1. Conditional Logic and Block Statements

Contents

[1. Introduction 1](#_Toc15853)

[2. Conditional Logic and Relational Operators 1](#_Toc13136)

[3. If-else Statement 1](#_Toc2149)

[4. Creating the CalcEngine Project 1](#_Toc26968)

[5. Logical Operators 1](#_Toc26135)

[6. Logical vs. Conditional 2](#_Toc4224)

[7. Block Statements 2](#_Toc28936)

[8. Using Block Statements in CalcEngine 2](#_Toc11034)

[9. Switch Statement 2](#_Toc14613)

[10. Using Switch Statement and Conditional Assignment 2](#_Toc11519)

[11. Summary 2](#_Toc16483)

# 1. Introduction

=>slides: Pg. 1

Welcome to our next module, Conditional Logic and Block Statements.

=>slides: Pg. 2

Conditional logic is an essential part of building any application that does anything interesting. Without conditional logic, there can only be one linear path of execution to any program. In order to use conditional logic, there's a few things we need to understand. We'll start out, we'll look at relational operators, which allows us to test a relationship between two values. We'll then look at conditional assignment, which allows us to return back one value or another based on condition. We'll then look at one the most familiar aspects of conditional logic, if‑else. If‑else allows us to check a condition, do one bit of work if that condition is true, a different bit of work if that condition is false. We'll also see how we can chain if‑else's together to test a series of conditions. We'll then look at logical operators, which allows us to create more complex conditions by combining different tests. We'll, then look at block statements which allow us to group multiple statements together. This is a feature we often have to use when working with if‑else. And then we'll finish up, we'll look at switch. Switch allows us to take a single variable and test it against multiple values, and then do the appropriate work for the matching value.

# Conditional Logic and Relational Operators

=>slides: Pg. 3

Conditional logic is what allows us to create applications that are interesting. Without conditional logic, we'd only have one single path all the way through our entire application. And conditional logic comes down to two parts. First of all, we have to perform a test. We need to do some kind of comparison. The comparison is either going to be true or it's going to be false. Then, based on that comparison, we're going to take some action. So when we talk about conditional logic, we're putting these two pieces together. We're performing a test, and then we're taking an action based on the results of that test. Now when we do comparisons, what we're doing is checking the relationship between two values.

=>slides: Pg. 4

So we use these things called relational operators, and there are six relational operators. So let's take a look at each of those. So our first one here is the greater than operator. And that operator's the traditional math greater than symbol (>). If we look at the values we have here, the number 5 is > the number 4. When it comes to characters, the comparison actually uses the Unicode code point. So the letter c has a larger Unicode code point than the letter a, so c is considered > a. Now in addition to >, we also have a concept known is greater than or equal to (>=). And what this comparison says is a particular value is >= than the value being compared to. So 5 is considered >= 4, but 4 is also considered >= 4. And if we look at our character example, c is, of course, >= a, but also a is considered >= to a as well. So just as we have >, we also have less than (<). So 4 < 5, and a < c. We have our less than or equal to (<=). So 4 <= 5 and 4 is also <= 4. Then we have our equal to (==) operator. And the == operator basically asks the question, Are these two things the same? And notice that the operator is a double equal sign. You want to make sure you always use the double equal sign because the single equal sign will have a different meeting. So in our numeric example, 5 is considered == 5, c is considered == c. And notice we have some Boolean examples in this case as well. So true is considered == true. And notice also, though, that false is considered == false because, again, the key thing we're testing here is sameness. And then, finally, we have our not equal to (!=). So that uses the exclamation point and the equal sign. 4 != 5, a != c, and true != false.

=>slides: Pg. 5

Now one of the simplest types of conditional logic we can have in our application is a conditional assignment. The conditional assignment returns a value based on the result of a condition. So the way we write this is we start out with our condition. We follow it by a question mark (?). And then we need to provide two values. The first value is a value we want returned if the condition is true. Then we place a colon. Then the other value we provide is the value we want returned if the condition is false. So, ultimately, we will return a single result based on whether the condition is true or false. So let's take a look at some code. So we have two variables here, value1 with a value of 7 and value2 with a value of 5. And let's say we want to get back the largest value between these two, so we can use a conditional assignment. So in this conditional assignment, we check to see if value1 is greater than value2. If that condition is true, we'll return back value1. If it's false, we'll return back value2. So, in other words, maxValue will have the value of value1 or value2 depending on the results of that condition. So in this case, if we go ahead and print out maxValue, it's going to print out a value of 7 because value1 was, indeed, larger than value2. So the value of value1, the number 7, was assigned to maxValue. All right, so now in our next clip, let's take a look at the if‑else statement and how we can chain together if‑else statements.

