6. Understanding Methods

Contents

[1. Introduction 1](#_Toc8244)

[2. Declaring and Calling Methods 1](#_Toc13929)

[3. Parameters 1](#_Toc14698)

[4. Parameter Passing Behavior 1](#_Toc32500)

[5. Exiting a Method 2](#_Toc9313)

[6. Returning a Value 2](#_Toc27884)

[7. CalcEngine with Methods 2](#_Toc22533)

[8. Command-line Arguments 2](#_Toc15099)

[9. CalcEngine Command-line Arguments 2](#_Toc3586)

[10. Running CalcEngine with Command-line Arguments 2](#_Toc10020)

[11. Summary 2](#_Toc21325)

# 1. Introduction

=>slides: Pg. 1-2

Welcome to our next module, Understanding Methods.

Methods are an important part of Java application development. Methods allow us to organize our code so our applications are more maintainable. In addition, methods make it easy for us to reuse code throughout an application. So in this module, we'll start out with a look at how we can declare methods and how we call those methods from within our applications. We'll, then see how we can pass data values into our methods using parameters. And as part of that discussion, we'll see the parameters are passed as what is known as by value. What the implications are to our methods, we try to make changes to those parameter values. We'll, then see the different ways that a method can be exited, including how it can exit a method before all the code in that method is run. We'll, then see how to return back a value from a method, and then we'll finish up. We'll look at command line arguments. And as we'll see, our application's main method receives command line information in much the same way as other methods receive their parameter values.

# Declaring and Calling Methods

=>slides: Pg. 3

Methods are a key building block of our applications. Fundamentally, a method is simply a mechanism for organizing code. We can create a block of code that's accessible from other parts of our application. And because they're accessible from other parts of our application, they enable the creation of reusable code blocks. So we can write a block of code once and access it from multiple other places inside that application. And methods can be specialized because methods can receive data. So you can have that same block of code operating on different data values. And you can even get results back because a method can return data. So an application of any sophistication is going to involve having a number of methods. Now just a quick note on terminology. Sometimes we say method, some people say function. In practice, both of those terms refer to the same thing. So whether you hear method or function, in general, we're talking about the same thing.

=>slides: Pg. 4

Now when we declare a method, we need to give it a name because we're going to access that method using its name. Now the rules for creating the name of a method are just like that of a variable. We're still going to use any numbers and letters, and we're going to use camel case for the name of the method. So when we declare a method, we declare it with a name, and following the name, we're going to place parentheses. Now, parentheses allows it to have a typed‑parameter‑list. What the typed‑parameter‑list does is allow us to pass data into the method. So we can pass data values into the method, and those values we strongly typed because each parameter will have a specific data type associated with it. Now, a method does not have to have parameters. It has to have a parameter list, but it can be an empty parameter list. And all that means is we have the parentheses with nothing in it.

=>slides: Pg. 5

Now, of course, the method's going to need to do some work, so there's going to be a method body. So each method will contain 0 or more statements, and those statements have to be enclosed in brackets. And that's true, even if you only have one statement. A method body is always enclosed in brackets. And then, of course, the method has to have a return type, because as we said, a method can return back a data value. Because Java is a strongly typed language, the method's going to have a return type, and that's going to indicate the type of data that the method will return. But not all methods want to return data, but the method has to have a return type. So in those cases, we'll give the method a return type of void indicating that no data value is actually returned.

=>slides: Pg. 6

So let's take a quick look now at declaring and using a method. So, we'll declare a method named doSomething. Remember that our method has to have a parameter list, so we'll put the parentheses after it, and in this case, it's an empty parameter list, meaning the method receives no data values. Now the method also has a return type, but if we don't want to return any data values, then we'll get a return type of void, and, of course, we have to have the body enclosed in brackets. Now this doSomething method, as it's written here, is a complete and valid method. It has a name, a parameter list, although an empty parameter list, and has a returned value, and although that's a void meaning it returns no data value, it still meets all those requirements. But now let's go ahead and make the method doSomething. So we'll just print out a couple of messages. So we'll print out a message that says we're inside the method, and we'll just go ahead and print out another message that says we're still inside. Now, one last thing we need to do here in order for our code to compile is we need to add a modifier to the method, which is static. Now we'll get into the details of static a little bit later. For right now, it's just something we have to do so that our code will compile. Without marking it as static, there's a number of issues about classes and objects we'd have to talk about that we'll get to shortly, but we just don't want to get into those details right now. So we'll just go ahead and mark the method as static. So now let's go ahead and use our method. So now to use a method, we'll simply use the method name, followed by parentheses. So now we'll call our method as part of the flow of our application code. So let's go ahead and do some other work around our method call. So before we call the method, we'll go ahead and print a message out, and then after we call the method, we'll go ahead and print another message out. So now as your application runs, we'll work through the statements. So the first one we'll do here is print out that "Before method call" message, then we'll get to our doSomething method. So now, at this point, a couple of interesting things have to happen. First of all, the application has to remember where this call is occurring from, and then, once it notes where the call is made from, it'll transfer control inside the method. So it goes ahead and prints out the first message, runs the next line of code, goes ahead and print out that message. Once it hits the end of the method, it needs to look where the method was called from. Once it notes where it was called from, it can just go ahead and run the next line of code after the method call. So now we'll go ahead and print out our last message here. So now that we know how to declare and call a method, in our next section, let's take a look at how we can pass in data to the method.

