1. Looping and Arrays

Contents

[1. Introduction 1](#_Toc29648)

[2. While Loops 1](#_Toc29373)

[3. Do-while Loop 1](#_Toc24473)

[4. For Loop 1](#_Toc30988)

[5. Arrays 1](#_Toc30153)

[6. CalcEngine with Parallel Arrays 2](#_Toc10432)

[7. For-each Loop 2](#_Toc18181)

[8. CalcEngine with For-each Loop 2](#_Toc28331)

[9. Summary 2](#_Toc14475)

# 1. Introduction

=>slides: Pg. 1-2

Welcome to our next module, Looping and Arrays. When building an application of any sophistication, an important thing we need to be able to do is perform blocks of repeated work across groups of related data. Doing that requires that we use loops, and often we have to manage our data on what are known as arrays. So that's what we'll be looking at throughout this module. Now the exact nature of how we manage a loop will vary depending on the situation, so Java provides a number of different loop types. So we'll start out with a look at the while loop. Then we'll see what's called a do‑while loop, and then we'll look at what's known as a for loop. Now when it comes to managing the data, we'll actually use what's known as an array. Arrays allow us, again, to manage groups of related data. And when we walk through arrays, there's often common sets of work we need to do, so to help with that, Java provides another loop type known as a for‑each loop that simplifies the task of walking through an array

# While Loops

=>slides: Pg. 3

Each of the fundamental loop types in Java do variations of what is basically the same thing. In each case, we provide a condition that we call a control condition, and the loop will continue running as long as that condition remains true. Now there are three fundamental loop types in Java. We have the while loop, which provides our most basic looping behavior. There's something called a do‑while loop, which provides looping behavior, but it defers when the condition for that loop gets checked. And then we have our for loop, and the for loop is similar to a while loop, but actually has a simplified notation for dealing with some of the most common use cases for when we create loops.

=>slides: Pg. 4

So let's look, first at our while loop. When we set up a while loop, the control condition is checked at the start of the loop. So when we create a while loop, we're going to use the while keyword. We'll place our control condition within parentheses, and then as long as that condition is true, we'll, then run the statement that's part of the loop. Now just like when we talked about conditional statements, by default, a loop will only run the one statement that follows it. If you want it to run multiple statements, we have to make this a compound statement by enclosing them in brackets. Now another key thing to keep in mind, because a while loop checks the condition at the start of the loop, it's possible that the loop body may never run. If when your code reaches a while loop, if the condition is already false, the loop body will never get executed. So let's take a look at an example of using a while loop.

=>slides: Pg. 5

And to do that, we'll calculate a factorial value. And if you're not familiar with factorial, in mathematics, the factorial of a number is that number multiplied by each integer below it down to 1. So to calculate the factorial of 4, we multiply 4 times 3 times 2 times 1. So to set that up in code, let's declare two variables. So I have a variable, someValue, and that's the number we want to calculate the factorial of, which in this case is 4, and then we'll declare a factorial, which will actually hold the result of the factorial calculation. Now remember to calculate the factorial, we're going to have to walk through all the integer values from 4 down to 1. So to do that, we'll set up a while loop. And our condition will be to make sure the value is still greater than 1. And then within the body of the loop, we'll do the factorial part of the calculation, and then we'll also reduce someValue's current value. Now we want this while loop to run not just one statement, but both of these statements. So we'll make them a compound statement by enclosing them in brackets. And then once the loop is finished, we'll display the factorial value. So when we run this code, the first thing that would happen is we'd allocate out to someValue variable, and it would have an initial value of 4, and we'd also have our factorial variable with a value of 1. Then when the code reaches the while statement, the first thing we'll do is check the condition and make sure that someValue is greater than 1. Well, someValue has a value of 4, so our condition is true. So we go into the body of the loop. Then we take factorial's current value, which is 1, multiply that by someValue, which is 4, 4 times 1 is 4, and we store that back into factorial. Then we reduce someValue by 1, so we go from 4 to 3, and then the while loop, again checks the condition. Well, 3 is greater than 1, so we go back into the body of the loop. We then multiply 4 times 3, which is 12. Store that into factorial, reduce someValue by 1, so it takes us down to 2. We, again check the condition, 2 is greater than 1. Back into the body of the loop, 12 times 2 is 24. Reduce someValue by 1, which makes it 1. So now we go back up to our condition in our while loop and we check 1 greater than 1, well, that condition is false. So we exit the loop, which will then print out the value factorial, which will display the value 24. So that shows how to use a while loop. In our next section, just take a look at the do‑while loop.

