in CalculateBase.



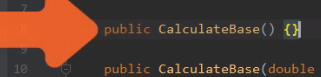
So let's replace those opening and closing brackets with a semicolon. Now one last thing we need to do, since CalculateBase contains an abstract method,



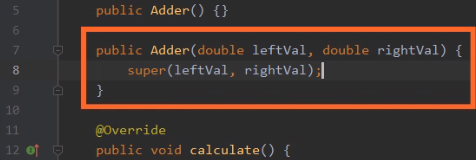
the class itself must also be marked as abstract. So now with that, we can be assured that no one can ever create an instance of CalculateBase. So that takes care of one key issue, but we have another important consideration. The way our classes are currently written, in order to use any of the classes that inherit from CalculateBase, we have to explicitly set the left and right values. It would be nice to be able to construct those instances providing those left and right values. So to do that, let's add a constructor here to our CalculateBase class.



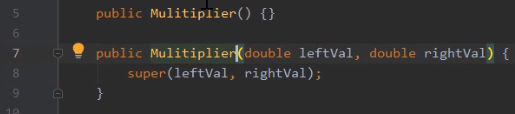
So we'll have a public instructor, CalculateBase, that accepts a left and right value. Then inside this constructor, we'll go and set the left and rightVal fields. So now we have a constructor that accepts the left and right values.

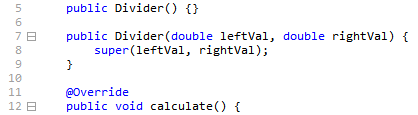


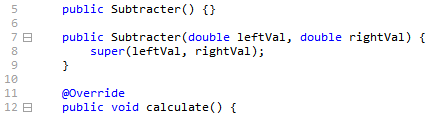
Let's go and add a default constructor as well. So now we have the constructors we need here in our CalculateBase class, but remember that other classes won't automatically inherit those constructors. We'll need to add those constructors to those classes individually. So let's head up here to our Adder class.



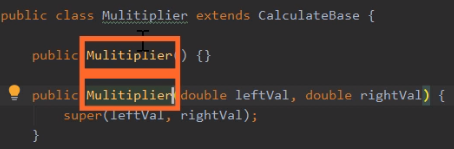
And here in our Adder class, we'll first add our default constructor. Now remember, because this constructor doesn't explicitly call one of the base class constructors, it will implicitly call the default constructor in CalculateBase. But now we also want a constructor that accepts the left and right values, so let's go and add that. And then here in this constructor, we want to call the CalculateBase class' constructor that accepts those values. So to do that, we use the super keyword. Then as parameters, we'll pass leftVal and rightVal. So that takes care of our Adder class. We can create an instance of Adder passing no parameters, or we can create an instance of Adder passing in values for our leftVal and rightVal.







So now I'll go and do the same thing for each of the other three classes. So now all of our classes, Adder, Subtracter, Divider, and Multiplier each have two constructors. They each have a default constructor, as well as a constructor that allows us to specify the left and right values. But now before we finish up here, we have one last little thing we need to do.

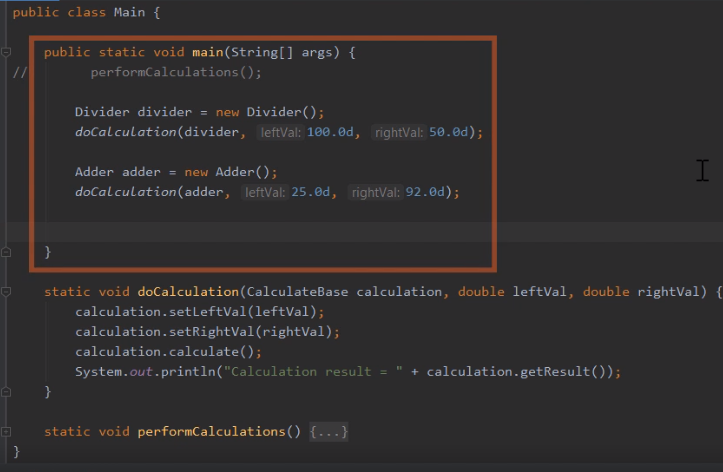


Notice I'm looking here at the Mulitiplier class, and as I mentioned in a previous module, I made a typo in this class name when I created it. Notice that I have an extra i here. And, of course, a typo like that could have far‑reaching impacts in my code. For example, each of the constructors needs to have that same misspelled name, and anywhere I use this class would also have that same improper spelling. And often fixing that sort of thing can be cumbersome, but fortunately, STS makes fixing this sort of thing really easy.

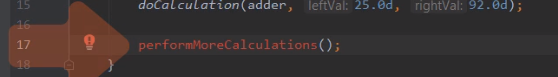


So if we go up here and right‑click on the class name, I'm going to head down here to where it says Refactor, and then I'm going to head up here to where it says Rename, and then I'll choose Rename. You'll notice what this will let me do now is go through here and actually make any changes to the name I need to. So I'll delete that extra i. And you'll notice that when I make that change, STS is automatically making that same change anywhere the name appears. So I'll go ahead and hit Enter, and now that easily, I fixed the typo I made in the name of my Multiplier class. So now in our next section, we'll write some code that takes advantage of the improvements that we've made.