# If-else Statement

=>slides: Pg. 6

One of the most common ways we include conditional logic in our applications is by using an if‑else. So basically, Java has an if statement that allows us to conditionally execute another statement. The way we do that is we use the if keyword, and when we provide the condition, we need to place that condition within parentheses and then we provide the statement we want to run if that condition is true. So looking at some code here, we've got two variables declared, value1 with a value of 10, value2 with a value of 4. So when I check to see if value1 is greater than value2, I'm going to use the if keyword placing my condition within parentheses, and then provide the statement that I want to run if the condition is true. So in this case, we would print out value1 is bigger. Now there is also an else clause and that allows us to execute a different statement if the condition was false. You can do that with the else keyword and then we simply provide the statement we want to run if the condition was false. Now the else clause is optional. You only need to provide an else clause if you want to run a different line of code in the case of the condition being false. So in our code up here above, we just simply add the else keyword, the work we want to do if the condition was false, So in this case, we would simply print out that value1 is not bigger.

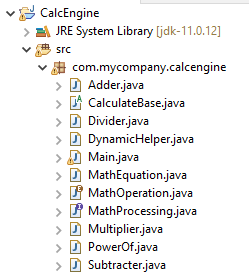
=>slides: Pg. 7

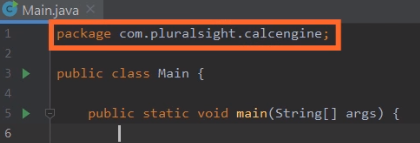
Now, in some cases, we need to test a series of conditions in sequence. Some languages have keywords like else if and so forth that deal with this as kind of a special case scenario. In Java that's not the case. In Java, we do this by chaining if‑elses together. So we can have a series of if‑elses running one right after the other. When we do that, these are evaluated in order, top to bottom. The first one to test true is the one that gets executed. Now we write this kind of like you would expect. We started with the if statement with its condition and what statement we want to run if that condition is true. Then we use the else keyword. And then to test our next condition, we simply do another if with that condition and the statement we want to run if that condition is true. We can repeat this as many times in a row as we need to and we could even optionally have an else that will run the appropriate code if none of the conditions tested as true.

=>slides: Pg. 8

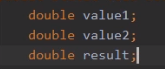
So let's look at a simple example. So again, we have two variables here, value1 and value2. I'll start with my if to check to see if value1 is greater than value2. Now, in this example, that's not the case. Value1 has a value of 10 value2 has a value of 40. So let's say we want to check the opposite condition. So we have the else keyword and then we'll check to see if value1 is less than value2. Now in this case, that's true so we'll go ahead and run that next line of code, which will print out that value2 is bigger. But remember, we can also include an else with no condition, and this else would run if none of the previous conditions were true. So in this example, the code for the else would only run if value1 and value2 were the same. So now in our next clip, let's jump back into STS. Let's create an application that we'll use as our demo throughout the rest of this course.

# Creating the CalcEngine Project





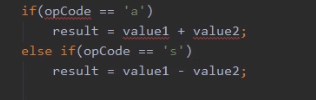
Here we are in STS, and what I want to do now is take a quick look at the behavior of chaining if‑else's together. Now currently on screen, I have a newly created project, a project I've named **CalcEngine**. Notice its package is com.pluralsight.calcengine, and this CalcEngine will be a demo that we'll visit a number of times throughout this module and throughout this course. Now what we're going to do with CalcEngine is build a system that's able to perform a variety of mathematical calculations. We'll start out with support for some simple calculations, but as time goes on, we'll be able to handle increasingly complex calculations. Now our focus here will be on the basic math operations. So to do that, we're going to need some variables.



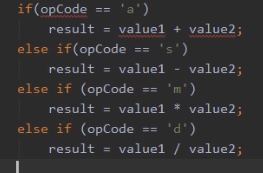
So let's declare three variables‑‑value1, value2, and result. So we have these three variables all of type double. Value1 and value2 will be the two values we use to perform a calculation, and, of course, the result will go into our variable result.