# Do-while Loop

=>slides: Pg. 6

So now let's take a look at the do‑while loop. Now the do‑while loop, like the while loop, continues running as long as the control condition is true. But the key difference is in the case of do‑while, that condition is checked at the end of the loop. Now we set up our do‑while by using two key words, do and while. We place a statement between to do and the while, so that's a statement we want to repeatedly execute within the loop. Now if you want to execute multiple statements within the loop, you will need to enclose those statements in brackets. And then we specify our condition within the parentheses that follow the while. Now the case of the do‑while, because the condition is checked at the end of the loop, the body of the loop will always run at least once. It runs the body of the loop and then checks the condition. So let's take a quick look at an example.

=>slides: Pg. 7

Let's write some code, then we'll simply print out a sequence of values. So we'll start out with our variable iVal; we'll initialize that to 5. Then we'll use our do‑while to run the body of a loop as long as the current value of iVal is less than 25. Now we'll need to run multiple statements within this loop, so we'll enclose those statements in brackets. And then here inside the loop, the first thing we'll do is print out the current value of iVal. Now you notice here when I print out the current value of iVal I'm using print rather than println, as we were in earlier cases. In the case of println, it'll always put a new line after it displays the value. Using simply print allows us to print multiple values on the same line. So we'll start out by printing out the value of iVal. Then we'll print out to text \* 2 =. Well then double the value of iVal and then print out iVal's new value. So if we run this code, again, we started with the value of iVal, the first thing we do is go right into the body of the loop. We haven't yet checked its condition. So we'll print out the current value of iVal, which is 5. We'll print out \* 2 =, then we do 5 \* 2 = 10, store that back in iVal, print that value out, and then we check the condition. Well, 10 is less than 25, so we go back into the body of the loop, we print out 10 \* 2 =, 10 times 2 is 20, so we print out the 20. We again check the condition; 20 is less than 25, so we go back into the loop. Display 20 \* 2 =, 20 times 2 = 40, so we put out to 40. And now we check the condition, 40 < 25 is false, so we exit the loop. Okay, so now that we've seen our loop work, let's make a small change to our code.

=>slides: Pg. 8

Let's go over here where we set the value of iVal. Instead of starting with a value of 5, let's change that to use a value of 80. Now remember, our loop control condition is for iVal to be less than 25. Now 80 is not less than 25, but if we run this code, we still go right into the body of the loop because we haven't checked our condition yet. So as we run this code, display 80 \* 2 =, 80 \* 2 = 160, so we'll print out 160. And only now do we check our condition; 160 is not less than 25, so we'll go ahead and exit the loop. So the do‑while loop is a great choice for scenarios where you always want to run the body of your loop at least once, wait until after the body of the loop is run to check the condition. All right, so now in our next section, let's take a look at the for‑loop.

# For Loop

=>slides: Pg. 9

When we work with a for loop, it, again, is going to check the condition at the start of the loop. The way we set up our for loop is by using the for keyword. We're going to place our control condition within the parentheses that follow the for keyword. Then, of course, we have the statement that makes up our loop body. Just as with the other loops, if we want to run multiple statements, we need to enclose those in brackets. Now because the for loop checks the condition at the start of the loop, again, it's possible that the loop body may never run. So, of course, this sounds a lot like a while loop, and a for loop is a lot like a while loop. But it provides one key benefit. It provides simplified notation for our loop control values. Often times when we're creating a loop, we need to do some initialization work, and at the end of each pass through the loop, we need to update some values. Well, the for statement allows us to do all that as part of the for statement itself. So within the parentheses after the for, we can do our loop initialization work, place a semicolon, the condition comes after that semicolon, and then following the condition, we can place a semicolon and do the update work. So the initialization work is done once before the loop starts, and the update work is done at the end of each pass through the loop.