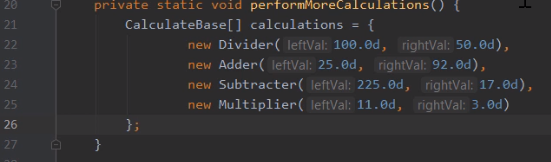
# Using Constructor Inheritance in CalcEngine



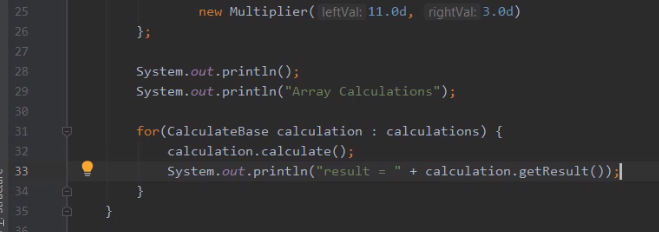
Here we are back in STS, and what we want to do now is use our new and improved MathOperation classes. So now I'm looking at our Main class, and I'm in our main method.



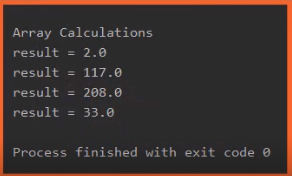
So let's do all that work in a new method we'll create named performMoreCalculations. We'll let STS create the method for us, so hit Alt+Enter, Enter, and then Enter once more. So we scroll up a bit so we have some room to work.



So here I perform more calculations. Let's create an array of type CalculateBase, and we'll name it calculations. And then here in our calculations array, we'll create instances of our individual classes that inherit from CalculateBase. So let's start out by creating a new instance of Divider, then here as we create this instance, let's pass in values for the leftVal and rightVal. So that gives us an instance of Divider that will divide 100 by 50. So now let's go and create instances of the other three classes. So now we have our array initialized to have one instance of Divider, one of Adder, one of Subtracter, and one of Multiplier. So now we can do some work that will actually walk through that array and perform the calculations.



Now before we do that, we just print out a message that indicates we're going to do the array calculations. Now once we print the message, we can go and process our array. So we'll start out with a for loop, we'll declare a variable named calculation of type CalculateBase, and here in our loop, we'll walk through our calculations array. And then here in the body of our loop, our work is really, really simple. We can simply call calculation.calculate, and then we can print out the result of our calculation. And then once we do that, let's go and run our application.



And if we look here at our output window, we can see our results. Dividing 100 by 50 gives us, 2, adding 25 and 92 gives us 117, subtracting 17 from 225 gives us 208, and multiplying 11 times 3 gives us 33. And as you can see, the work we've done has really improved our classes. By making CalculateBase abstract, it assures that no one ever tries to directly create an instance of CalculateBase. And by providing the appropriate constructors, it was very easy to create instances of our individual classes.

# Summary

=>slides: Pg. 27  
Here are some of the key things you want to remember from this module. Remember that we have the super reference. The super reference is kind of like the this reference in the sense that it refers to the current object. The key is the super reference treats that object as if it's an instance of its base class. So it allows us to access method implementations in the base class that we might have overridden in the derived class.

=>slides: Pg. 28

Then we looked at some of the different ways we could override default behavior of classes. One of the things we saw is we can actually prevent inheritance. Remember that by default, a class can be extended. In some cases, we don't want our classes to be extended. So in those cases we can mark that class as final, and once it's marked as final, Java will not allow any class to extend that final class. In other situations, we're okay with having our class extended, but there are certain methods that we don't want overridden. In general, these are methods that we consider critical to behavior of our class. So in these situations, we can mark individual methods as final, and when we do that. people are free to extend our class, but any methods that are marked as final cannot be overridden. Then we saw we actually had the option to require inheritance. So we could declare a class that we never want directly created. Instead, we want it to serve as a base for other classes. We consider that an abstract class. So we use the abstract keyword to mark that class as abstract, and in general, when we have an abstract class, it's because that class has one or more methods that we're not going to provide implementations of. We're going to leave that implementation work to anyone who derives from our class. So in that case, we would have one or more abstract methods. Remember, that an abstract method has a signature and a return type, but it doesn't have a body. So it's the responsibility of classes that inherit from our class to provide that method body.

=>slides: Pg. 29

And then we finished up with a look at constructors. Remember that constructors are not inherited. Each class has its own constructors. So any special constructors we want our class to have, we have to explicitly make them part of the class. We won't inherit those constructors from our base class. And remember that when we construct an instance of our derived class, because that derived class has the characteristics of its base class, a base class constructor always has to be called. Now, we don't have to have an explicit call to a base class constructor, because by default, the no‑argument instructor of the base class is automatically called. But we also have the option to explicitly call one of the constructors in the base class. Remember, we do that by using the super keyword and providing any parameters we want to pass to that constructor. Alright, that wraps up this module. In our next module, we'll take a look at a special kind of type in Java known as enums.

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