# Parameters

=>slides: Pg. 7

There are methods, of course, that are going to work with data. And a fundamental part of working with data is variables. But an important thing to understand about variables is variables are scoped to the method where they're declared. In other words, a variable declared in one method is not visible to any other method. Now we, of course, need some way to pass data into our methods, and that's where parameters come in. Parameters enable us to pass data values to our methods. So one method can pass data values to another method. Now parameters are matched up positionally, meaning the first data value passed goes into the first parameter. The second data value passed goes into the second parameter, and so forth. So let's take a quick look at an example.

=>slides: Pg. 8

So I'll declare a method here, showSum, and showSum has three parameters. Two parameters are floats, that's x and y, and the third parameter's an int, which is count. So what we'll do here inside of showSum, so I add x and y together, and we're assigning that to a local variable named sum. Now because sum is a variable, it's not visible outside of our showSum method. Now once we have this sum, we'll go ahead and run a loop, and this loop will run count number of times. Each pass through the loop, it will print out the value of sum. So if in our code, we call showSum, we need to provide this three parameter values. So we have the values 7.5, 1.4, and 3. So when we call showSum, 7.5, the first parameter value, goes into our first parameter x, 1.4 goes into y, and 3 goes into count. So when we call the method, those at the data values we'll use in each of those parameters. So the first thing we'll do here is when we calculate the sum, we'll add x and y together, so 7.5 and 1.4 is 8.9, so sum gets a value of 8.9. And then when we get to the loop, the loop runs count number of times. Well, since we passed in the data value of 3 for count, the loop will run 3 times, printing out the value 8.9 3 times. So now parameters are a key part of using methods. But there are some important things we need to understand about the way parameters are passed. We'll take a look at that in our next section.

# Parameter Passing Behavior

=>slides: Pg. 9

In Java, parameters are passed by value. What that means is the parameter receives a copy of the original value. It doesn't actually operate on that original value. And the reason that's important is when it comes to making changes to parameters inside of methods. If you make a change to a parameter inside the method, that change will be visible within the method, but that change will not be visible outside of the method. So let's take a quick look at an example to see what that means.

=>slides: Pg. 10

So here we're looking at two different sections of the same source file, so looking at our Main.java file. We'll create a method here called swap. Swap receives two integer parameters, i and j, and the job of swap will be to swap those two values, give j i's value and give i j's value. Now, in order to call the method we'll need some values to pass in, so we'll declare 2 variables, val1 and val2. So that means there's a storage area named val1 that contains the value 10, and the storage area named val2 the contains the value 20. So when we call swap, we'll pass in val1 and val2. Now, in order for swap to do its work, we'll need those parameter values. So the system will allocate space for i; it'll allocate space for j. And then it takes these original values and starts copying them. So the value of val1 of 10 is copied over into our parameter i, and the value of val2 of 20 is copied over into the parameter j. So now all code inside of swap will work on these copies. So we'll go ahead and declare a local variable here named k, and we'll set it to the value of i, so that means we allocate a space named k that has that value 10 from the parameter i. We'll then say i = j, so the value i switches from 10 to 20. Then we'll set j = k, so the value of j will go from 20 to 10. So at this point, we've successfully swapped those values. If we were to print them out, the value of i would print out as 20; the value of j would print out as 10. So the parameter values have swapped inside of the method. But what happens when the method ends? Well, all that gets cleaned up, and none of those values get copied back. So at this point, if we print out the value of val1, we still print out that original value of 10. If we print out val2, we still print out the original value of 20. So the important takeaway here is that parameters allow us to pass values into the method, but the method cannot make changes to the parameter values themselves. Okay, so now in our next section, let's take a closer look at how methods exit.