=>slides: Pg. 10

Now probably the easiest way to see the way a for loop works is compare it to doing the same work with a while loop. So let's say we want to go through a loop for a sequence of values simply displaying those values out each pass through the loop. So in a while loop, we first do our initialization work as its own statement. So we declare a variable, i. We give it an initial value of 1. We'll then set up the while loop itself. We'll then place our control condition in the parentheses that follow the while. Then we'll do the work of the loop. So let's print out our value, i. Then let's say at the end of each pass through the loop, we want to double the value of i. So in this scenario, we initialize our control variable, i, before we start the loop. Our control condition checks the value of i in the while statements. So we keep checking to see if i is less than 100. And at the end of each pass through the loop, we update our control value. We double the value of i. Well, in our for loop, this is going to work the same way. We just have a simpler notation for doing so. So in our for loop, we'll start with our for keyword. We then want to initialize i. So we'll declare i and give it an initial value of 1. We add the semicolon after that. We then place our control condition followed by a semicolon. So as this code runs, we initialize i to 1. We check to see if i is less than 100. As long as it is, we'll do the work in the loop. We just print out the value of i. And then at the end of the pass through the loop, we want to take i and double it. So we provide that code after the semicolon following our condition. So this way, our for statement makes it very clear what our initialization work is, what our control condition is, and what update work we want to do at the end of each pass through the loop. So now with all that in mind, in our next section, let's take a look at using arrays in Java, and as part of our arrays discussion, we'll look a bit more at our for loop.

# Arrays

=>slides: Pg. 11

Oftentimes in our applications, we need to deal with a series of values, and that's where arrays come in. Arrays allow us to specify an ordered collection of values. Each individual value is known as an element. So here, I'm declaring a variable called theVals that could reference an array of floating point values. We indicate that it's an array by specifying the square brackets after the type. So basically, we're saying we're declaring a variable that can reference a floating point array. Now, in addition to declaring the variable, we actually have to create the array itself. We do that by using the new keyword, followed by the data type, and then in square brackets, specify how many elements we want the array to have. So in this line of code, we've declared an array that has three elements and we can reference it by using our variable theVals. Now we need to have a way to access each individual element. We do that by using an index, and Java uses a 0 based index system, which means the indexes will run from 0 to 1 less than the number of elements. So in our example here, the indexes will be 0, 1, and 2. So we're going to access an individual element, we'll have to specify its index. So if I want to put a value into the very first element, I'll use my variable name, theVals in square brackets specifying the index, remember the first index is 0 and then I provide its value. So this stores the value 10 in the element with index 0 and I can do that same sort of thing for the other elements. So I can set the value for index 1 of 20 and the value for Index 2 of 15. Now as we're working with our arrays, we often need to know just how many elements there are in the array.