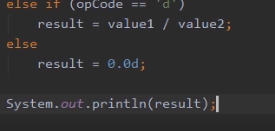
Now to indicate the kind of calculation we want to do, let's use a simple one‑character opcode. So we'll declare a variable named opCode of type char. So with our opCode variable, we can use things like an a to indicate an add or an s to indicate subtraction. And then to perform the actual calculations, we'll just have a series of chained if‑else's together.



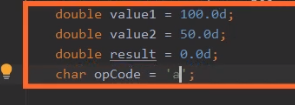
So let's start out with checking for the add operation. So I add an if that looks for an opCode of a. And if we have an opCode of a, we'll simply add value1 to value2, assign the result to result. Now if the opCode is not an a, we want to check to see if it's one of the other codes. So we'll start out with an else, and then we'll add another if to check to see if the opCode is an s. And if the opCode is an s, we'll go ahead and subtract value2 from value1. So that takes care of our add and subtract.



Let's go and do the same thing for multiplication and division. And so now with that, we have support for four opCodes.



Now, of course, it is possible that someone might provide an opCode that we don't support. So in that case, let's just go ahead and explicitly set the result to 0. And then, finally, let's just go ahead and print out the value of result. Now you'll notice here that our opCode value1 and value2 our all underlined in red.



So let's hover over those, and let's see what the problem is. You'll notice the problem is we haven't initialized them. So let's go ahead and initialize value1 to have a value of 100, value2 to have a value of 50. Let's give result an initial value of 0. And let's start out with an opCode of a. So with that all in place, when we run our program, we're indicating we want to do an add operation, so we should add 100 to 50 and print out a result of 150. Let's go ahead and run it and see if that's the case. And we printed out 150. So we know our add handling is working correctly.



Let's try changing our opCode from an a to a d. And so now when we run our program, we should do division. I'll go ahead and run it. And when I run it, we get that result of 2, and 100 divided by 50 is, indeed, 2. So our application is working just like we want it to. Okay, so we're off to a great start. So now in our next clip, let's take a look at the logical operators.

# Logical Operators

=>slides: Pg.9

Earlier in the module, we looked at the relational operators. What we'll look at now are the logical operators. What logical operators do is they allow us to take two true or false values and resolve them into a single true or false value. So what this allows us to do is include more complex conditions in our code. One of the most common things we'll do is combine two relational tests. So we can do one relational test, combine it with another to arrive at a single result, but we can use it with any pair of true or false values, so we can also use logical operators when we're dealing with multiple Boolean variables. Again, we're simply combining two true or false values to arrive at a single true or false value. Now let's look at the individual logical operators, and there are four basic logical operators. So the first one we have here is the and logical operator. Notice that the operator is a single &, and this operator only resolves to true if you combine two true values. So let's take a look at an example.

=>slides: Pg. 10

So we're going to have three variables declared and notice I'm declaring these variables in a way we haven't seen before. Up until now, when we declare multiple variables, each variable is declared on a separate line, but notice here, as long as they're all the same type, I can declare multiple variables on a single line. So here, I have three variables, each initialized to a separate value, but they're all of type int. So now I'll go ahead and add an if condition here and it includes two relational tests, and I want to combine these relational tests into a single result and I'm doing that using our and logical operator. So to do this, we first go one side of the condition so we'll look at a greater than b, well a is 20 and b is 14, so a greater than b is true. Now look at the other side of the condition, well b is 14 and c is 5, so b is greater than c so that resolves to true. Then I combine these with the and, and the and is true if both sides are true so this resolves to true. So my condition is true, so now I'll run that next statement, which in this case would print out a is greater than c. Now in addition to the and operator, we also have the or operator. Notice this is a single vertical bar, and in the case of the or operator, this resolves to true. If one side is true, the other side is true or both sides are true. So we have false or true that's true, we have true or false that's also true, and even true or true resolves to true. Now we also have an exclusive or what some people call an XOR, notice that's the caret symbol as the operator. And the exclusive or is true if one side or the other is true, but not both. So false or true is true, true or false is true, but when doing exclusive or true or true would not be true. And then we have the negation operator and this is one that sometimes creates some confusion. What the negation operator does is reverse the value. So the only thing that resolves to true with a negation operator is false.