# Exiting a Method

=>slides: Pg. 11

There are three basic reasons that we exit a method. Now, one reason is we reached the end of the method, meaning that there's no more code inside the method to run, and up until now, the methods we've looked at have all exited because there's no more code. We can also be very explicit about leaving a method. We can use what's called a return statement, and when we hit a return statement, we will exit the method at the time we call the return statement. Now, there's also a third scenario, which is that an error occurs. If an error occurs, we just abruptly exit the method because of that error, and in Java that uses a mechanism known as exceptions. So in Java, if the error occurs, an exception is thrown, and we have the opportunity to handle that using what's known as a try catch block. Now, the exception mechanism in Java is a very rich and powerful mechanism, so we're going to talk about that in detail later in the course series. For now, I want to take a closer look at the return statement. So let's take a look at some code.

=>slides: Pg. 12

We've got the same showSum method on screen here that we were looking at earlier in the module. So let's go ahead and add some code to call the showSum method, and we'll go ahead and add some code to print out a value after we return back from the method. So if we run this code, the first thing we'll do is call showSum, that transfers control into showSum to do the work of calculating the sum, then we reach our loop and print out the value of the sum multiple times. Now, once we exit the loop, there's no more code in the method, so we exit the method and resume the flow of control after where the method was called. So this is a case where we exited the method because there was no more code. Now, we can explicitly exit the method by adding a return statement. Now, in this case the return statement is at the very end of the code, so it won't change the behavior of the method. So if we run the code, we call showSum, we go inside here calculate the sum, run our loop. After we exit the loop, we add a return statement, so we exit the method and again resume that flow of control after where we call the method. Now, as I mentioned, it's scenarios like this where the return statement is the last line of code, it's really no different than letting the method exit because there's no more code in the method. But there are times where we want to have the opportunity to exit a method before we reach the last line of code. You may have done all the work that needs to be done, or there may be certain criteria that's not met so there's no work for you to do. Let's see how that would affect our showSum method.

=>slides: Pg. 13

So here in our showSum method, let's change this last parameter value from 3 to 0. So let's look at how that affects the behavior of our showSum method. So if we want to go in here, and we go ahead and run the code, we call showSum, we transfer control inside of showSum, we do the method's main work, which is calculate our sum, but then when we get to the loop here, let's take a close look at what happens. Well, in our loop the first thing we do is initialize i to be 0, and then check the condition, i < count. Well, count has a value of 0, so 0 < 0 is false, so we immediately exit the loop without doing anything, which means we'll exit the method without doing anything useful. Now, in this case the work we did in the method that got wasted, add two numbers together, is very inexpensive work, so doing that work and not using it is not a big deal, but in some cases, the work that you do inside of a method may be substantial, so you don't want to bother doing that work unless you're sure the parameters meet certain criteria.

=>slides: Pg. 14

So let's see what that would look like here inside of our showSum method. So we'll add some extra room inside of our showSum method, and what we want to do is make sure the value of count is not so small that the loop never has an opportunity to display anything. So what we'll do here in the first line of code inside of showSum will be an if statement, and what we're doing is checking to see if count is less than 1, because we know if count is less than 1, then our loop is never going to print anything out. And if that's the case, we don't want to do any of the work in this method. So if count is less than 1, we'll simply return. So that allows us to validate the value of that parameter and exit the loop without wasting our time doing work that doesn't really matter. So if we run this code, we'll first call showSum, transfer control to showSum, check the condition. Well, 0 is less than 1, so that's true, so we'll then hit the return statement and immediately return from our showSum method. So the return statement allows us to control when we exit the method. So now in our next section, let's see how we can use a return statement to return a value from our method.

# Returning a Value

=>slides: Pg. 15

Now as we've mentioned, a method can return a value, and we're going to return that value by using a return statement. Now when it comes to the method return type, it can be a simple type such as a primitive type, but it can also be a more complex type, such as an array. So let's take a look at a couple of examples.

=>slides: Pg. 16

So I've got a method here, calculateInterest. So if we give it the amount of money we've borrowed, the interest rate, and the number of years we're borrowing it for, we want to get back the total interest we have to pay. And for simplicity, we'll just use a simple interest calculation. So to calculate the interest, all we need to do is multiply the amount times the rate times the of number of years, and then we assign that to a local variable named interest. So now to return the interest back is our method return value. We're going to use the return keyword, and then we can simply place the interest variable. So now to use the method, we'll call it just as we have previously. So we'll use the method name followed by its parameters. And then that return value will be returned as if it's being returned by the method name. So we can simply assign that method to a variable. So the variable named result will receive the return value of our calculateInterest method. So if we print out result, it'll print out that interest value, which is 50. Now the return value of a method can use a variable, it can use a literal, or it can even use a calculation. So with that in mind, let's take a closer look at our calculateInterest method. Well the way this method is currently written, we do the interest calculation, assign it to a variable, and then return the variable. Well we can actually simplify this method. Since the only reason we're using this interest variable is to hold the result of the calculation, we can get rid of that variable and simply return the calculation itself.