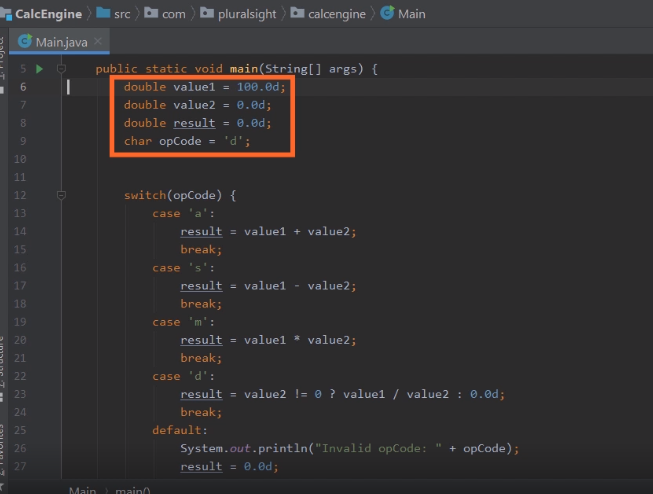
=>slides: Pg. 12-13

So arrays expose a value named length and length indicates how many elements are in the array. So in our example, our arrays length is three. So let's use that array to do some work. Let's just go ahead and sum up the values in the array. So we'll start out with this same array setup that we just had, if we're going to sum up the values, we'll need a variable to hold that sum, and then we want to use a loop to go through and get the value of each element in the array. So to do that, we use a for loop. In our for loop, we use an index to get to each of the elements. So on our for loop, our initialization work, we declare a variable named index and set it to the value of the first index that we want to access, which is index 0, our for loop needs a control condition so we want to keep running as long as the index is less than the length of the array, and at the end of each pass through the loop, we want to go ahead and increment the index. Then the work we want to do inside the loop is add the value of each array element to our sum variable. When the loop finishes, we'll go ahead and print out the value of sum. So as we run our code, we'll set up the array, and then we reach the four loop, the first one we do is the initialization work, so that sets our index to 0, then it checks the condition, 0 less than the length of the array, the length of our array is 3, 0 is less than 3. So we go into the body of the loop, we then add 10 to the sum, which makes our sum 10. At the end of the pass through the loop, we do the update work. So we increment index from 0 to 1, then the condition is again checked, 1 is less than 3, back into the body of the loop. So then we add 20 to the sum giving us a sum of 30. At the end, it is passed through the loop, we do the update. Our index goes from 1 to 2, we check the condition again, 2 is still less than 3, so back into the body of the loop, we add 15 to our sum, we do the update work, our index goes from 2 to 3. Now when we check the condition, three less than three is false. So we go ahead and exit the loop, we print out the value of sum, which will display the value 45. So as you can see, arrays make it easy for us to work through series of values. Now, before I finish up, let's take a closer look here where we declare and initialize the array.

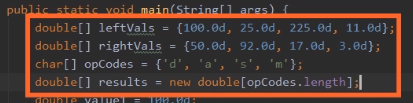
=>slides: Pg. 14-15

Now as this code is currently written, we're doing a number of explicit steps here. We're explicitly sizing the array to have three elements and then we go through element by element, setting each initial value. Now as you might imagine, initializing array with a starting set of values is the sort of thing we have to calmly do in our applications. So because of that, Java gives us access to a shorthand that's simpler than having to do all that work manually. If we want to initialize an array with starting set of values as part of the array declaration, we can simply provide the list of values in closed in brackets. By doing it this way, Java will take care of the details of sizing the array and setting the appropriate value in each of the elements. So to help us get a better understanding of working with arrays, we'll jump into our CalcEngine project and start converting that code to use arrays.

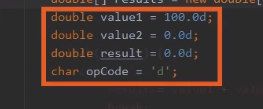
# CalcEngine with Parallel Arrays



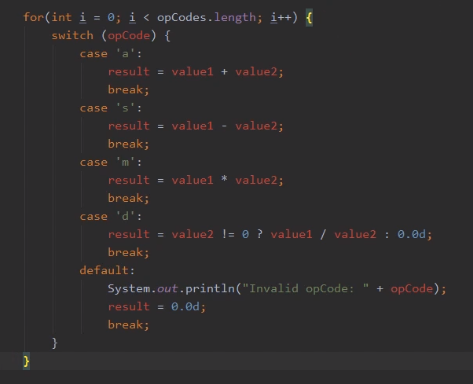
Here we are in STSlooking at the code for our CalcEngine application. Now what we want to do now is enhance the application to allow us to perform multiple calculations, and we'll do that by using arrays. Now, just as a quick reminder, the way the application works is that we currently have two variables declared, value1 and value2, and the plan is to use those to perform calculations to where we indicate what calculation we want to perform by using this variable opCode. So then we have this switch statement that will look at the opCode, perform the calculation using value1 and value2, and then take the result of the calculation and store it in a variable named result. Now each of these variables is just a simple variable, and as a result, the application can only do one calculation each time it's run, but if we replace these with arrays, we can actually run multiple calculations on each program execution. So let's start adding some arrays to the application.