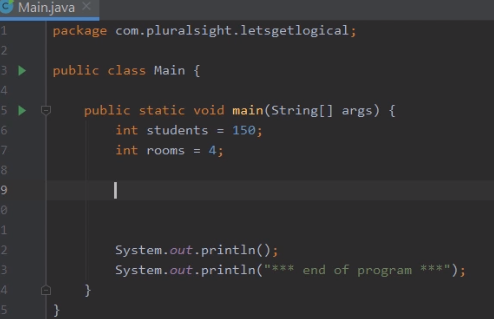
=>slides: Pg. 11

So let's take a quick look at some code to see how that works. So I've got a Boolean variable here named done and I've initialized it to false. So my if condition, I have if not done. Well, the first thing we do is look at the done, done has a value of false, and then we take the negation operator and apply to that. Well, the negation operator reverses it, so not false is true. So in this case, we would go ahead and run that next line, which would print out the message, Keep going.

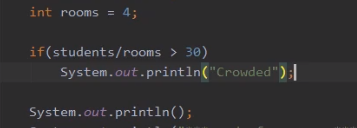
=>slides: Pg. 12

In addition to the standard logical operators, we also have the conditional logical operators and these conditional logical operators are very similar to the standard logical operators. Now there are only two of these, there is an and along with an or, but you'll notice there is a key difference in the operators. The standard and was a single ampersand, the conditional and is a double ampersand. In the case of the or, the standard or was a single vertical bar, the conditional or is a double vertical bar. What makes the conditional logical operators different is they only perform the right side of the operation when it needs to. In other words, if it knows the result of the logical operation by the left side and never bothers doing the right side. So in the case of the conditional and, it will only execute the right side when the left side is true because it would know if the left side was false, the condition is not going to pass. So we take our double ampersand, again, that's our conditional and, as long as the left side resolves to true, it will then check the right side, if that also resolves to true, then the result is true. Now, in the case of the conditional or operator, it executes the right side only when the left side is false because if the left side was true, it knows this operation would resolve to true. So we have a conditional or operator, if the left side is false, it will go ahead and check the right side, if that resolves to true, then the result is true. If we take our conditional or operator and if the left side is true, it won't even bother to check the right side because the operation has already resolved to be true. So to get a better understanding of all this, in our next clip, let's take a closer look at the standard logical operators versus the conditional logical operators.

# Logical vs. Conditional

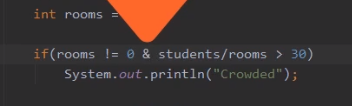


Here we are back in STS. You notice we're looking at another new project here, LetsGetLogical. Now we'll get back into our CalcEngine demo shortly. But before we do that, I want to quickly look at the behavior of logical operators. Specifically, I want to compare the behavior of the standard and operator without a conditional and operator.



So in this program we have here, notice we have two variables at the top, students in this slides to 150 and rooms in this slides to 4. Let's say I want to do something very simple. We're going to determine on average how many students would be in a room. If that number's greater than 30, I want to print out a message that says crowded. So we'll start out with an if statement. Then right here in our if, we'll divide students by rooms. And let's check to see if that result is greater than 30. And if the result is greater than 30, we'll print out the message "Crowded". Alright, so with that, let's just go ahead and run the program. And you can see by our results, we printed out our message 'Crowded". So that's working fine. We can perform that calculation, dividing students by rooms directly here in our if, and then take that result and compare it to greater than 30. But now let's make a small change to our program.



Instead of there being 4 rooms, let's set rooms to be 0. And once I do that, I'll go ahead and run the program. And you'll notice we get an error. Because just like in regular arithmetic, you can't divide by 0, computer‑based arithmetic can't divide by 0 either. So what we really need to do is first make sure that rooms is not equal to 0. 

We'd like to do that right here in our if statement. So in our if statement, let's start out with rooms != 0. So we've got one condition, room != 0, then another condition, the result of students/rooms > 30. And what we want to do is and these together. So let's use the standard logical and, which is a single ampersand. So now we're adding those two conditions together, and only if both conditions are true should we print out the word "Crowded". Now before we run this, let me head down here to the bottom, and I'll go ahead and clear out this bottom window. To do that, I'll right‑click, I'll choose Clear All. Now I'll go ahead and run it again. And you'll notice we still get the error. And that's because we used a standard logical and. When you use a standard logical and, both sides of the comparison always happen. So even though rooms != 0 tested to false because rooms is equal to 0, we still went ahead and tried to do the division.