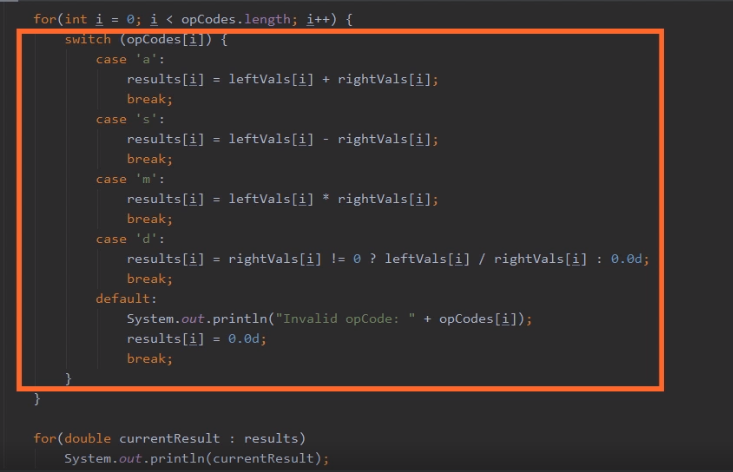
=>slides: Pg. 17

So what will happen here is the calculation will be performed, and then the result of that calculation is what actually gets returned. Now whether we have a simple return value, like a primitive type, or a more complex return value, like an array, returning the value still works the same.

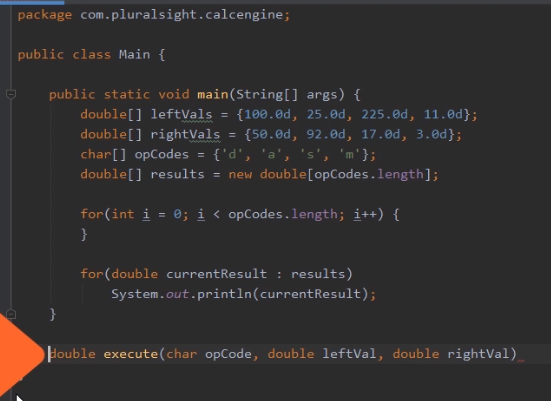
=>slides: Pg. 18

So we have another method here, produceInterestHistory. You notice that the return type of this method is a double array. And what we want this method to do is give us back an array that will tell us the total interest we have to pay year by year. So if the interest is $5 a year, in the first year, we pay a total $5 in interest. Second year, we pay a total of 10. Third year, a total of 15, and so forth. So inside this method, the first thing we'll do is allocate out a double array and assign that to a local variable. Then we'll have a loop that will loop through for each of the years. And inside the loop, we'll calculate the total interest year by year. So once that loop completes, we want to return back the array. We're going to return back the array just like we did this simple value. We simply used to return statement, followed by the array we want to return. So whether our return value is a simple primitive type or a more complex type, like an array, our methods return their values using the same technique. So now in our next section, let's jump back into our CalCengine demo, and let's update that code to start taking advantage of methods.