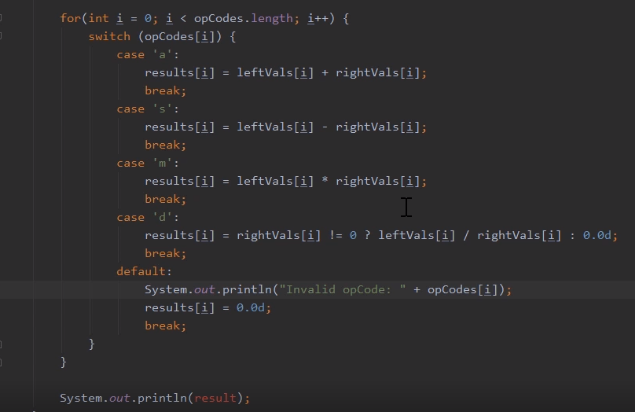
So let's start out with a double array and let's call it leftVals and the reason I've named it leftVals is that if you look at each of our calculations, the variable value1 is always used on the left side of the equation. So what we'll do here with our leftVals array is populate with a series of values that we'll use in the left side of our equations. So that gives us four values in our leftVals array. Let's go ahead and declare a rightVals array that's also of type double. We'll go ahead and initialize it with four values. So that gives our values, so now we'll also need an array for our opCodes. So let's declare a character array named opCodes and we'll initialize it with four opCodes. And then we need one more array to hold our results. So we'll declare a double array named results. Now in the case of our results array, we don't want to initialize it with a series of values because the results array is going to hold the results of the calculations. So rather than give it a bunch of values, we're instead simply going to make space to hold those values. So to do that, we'll create a new double array. Now when we create the array, we want to specify its size. Now we could give it a hardcoded size of 4, but there is actually a better solution here. We know that we're going to have a result for each of our opCodes. So for the size of the array, let's just use the length of the opCodes array. Now, these four arrays, leftVals, rightVals, opCodes, and results are what are known as parallel arrays, and as parallel arrays, what we mean by that is the elements in each array are meant to be used with a corresponding element in each of the other arrays. So that means when we do our first calculation, we're going to get the first member of the leftVals array, use that with the first member of the rightVals array, use that with the first member of the opCodes array, and when we do the calculation, we'll place it into the first member of the results array and that same process will be applied to each of the elements across all of the arrays. So now we have our array variables in place, we no longer need those original variables.



So I'll go ahead and delete those guys out. So now with those variables gone, we'll need to update the rest of our application to use our array variables. Now before we do that, let me just scroll down here a bit. So now once we scroll down, we can see our entire switch statement and what we need to do is place a switch statement in a for loop, and we'll use the for loop to count through the indexes so we can work on the elements within the arrays. So we'll start out with the for keyword.



Since we're counting through the arrays, we'll declare an integer variable named i, we'll give it an initial value of 0. We want to keep running inside the loop as long as we haven't reached the end of the arrays. So we'll keep running as long as i is less than the length of our opCodes array, and at the end of each pass through the loop, we want to increment i by one. So that takes care of setting up our for loop. Let's go ahead and wrap the switch statement in brackets to show that that's all part of the for loop. So with that done, all we need to do now is update all of our variables to use the array variables. So let's go up here where we check the opCode. Remember, that we're walking through our opCodes array.