But now if we take that standard logical and and instead use a conditional logical and, which we can do by simply going from one ampersand to two, and now we'll go ahead and run our program, notice that our program now runs to completion. Now we didn't print out the message "Crowded" because we couldn't determine how many students were in the rooms, but we didn't crash. And that's because by using the conditional logical and, when we did the evaluation to see if rooms != 0, it tested to false. Because that tested to false, we never bothered doing the right side of the condition. Alright, so now in our next clip, let's take a look at block statements.

# Block Statements

=>slides: Pg. 13

Java enables us to create what's known as a block statement. What a block statement does is it groups multiple statements together, creating, in effect, a compound statement. And the reason this is important is because certain parts of the Java language only operate the one line that follows it. In the case of the if statement, the only line that's associated with the if is the line that follows. So we need some way to group multiple lines together, and that's where the block statement comes in. We create a block statement by enclosing statements in opening and closing brackets. So basically we can have a series of independent lines so they're not really grouped in any way. So if we enclose them in these opening and closing brackets, some people call these curly brackets, some people call them squirrelly braces, but basically enclose the statements in these opening and closing brackets, and what will happen then is that Java will now treat them as a block statement, meaning that they're all grouped together. Let's take a look at some code to see how this works.

=>slides: Pg. 14

So I've got two variables declared here, v1 and v2. I'm going to go ahead and declare a third variable named diff. Notice that diff is final, meaning we can only assign to it once. So let's go ahead and add a condition, if v1 is greater than v2. And if that condition is true, I want to calculate the difference by subtracting v2 from v1, then printing out the message v1 is bigger than v2 and displaying that difference. But we also want to handle the condition not being true, so we'll have an else, and for the else we want to calculate the difference by subtracting v1 from v2, and then printing out the message v1 is not bigger than v2, and then showing the difference. Now this code as it's written has some significant issues. Let's first look at the if statement. Remember we said when the if is true, we want to calculate the difference and print out the message, but as this code is written, the only thing associate with the if is the diff calculation, because remember, an if only runs the one line that follows it. Now, in addition to it not including all the correct lines, this code won't even compile. The else would actually not be valid, because it's not associated with the if. The reason is there's more than one statement between the if and the else. So if we want to run these multiple statements with the if, we've got to group them together somehow. That's where the block statement comes in.

=>slides: Pg. 15

So we'll put an opening bracket after the if and a closing bracket after the last line we want to associate with the if, and now that turns those two statements into a block statement. So by doing that when the if is true, both the diff calculation and that print line will occur. Also, the code will now compile because the else is associated with the if. But there's still an issue, because that else will only run the one line that follows it. So if we want the else to do the diff calculation and print out the value, we again need to create a block statement. So we'll have the opening bracket after the else, the closing bracket after the print line, and then by doing that, they become a block statement, so both of those statements are now associated with the else.

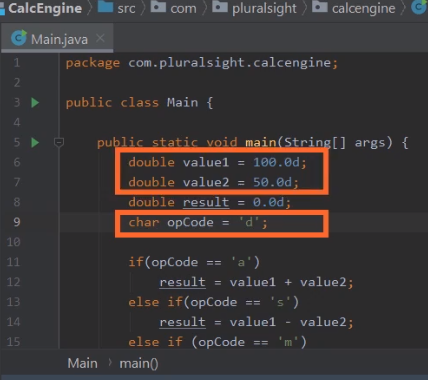
=>slides: Pg. 16

When we use a block statement, it impacts something known as variable scope. And when we talk about variable scope, what we're talking about is where is a particular variable visible from. What's that variable's range of visibility? In the case of a block statement, when you declare a variable within the block statement, the scope of that variable is limited to that block. In other words, the variable cannot be seen from outside the block. If you try to use it from outside the block, you'll actually get a compile error. So let's take a look at some code.