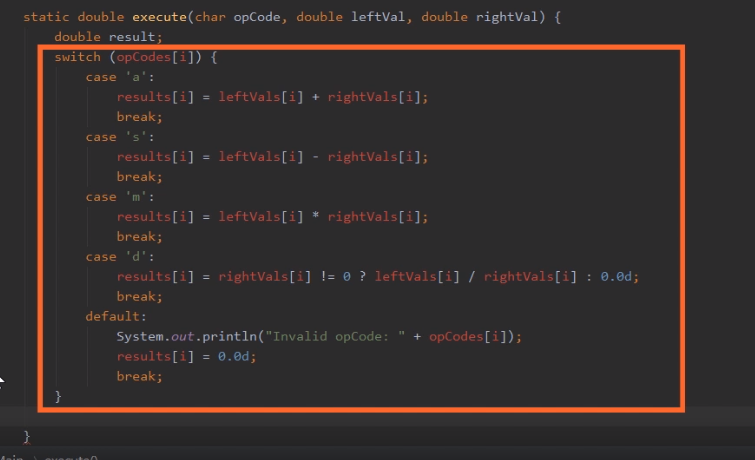
# CalcEngine with Methods



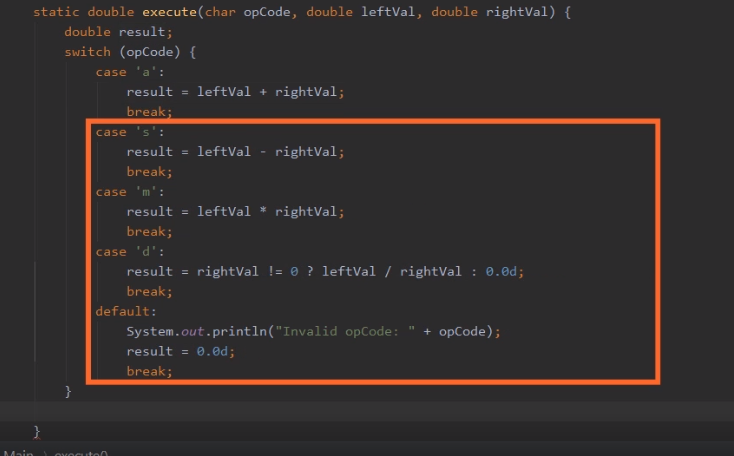
Here we are in STS and we're looking at our CalcEngine project. Now as you recall, our project has this list of arrays at the top to indicate a series of values and the operations we want to perform in those values. So we have to loop through and do all that work. Now let me just scroll down here a bit so we can see exactly the way we do the work. Now, as we look at this code, there is kind of two levels of work going on, there is the larger task that the application performs, which is looping through the arrays and requesting that the operations be performed and that's really represented by our for loop. But then within our for loop, we have the switch statement, which is actually the details of how to perform any given operation. Well as this code is currently written, the switch statement is very tightly tied to these arrays, which means it would be very hard to use this code for any other scenario. So the switch statement is a great opportunity to break the code out into a separate method.



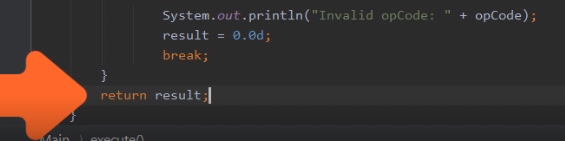
So I'll first highlight our switch statement, so now I'll cut that switch statement from the code, and then what we can do is add a method that handles the details of executing any given operation, so we'll add that just below our main method and let's give it a name of execute. And our execute method will accept three parameters, an opCode, a leftVal, and a rightVal. So now we have an opCode parameter of type char and leftVal and rightVal parameters are of type double. Now we'll have our execute method return back the result of the calculation. So let's give our execute method a return type of double. So now our method has its return type and its list of parameters. Now as we talked about in the slides, for now, we'll also need to go ahead and mark this method as static. Now remember, our method needs a body so we'll add opening and closing brackets. Now remember, we mentioned that our execute method will do the calculation and return the result of that calculation. So let's go ahead and clear our local variable of type double named result. So now with that, we're ready to add the code to do the actual work. So I'll go ahead and paste in that switch statement that we copied from our main method. So now we have our switch statement here, and as it's currently written, it's still working in terms of those arrays that were up in our main method and we want to change this code to use the parameters that we're passing in to our execute method.



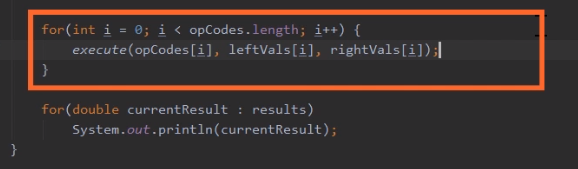
So up here by our switch keyword, let's change that from the opCodes array to be our opCode parameter.



And now, we need to do that same sort of thing in each of our cases. So, like here in our case a where we have leftVals of i and rightVals of i, that should instead be our leftVal and rightVal parameters. And then when we do the calculation, we'll assign the result to our local variable named result. So that takes care of the first case in the switch, let's make those same changes to the rest of the switch statement. So now our switch statement does all of its work using the parameters assigning each result to a local variable named result.



So now the last thing we need to do here is return back result as a return value of our execute method. So now our execute method can receive an opCode, a leftVal, and a rightVal, we'll take care of the details of doing that work and return back the result. So now that we have this in place, let's return back up to our main method. So now we're back here in our main method and we have the for loop here that loops with those arrays.