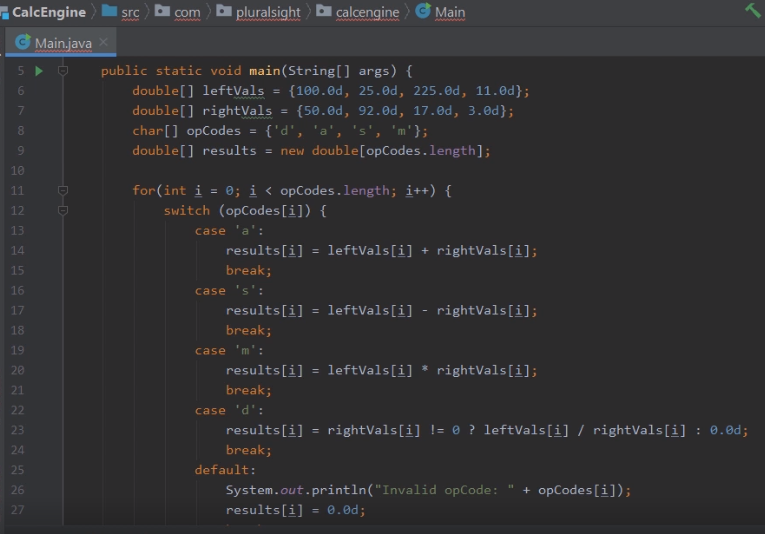
So what we want to switch on is opCodes sub i. So that indicates on the first pass through the loop, when the value of i is 0, we'll use the opCodes value that corresponds to index 0. On the next pass, we use the opCodes value that corresponds to index one, and we'll keep doing that through the end of the opCodes array. And we need to do that same sort of thing for each of our calculations. So if I look here at case a, we'll replace value one with leftVals sub i, and then we'll replace value2 with rightVals sub i. And then once we do the calculation, we want to place the result into our results array at that same index. And once we do the work for case a, we want to do the same thing for all the other cases, anywhere we use value1, we use the leftVals array, anywhere we use value2, we use the rightVals array, and with some of our results, the results array. So now we have all of our code set up to do the calculations walking through each of the arrays. Now the last thing we need to do is print out our results, and we could print out our results using the same sort of for loop that we've used here to do the calculations, but it turns out for printing out the results, there is a better choice known as the foreach loop. And next , we'll take a look at how we use a foreach loop.

# For-each Loop

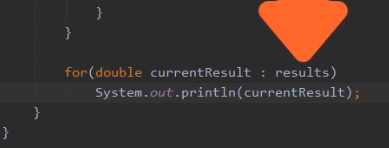
=>slides: Pg. 16

To help simplify working with the members of an array, we have the for‑each loop. And the for‑each loop runs the body of the loop once for each array member. Now one thing to note, as this course here goes on, you'll see there's some additional types that you can use for each with, but for now, we're just going to focus on using for each with arrays. Now when we use a for‑each loop, we're still going to use the for keyword, but when we use for each, we don't have to get tied up in the details of indexes. Instead, we can simply specify the array that we want to walk through, we'll have a colon before that, and then we'll declare a loop variable. And that loop variable's value will actually change throughout the loop, and it will represent the current value in the array for each pass of the loop. So in the body of the array, we can simply process that variable. And the for‑each loop takes care of the details of getting the length of the array, and it takes care of the details of accessing each value in the array. So let's look at the example we used earlier of summing up the values within an array. So here we have that same array, theVals. It has the same three values it had earlier. But now this time as we sum up the values, rather than using a regular for loop, we'll use a for‑each loop. So we'll start out with the for keyword. Remember, we don't have to do anything in terms of initializing indexes or updating the indexes. Instead, we simply specify the array. That's our theVals array. We have our colon, and then we're going to declare a variable here, currentVal. Notice that currentVal's type is the same type of each member of the array. And as we run through the loop, the value of currentVal will change on each pass through the loop. So to sum up all the values in the array, we can simply say sum += currentVal. So on the first pass through the loop, currentVal will have the value of the first element in the array. So on the first pass of the loop, its value will be 10. On the next pass to the loop, it moves on to the next value in the array. So in the second pass through the loop, currentVal's value will be 20. On the third pass, currentVal's value will be 15. And then once the loops loop through all the members in the array, the loop automatically exits. So when we print out the value of sum, we'll, again print out that value of 45. Alright, to get a better understanding of working with for each, in our next section, we'll jump back into our CalcEngine example. We'll update some of that code to use a for‑each loop.