=>slides: Pg. 17

So I've got some code here that declares two variables, students and rooms, and I have a condition here where I check to see if rooms is greater than zero, and you notice that I've got a block statement that I've started and ended after the if statement. Now inside the block statement I'm printing out both students and rooms. Students and rooms are declared outside the block, but I can still use those inside the block, so variables are allowed to come inside of a block. But let's go ahead and add another line inside the block. Let's go ahead and divide students by rooms. We'll assign it to a variable named avg, and notice that avg is declared inside the block. The scope of that variable avg is limited to the block. So if I tried to print out the value of avg after the end of the block, I would actually get a compile time error, because avg is not visible once the block ends. So to solve this problem, I've got two choices. I could declare avg before the if statement, because remember a variable is allowed to flow into a block, or I could simply take the print line and move it up inside the block itself, and that takes care of the issue. So, again, variables declared before a block are visible inside that block, but any variable declared within a block is not visible outside of the block. Alright, so now in our next clip, let's get back into IntelliJ and we'll take a closer look at block statements.

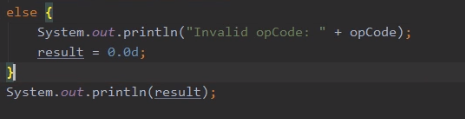
# Using Block Statements in CalcEngine



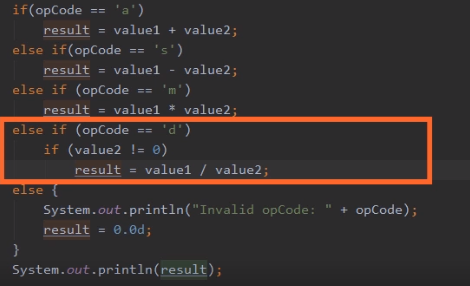
Here we are back in STS, and we're again looking at our CalcEngine project. The last time we ran it, we used an opCode of d, indicating division, and the values we were working with was 100 divided by 50, which as we can see, displayed that result of 2.0. So our application's working just like we want it to in this scenario. Now, to see our source code a bit better, I'm going to go ahead and collapse this window here at the bottom, and then once we do that, we can see our entire if else chain. And notice here at the end of the if else chain, we have a simple else. In that case, we simply set the result to 0, and this is meant to handle the situation when we have an invalid opCode. Well, if we have an invalid opCode, rather than simply setting the result to 0, we should probably go ahead and print out a message.



So let's just go ahead and print out a message that says invalid opCode, and it shows the opCode. So now what should happen is that if it's a valid opCode, we'll perform the appropriate math operation, and if it's an invalid opCode, we want to print out the invalid opCode message and set the result to 0. But let's see what happens now when I go ahead and run it. And you'll notice it prints out a result of 0, even though I had a valid opCode of d. Well, let me go ahead and collapse this window here at the bottom and scroll a code down so we can see what's going on. Now, once we do that, we can figure out pretty quickly what the problem is.



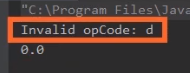
Even though in the case of this final else, I wanted that println statement and the result set to 0 to run, as it's currently written, that's not the case. Remember that the else will only run the one line after it. So what that means is setting result to 0 is actually not part of the if else chain. It actually runs every time. So to deal with this, all we have to do is put the code related to the else inside a block statement. Then once we do that, we'll go ahead and run it again, and now when we run it, we again get the appropriate result, which is 2.0



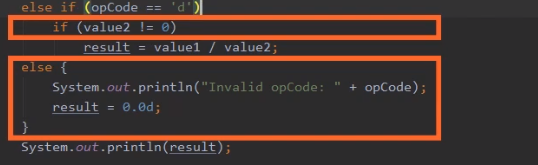
. Okay, so I'll go ahead and collapse the window at the bottom and scroll our code back down. Okay, so now let's take a closer look where we handle the opCode of d. Remember, in the case of that opCode, we divide value 1 by value2, and as we saw in a demo earlier in this module, it's important that we make sure that the value we're dividing by is non‑0. So what we should do here is have an if to make sure that value2 is non‑0. So now if the opCode is d, we check to see if value2 is non‑0, and as long as that's the case, we'll go ahead and do the division. Let me go ahead and run it again, and let's see what happens. And as we can see, our program is still running correctly. Let me scroll back here to the top of our source code.