So all we need to do here in the by the for Loop is call our execute method using the appropriate members of the array. So we'll first call execute and then we'll pass in the appropriate members from each of the arrays. So on the first pass through the loop, we'll pass in the first opCode, the first leftVal, the first rightVal, second pass through the loop, second opCode, second leftVal, second rightVal, and we'll continue doing that for each member of the arrays. So now the last thing we need to do is assign the result of the execute method to the appropriate member of our results array. Then once we do that, we're all set. From a user perspective, the program is not really any different, it will still do the same basic work it did before, it loops through the arrays doing the calculations and displaying the results. But from a developer perspective, our application is greatly improved because it's much easier to read, which will make it much easier to maintain because we've divided the work into separate methods. The overall task of looping through the arrays, initiating the calculations, and displaying the results are in one method, our main method, but then the details of doing any given operation are down in our execute method. So each method is focused on doing one task well. Alright, so now in our next section, let's take a look at using command‑line arguments.

# Command-line Arguments

=>slides: Pg. 19

So now let's take a quick look at command‑line arguments. And command‑line arguments are a way for us to pass information into our programs as part of launching the programs. Now this may seem like an odd topic to talk about as part of our discussion of methods, and so we remember how our applications start. Remember that our applications start with a main method, and this main method serves as a program entry point. And, as you may recall, the main method accepts a parameter, and that parameter is an array parameter, and that parameter receives any command‑line arguments that are passed to the program. So let's take a look at some code.

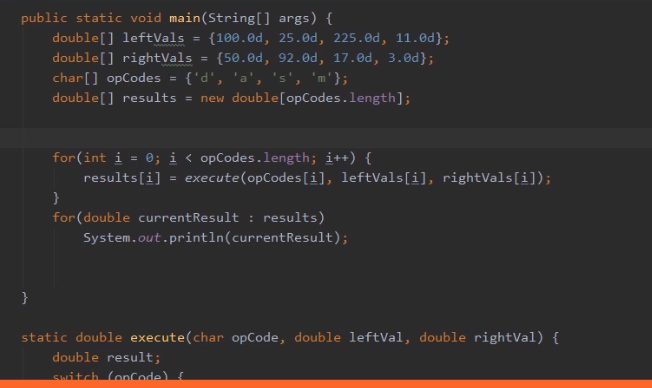
=>slides: Pg. 20

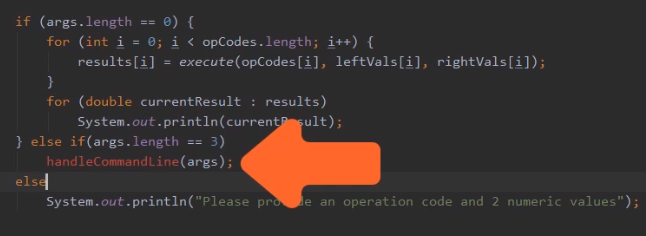
So I've got a main method here, and you know so our main method has a parameter, and that parameter is an array of type string. Now we're going to talk about strings in detail in the next module. The main thing we need to understand at this point is that a string is simply a sequence of characters. Now because our args parameter is an array parameter, that means that each command‑line argument will be a separate element in the array. And as an array, we can do any array type stuff with it. So, for example, we can use this length number to ask it how long it is. So if the length is less than 1, that tells us, well, there were no command‑line arguments provided. But if it's 1 or greater, we know we have command‑line arguments. So we can do things like loop through that array and print out the value of each command‑line argument. Now, command‑line arguments are provided as part of launching the program. So to provide command‑line arguments, the first thing we have to do, of course, is launch the program, and then we provide the command‑line arguments as a list of values after the name of the program class. And each value is whitespace‑separated.

=>slides: Pg. 21

So I have Hello, Mary, and Ann, that would give me three command‑line arguments because they're each whitespace‑separated. But now, in the case of Mary Ann, in some cases, we may want that treated as one argument because sometimes Mary Ann is a single name. Well, command‑line arguments respect the operating system quote characters. So, for example, in Windows, you can quote any string by using the quote character. So I can put quotes around Mary Ann, and that becomes a single argument. So now when I launch the program, the first element in the args array will be Hello, and the second element will be Mary Ann. So as the code runs, the first thing we'll do is check the length of the array. Well, the length of the array is 2, so it's not less than 1, so we'll go down here to our for loop, and we can loop through and print each of those command‑line arguments out. So command‑line arguments are a great way to pass information into our programs. So in our next section, let's update our CalcEngine application to include support for command‑line arguments.