# CalcEngine with For-each Loop



Here we are back in STS, and what we want to do now is update our CalcEngine app to display our list of results using a for‑each loop. As you recall earlier in the module, we updated the application to use arrays. So we have these first three arrays, leftVals, rightVals, and opCodes, and we used a loop to walk through the elements in each of those arrays to perform the calculations specified in those arrays. In each calculation, we stored a result here in our results array. Now when we were doing the work of performing the calculations, we used a traditional for loop because that allowed us to increment an index, which made it very easy for us to access the individual elements across these four arrays. Well now to display our results, all of our results are in one array. So in this case, we can use a for‑each loop. Let's scroll down here towards the bottom of our code, and let's add that for‑each loop.



So we're down here at the bottom of our code. I'm going to start out a for‑each loop. We'll use the for keyword. Remember when using a for‑each loop, we'll need a variable to represent the individual array elements. Well our results array is an array of type double, so we'll need a variable of type double, and let's call it currentResult. Then we'll place a colon followed by the array name. And then here where we print out the value, we're going to simply print out currentResult. And that's all there is to it. The for‑each loop takes care of all the details. You simply walk through the results array, element by element. Each pass through the loop. CurrentResult will contain the value of the current element. So all we need to take care of is doing the work that we need to do, which is printing out the value of currentResult. So now with that in place, let me scroll back up here to the top of our code. So now we're back here at the top of our code, so I'll go ahead and run the app. Once we run the app, we can see our four results displayed. Let's take a look and see if those results are correct. If we go up here to our opCodes array, the first opCode is d, which means division. The first element of leftVals is 100. The first element of rightVals is 50. If we divide 100 by 50, we get the result of 2. Our next opCode is a, which says we want to add the values. So we add 25 and 92, which gives us 117. Then we have our opCode s, which means you want to subtract, 225 ‑ 17 is 208. And then opCode m for multiply. We multiply 11 by 3, we get 33. So our application is working perfectly. By using arrays, a traditional for loop, and a for‑each loop, we successfully updated the application to be able to perform multiple calculations each time we execute the app.

# Summary

=>slides: Pg. 17

To wrap up, here are some of the key things you want to remember from this module. Remember that Java provides a number of different loop types. So the first loop type we looked at was the while loop. Now remember the while loop continues to run as long as its control condition remains true. And in a while loop, that control condition is checked at the start of the loop. In other words, the condition is checked before the loop body ever begins to run. And because of this, it's possible the while loop to the loop body may never run. If the control condition was false when our code reaches the while loop, the code will simply jump right over the body of the loop. And we also have the do‑while loop. The do‑while loop also continues to run as long as the condition is true. In a do‑while loop, the condition is checked at the end of the loop, which means that the loop body always runs at least once. When we use a do‑while loop, our code runs the body of the loop. And then after that first execution of the body of the loop, only then is the loop condition checked. Then we have the for loop, and the for loop is very similar to a while loop. It continues to run as long as the control condition is true, and the condition is checked at the start of the loop. But the key value of the for loop is it provides simplified notation for loop initialization and managing the loop control. Remember in our for statement, we can specify our initialization work, we specify a condition, and we can also specify the work that we want to have happen at the end of each pass through the loop.

=>slides: Pg. 18

From there, we looked at arrays. An array is simply an ordered collection of elements. In other words, it allows us to manage a group or series of values. Remember that in order to access each individual element in the array, we use an index, and the index is simply a number that identifies the position of that element in the array. And in Java, indexes are zero‑based. That means the first element has an index of 0, the second element has an index of 1 and so forth. And then we also looked at another type of loop known as a for‑each loop. And the for‑each loop simplifies working with array members. Now we can, of course, work through an array with a traditional for loop and manage that index ourselves, and there are some cases where that's useful. But commonly, we simply want to walk through an individual array, processing each of the individual elements. That's where the for‑each loop comes in because the for‑each loop handles the details of getting to those array members. It takes care of the details of getting the array's length. It takes care of the details of accessing each individual element in the array. All right, that wraps of this module. In our next module, we're going to see how we can better organize and manage our code using something known as methods.

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