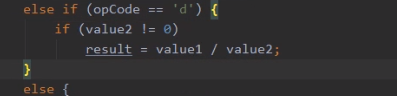
And up here at the top where we initialize the variables, let's try setting value2 from 50 to simply be 0. So now with value2 being set to 0, that if that we've added should make sure we don't do the division. So let me go ahead and run it again



And you'll notice the result is 0, which is correct, because we didn't want to do any division, but we've also got this message invalid opCode, but it's telling us that d is an invalid opCode. So the question is, what's happening here? Well, let me go ahead and collapse this window here at the bottom.



So if we look at what we have here, we have the else if where we check to see if the opCode is d. Underneath there we have the if to check to see if value2 is 0, and then we have the else for dealing with the invalid opCode. And the problem is that else is connecting to the if that's closest to it. So the else is actually connecting to the check to see if value2 is non‑0. And that's not what we want to have happen. We want that else to be tied to the if for the opCode of d.



So what we need to do is enclose the check for value2 not equal to 0 inside of a block statement. With that block statement in place, the else for the invalid opCode is now associated with the if for where we check to see if the opCode is equal to d. So let's go ahead and run it now and make sure everything works as we expect. And you can see it now does the correct thing. With a value2 being set to 0, we display a result of 0, and we no longer display that message indicating that the opCode was invalid. All right, so next , let's take a look at something known as a switch statement.

# Switch Statement

=>slides: Pg. 18

Now let's take a look at something known as a switch. And a switch is a way for us to implement conditional logic, but what it does is allow us to test a single value against multiple matches. So the way we set this up is with a switch keyword, but when doing a switch within the parentheses that follow it, we don't put an actual condition. What we're going to do is put the variable that we want to check to see its current value. And then within the switch, we list a series of cases. Each case provides a potential matching value. So what the program will do is when it hits the switch statement, it'll look at the current value that's in the variable, find the case that matches it, and then transfer control to that match, and then run all the statements within that case until it hits the break keyword. The break keyword indicates that we want to jump out of the switch. Now, switches can have a default case. That's simply the code you want to run if none of the listing matches are matched.

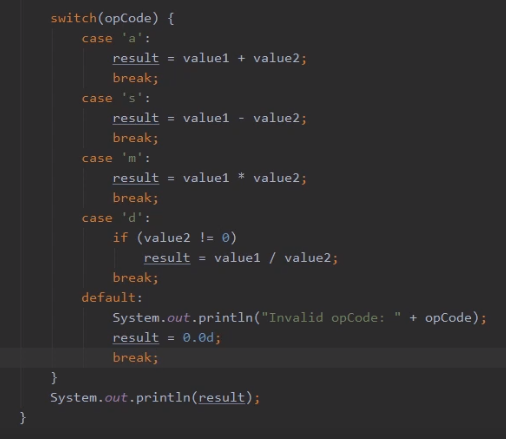
=>slides: Pg. 18

Now, there's a few really important things you want to understand about the switch statement. First of all, switch can only be used with certain data types. Of the primitive data types we've talked about, the only ones that can be used with a switch are the integer types, like byte, short, int, and long, as well as the char data type. Any of the other primitive data types cannot be used with a switch. Now, as we go throughout the series, we'll learn about some other data types that are not primitive types that can be used with a switch, but these are the only primitive types that can be used with a switch. Now within our switch, each case is able to run multiple statements. So unlike an if statement where we had to use a block to run multiple lines for an if, that's not how it works with the case. Within the case, there can be multiple statements. But we want to be sure to end each case's statements with a break. Because if we don't do that, our code will do what we call "fall through." And what that means is it would find the match, start running the code that's within that case, and if there was no break statement there, it would just run right through the next case. So it would basically run all the code that remains through the end of the switch statement. And although there are rare cases where that's what we want to do, that's pretty uncommon. So the statements in each case section are almost always going to end with a break. So now to get a better understanding of all this, in our next clip, let's jump back into our CalcEngine project, and let's convert that to use a switch.

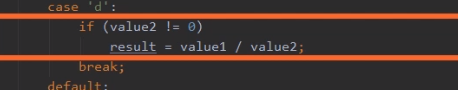
# Using Switch Statement and Conditional Assignment



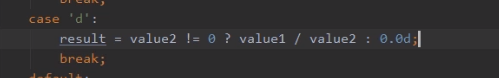
Here we are back in our CalcEngine project, and what we want to do now is convert the project to use a switch statement, and if we look our code as it's currently written, we have a series of if‑elses chained together, and for each of the ifs in that chain, we're always checking the value of the same variable, we're always checking the value of the variable opCode. Well, since we're always checking the same variable, rather than using a series of if‑elses chained together, we can instead use a switch statement, so we'll do that by starting out with a switch keyword.