# CalcEngine Command-line Arguments

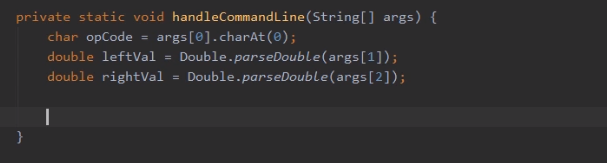
Here we are back in STS, and what we want to do now is add command‑line support to our CalcEngine application. So now what that means is our application will have two distinct modes. If the user provides no command‑line arguments, we'll do the same work we always did, which means we'll walk through these for loops here, processing the arrays and displaying those results. But if the user provides us with the correct number of command‑line arguments, which will be 3, an opCode, a leftVal, and a rightVal, then in that case, we'll do our calculation using the command‑line arguments. Now, remember, we receive our command‑line arguments as a parameter to our main method, so that's this args parameter. So the first we'll need to do is check the length of that args array to see if we received any command‑line arguments.



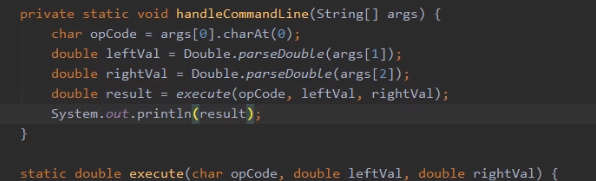
So what I'll do is I'll start with an if statement here to check to see if there are 0 command‑line arguments. So now, remember, if the length is 0, that means there are no command‑line arguments. So we want to loop through the arrays just like we always did. So we'll enclose these for loops in a set of brackets. So that takes care of having no arguments. So now, next, we want to see if we've got the correct number of command‑line arguments, which is 3. So we'll use an else‑if to check to see if the length of the array is 3. Now if we do have 3 command‑line arguments, we'll call a method that we'll create called handleCommandLine. And we want that method to process through the command‑line arguments. So we need to pass our args array as a parameter to our handleCommandLine method. So now there's one other scenario we need to handle. What if we received command‑line arguments, but they weren't the correct number? Well, to handle that case, we'll add an else here, and we'll just print out a message to the user. So now that takes care of all the scenarios. So now we're ready to create our handleCommandLine method.



Now we could create that method manually, but actually STS can help us do that. So let me take my cursor and put it up here where we call handleCommandLine. You'll notice that this red lightbulb pops up. If I go over here and I click on that, notice one of the options I have is Create method. So I'll choose that. Then once I do that, you can see that STS lays out that method for me, and what it wants me to do now is confirm a few aspects of that method. The first thing it wants to do is confirm that the return type I want is void. And I do, so I'll hit Tab. Next, it wants me to confirm the type of the parameter and the parameter name, and I can tab through those as well. So now I'm ready to start implementing the method. And before I do that, let me just scroll up a bit so we have some room to work. So now before we start implementing handleCommandLine, remember that we have this execute method here. And the execute method knows how to do the work of performing an operation. So all our handleCommandLine has to do is get the data from the command line, and then we can pass that data into the execute method for the execute method to do the work. But getting the data from the command line will require a few steps on our part because remember that Java is a strongly typed language. These arguments are coming in as strings. Now as I mentioned, we'll talk about strings in detail in our next module. The main thing we need to understand here is that each individual string is considered a sequence of characters. Even if it only has one character in it, it's treated as a sequence of characters. So we'll need to convert each of these arguments into the appropriate type to use with our execute method. So now, remember, that first command‑line argument will be the opCode.



So let's declare a local variable named opCode of type char. We want the very first argument, so it's args[0]. And because a string is considered a sequence of characters, we need to extract the first character from it. So we use a string method called charAt, and we'll pass in 0. So that long line there, all it really did was convert the string representation of a character into the char representation of a character. So now we need to start getting the numeric values. Remember that our leftVal and our rightVal are each double. So let's declare a local variable named leftVal of type double. And then what we need to do is take that next argument and convert it into a double. Well, Java provides a class named Double that has a utility method named parseDouble, and that utility knows how to convert a sequence of characters into a double. So we want the next argument, so we want args[1]. So that'll take that string representation of a number and convert it into a double. So we'll do the same thing to get our rightVal. So now with that, we've done all the work necessary to convert those string representations into the correct types. So now doing the work will be easy.

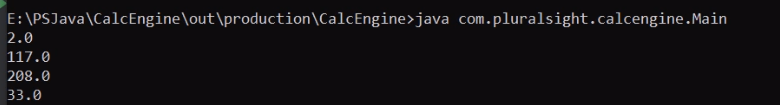


So the first thing we'll need is a local variable of type double named result. We'll call our execute method, and then we'll pass in our opCode, leftVal, and rightVal. And then we can simply print out the result. So that gives us the code we need to process the command line. So in our next section, we'll try running our program to just verify that everything works.

# Running CalcEngine with Command-line Arguments

Here we are back in STS. We're just about ready to run our CalcEngine application to verify that it works as expected. But now, before we do that, let's just quickly walk through the code. Now as you recall, when the user passes in command‑line arguments, they come in as a parameter to our main method and they come in as this args parameter. So down here in our code, first thing we do is check the length of the args parameter. If that length is 0, then we do the same work we always did, which was walking through the list of values in the arrays. But now if that length is 3, that means we have the correct number of parameters and we call our handle command‑line method. If it's anything else, we display an error message to the user.

Now, when we call our handleCommandLine method, we pass in the args array, then down here our handleCommandLine method, we receive that are args array and the main work we had to do here was convert the string representation of those values from the command line into the correct data types that we expect and that gives access to our op code, our leftVal and our rightVal. Once we have those, we could leverage the work we've already done, which was all inside of our execute method. So we call the execute method to do the work and then display the result. So now, before we test out the program, we want to do an explicit build because up until now, we've been running the program from inside of STS. This time, we're going to run it out from the actual command line itself. So to do the build, we'll head up here to our Build menu, I'll choose that, then we'll just choose Build Project. Once that build completes, we'll switch over to the command line. So now we're out here in our Command window in the folder where STS places the output of the build. Now, remember we talked about how to find this folder earlier in this course when we talked about building your first application, but now just quickly as a reminder, the output is placed into your project folder, which for me was PSJavaCalcEngine, and then underneath there, there is an out folder, production folder, and again, the name of the project, which is again CalcEngine. So to run the program, I'm going to use the Java command, my application's main class is com.pluralsight.calcengine.Main. Now just as a reminder, that class name is case sensitive.



So now I can run the program by just pressing Enter, and you notice when I did that, the program did what it always did, it printed out the list of results from the arrays inside the application, and that's because I didn't provide any command‑line arguments, but now let's go ahead and launch the program again, but this time, let's provide some command‑line arguments.



So our first argument is the op code, so I'll provide an a. Now a represents add so let's say I want to add 20 and 30. So I can place a space and the number 20 and then a space and the number 30. So that gives me my three values, the op code of a, a left value of 20, and a right value of 30. So if I hit Enter, I get that result of 50. So it's doing exactly the right thing. So now let's run it one more time, but this time, let's give it the incorrect number of arguments, so I'll give it just one argument.



So now when I run it with one argument, I see the error message displayed. So our application is working exactly as we want it to. We have all the same capabilities we had before, but now we include support for command‑line arguments.

# Summary

=>slides: Pg. 22

To wrap up, here are some of the key things we want to remember from this module. Remember that throughout this module, we've been talking about methods. The methods are a really important part of our job application development because methods give us a mechanism for organizing our code, and in addition, they make it very easy for us to create reusable code. So we can put a block of code in a method and then call that method from throughout our application. Now remember that inside of our methods, we can declare variables. Remember, those variables are scoped to that method, meaning that any variable we declare within a method is not visible outside of the method. So from there we talked about how we could pass information into our methods, and that's where parameters came in.

>slides: Pg. 23

Parameters enable passing values to our methods. And remember that parameters are passed positionally, so when we make a call to a method, the first data value we provide goes to the first parameter, the second data value we provide goes to the second parameter, and so forth. Now remember the parameters are passed by value, and what that means is when we pass a data value to a parameter, a copy is actually made of the original value. So the method never works directly on the original value, it works on a copy. And that's important to keep in mind because that means that any changes we make to a parameter value inside of a method is not visible to the caller when the method returns. Now we, of course, need a way to get data values back to the caller.

>slides: Pg. 24

And that's where our method return values come in. Remember that a method can return a value, and we do that using a return statement. And as we saw, a method can return simple types like the primitive types, but it can also return complex types, such as an array. And whether it's a primitive type or a more complex type, fundamentally we return the value the same way simply using the return statement. Then we finished up, we looked at command‑line arguments. And as we saw, the way our application receives command‑line information is into our main method, and our main method receives that as a parameter. The parameter is an array of type string, and that gives the user the opportunity to pass information into our application. Now as you may recall, as we were talking about string arrays, I mentioned we would talk about strings shortly, and in fact, that's what we'll do in our next module. So in our next module, let's take a closer look at strings.

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