Within the parentheses for the switch, we'll identify the variable we want to test, which is our variable opCode. Remember that the body of a switch is enclosed in brackets, then here inside of the switch, we want to provide the values that we want to test opCode against. And so, the first one we'll test for is a value a. So here where we currently have if opCode equals a, we'll instead, make that a case for a. So now if opCode had a value of a, it would jump to this case, run the code within that case, which is where we add value1 to value2, and then we want to make sure that it doesn't fall through. So after we do the work we want to do, we'll add a break. So that takes care of an opCode of a. So now we want to do the same sort of thing for s. And then we want to do the same thing for both m and d. So now with that, we have all four of our opCodes handled. Now let me just scroll down here a little bit. And once I scroll down, you can see the code we have here that we want to run if the opCode that's provided isn't valid. Well, in the case of working with a switch, we don't use else, we instead use default. So if we get an invalid opCode, we'll print out that message and we'll set our result to 0. So let me just scroll up here now so we can see our entire switch statement. So now, looking at the entire switch statement, you can see the specific work we're doing is more clear than it was in the if else chain. By using a switch, it's very clear that we're always testing opCode and it's very clear what work we do for each value of opCode. So this takes care of our switch statement, but there is another way we can actually tighten this code up a little bit.



Notice here in the case d, we have that if statement where we only do the division if value2 is not equal to 0. Well, this is a great chance for us to use the conditional assignment we talked about earlier in this module. So let me remove that if statement that we currently have there, but with that gone, let's assign result using a conditional assignment.



So we'll start out with the result. As a conditional assignment, we want to place the condition we want to test and the condition we want to test is that value2 is not equal to 0. Well, if the condition is true, we want to divide value1 by value2, and then we provide the value we want to return if the condition is false, but if the condition is false, we'll simply return 0. Then we'll go ahead and delete the blank line. So now our application does the same work it was doing previously, but the code is much cleaner and easier to read.

# Summary

=>slides: Pg. 20

To wrap up, here are some of the key things you want to remember from this module. Remember that one of the first things we looked at was conditional assignment. The conditional assignment was an easy way to return back a specific value based on a condition. We could return back one value if the condition was true, a separate value if the condition was false. Then we looked at if‑else, and if‑else allows us to conditionally execute a statement. So we have an if with a condition, if the condition is true, we then run the statement that follows it, and we could optionally have the else. If we had an else, then if the condition was false, we'd run the statement that followed the else. Remember that we could actually chain if‑elses together, which allows us to have a series of conditions testing each of those conditions in order. The first condition to test the true is the condition whose work is done.

=>slides: Pg. 21

Now in order to use any kind of conditional logic, we need a way to test conditions, and that's where relational operators come in. Relational operators allow us to compare one value to another. So these were things like greater than, less than, equal to, not equal to. Now relational operators allow us to test individual conditions, but sometimes we need to combine conditions, and that's where the logical operators came in. Logical operators allow us to produce a single true or false result from two true or false results. So basically, logical operators allow us to combine conditions. Now we have the standard logical operators, but we also have conditional logical operators, and conditional logical operators perform logical tests similar to the standard logical operators. In both cases an and only returned true if both the left side and the right side are true, an or will return true if one side, the other, or both are true. The key difference is how they go about doing the details of the work. In the case of the conditional logical operators, they only do enough work to determine the outcome. In other words, they always do the left side test, but they only do the right side test if it's needed.

=>slides: Pg. 22

Now we also talked about block statements, and block statements group statements together, and this is important when doing ifs and elses because an if and an else each only run the one line that follows them. So if we want to run multiple lines, we have to put those lines within a block statement. Now remember that block statements affect variable scope. Remember that variable scope is a variable's array to visibility, and if you declare a variable within a block statement, its scope is that block statement. In other words, it's not visible outside of the block, and then we finished up with switch. Remember that switch allows us to test a single value against multiple matching values, and the switch will transfer control based on the matching value, and then when it finds that match, it will run the statements that follow the match until it reaches a break statement. So in practice, the work for each match normally ends with a break. All right, that wraps up this module. In our next module, we'll dig into looping and arrays.

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