7. Working with String

Contents

[1. Introduction 1](#_Toc23338)

[2. String Class 1](#_Toc30187)

[3. String Equality 1](#_Toc12748)

[4. String Methods and String Conversions 1](#_Toc6376)

[5. Adding String Support to CalcEngine 1](#_Toc5578)

[6. Making CalcEngine Interactive 2](#_Toc16691)

[7. Running CalcEngine Interactively 2](#_Toc28354)

[8. StringBuilder 2](#_Toc21735)

[9. Building String Output in CalcEngine 2](#_Toc7022)

[10.Summary 2](#_Toc19616)

# Introduction

=>slides: Pg. 1-2

Welcome to our next module, Working with Strings. Most any application you write is going to have to work with some kind of text, and often we need to do a great deal of work with text. So we need to know how to do that work effectively and efficiently. And in Java, strings are how we do that. So we'll start out, we'll look at the String class itself. We'll then take a look at string equality, and what we'll find is comparing two string values to see if there are equal behaves very differently when we compare primitive types for equality. We'll then take a look at some of the string methods that are available and see how we can manipulate and work with strings throughout our applications. From there, we'll see how to do string conversions, and the issue here is that because we use text so much throughout our applications, we often need to convert non‑string types into strings. So we'll see how they do that explicitly, as well as when Java can do that for us implicitly. And then we'll finish up with a look at the StringBuilder class. And the StringBuilder class is an important part of building up strings piece by piece because the StringBuilder class can actually construct strings piece by piece much more efficiently than the string type itself.

# String Class

=>slides: Pg. 3

Now let's take a look now at the String class. Now the String class stores a sequence of characters, these characters are Unicode characters, and technically, the storing of what's known as UTF 16 format, and what that really means is that strings in Java can store pretty much any character in any language you're ever likely to have to work with. Now String classes have a concept of literals. Literals are specified in double quotes. So here I have a variable name of type String and I've set it to the string literal Jim. Notice we're using double quotes here. In our character types, we use single quotes, in strings, we use double quotes. Now string support concatenation. So we can use the plus operator to concatenate two strings together. So here, I've got a new variable greeting of type String and its value is the concatenation of the string literal Hello and the string variable name. So when I print this out, I'll get Hello Jim. Now we can also do concatenation with the += operator. So here, greeting += good to see you updates the value of greeting to now be the combination of its existing value, Hello Jim followed by space good to see you.

=>slides: Pg. 4

Now, there are some important things we need to understand about how string values are actually managed and stored. It turns out that our string variables do not directly hold their string value. Instead, they hold a reference to an instance of the string. So if I have this variable message of type String and I set her to the literal string I, again notice this is a string, not a char I for the String I, what happens here is that the memory containing that string I is allocated and has the value set. Our variable message doesn't directly hold that value, instead it holds what's known as a reference to the value. What that means is, although message doesn't have the value, it knows how to find the value. Now strings are what's known as immutable. What that means is the value of a string can never be directly changed instead, any changes to the value actually creates a new string instance. So if we add this line of code message += Love. Now conceptually, we're just adding space Love to the end of message's existing value. But what happens under the covers is the Java environment creates a brand new string that now has that new value and then our message variable is automatically updated to point to that new string. So Java took care of the details of allocating a new area of memory setting its value and updating our message variable. So if we then add another line message += Java, we can get a new value, I space Love space Java, and the message variable is again automatically updated to reference that new value. Now all this happens under the covers, so we don't have to do any special programming to deal with it, but it is an important aspect of efficiency we need to consider. Each modification to the strings value is actually allocating brand new areas of memory. Alright, so now in our next section, let's take a look at string equality.

# String Equality

=>slides: Pg. 5

So let's take a look now at how we compare strings to see if they're equal. So I have this code here that sets a variable s1 and assigns it a string value, and we know that based on the way strings are stored, we'll have a string instance that contains I love Java, and s1 will reference that string Instance. Then we'll go ahead and declare another variable s2, and s2 will also reference a string instance that has the value I love Java. Now if we want to compare these two strings for equality, if might make sense to simply say if s1 == s2. Because up until now that's how we always compared values for equality. That's how we compared two characters to see if they were equal, so it might make sense to do the string comparison that same way. But there's something important we need to understand about the equality operator when applying it to strings. The equality operator does not check the value of the string instance itself. The equality operator simply checks to see if both string variables reference the same string instance. So in this case, what happens is, it first looks it s1 and says, well, what string instance do you reference and then looks at s2 and says, what string instance do you reference? If they're both the same, it returns true. But in this case, they're not both the same. Even though their values are the same, they're two different string instances. So in this case, it would actually result in false. But of course, in most cases, we want to know if the value of the strings are the same. So that's where the equals method comes in. The equals method does a character‑by‑character comparison of the two strings

=>slides: Pg. 6

So we have some code here where we actually say if s1.equals(s2). What the equals method then does is look at the first character, the second character, the third character, and walks all the way through the strings. And as long as all the characters match up and there's two strings containing the exact same value, then in that case, it results in true. So what that tells us is that when we want to check two strings to see if they're equal, the overwhelming majority of the time, we want to use the equals method. We want to do that character‑by‑character comparison. =>slides: Pg. 6

=>slides: Pg. 7

Now, as you might expect, doing character‑by‑character comparisons of a string is certainly more expensive than checking to see if two references point to the same string instance, particularly if the strings you're checking are very long strings. So in cases where we need to frequently check strings for equality, we have this concept known as interning a string, and what string interning does is provide what's known as a canonicalized value. And that's simply a fancy way of saying that when you intern a string, a string of any given value will always return back a reference to the same string instance. And what that makes possible is that we can actually do the comparison using the == operator, which will then improve our application performance in cases where we frequently need to compare string values.

=>slides: Pg. 8

So starting out with those same two string variables we had before, s1 referencing one string instance that says I love Java and s2 referencing a different string instance that has that same value. And as we saw in this case, if we used the == operator, we got back false. But now, if we apply interning to those values, so if we say String s3 = s1.intern. What the intern method does is, it looks at the value of the string, and it looks around to see if there's already an interned version of that string. If it can't find one, it provides that interned version. So then s3 is set to reference the interned version of the string. So now if I say s4 = s2.intern, the intern method will look at the value of the string that s2 references, and it'll look around for an interned version of that string. And in this case, it would find one. It's the same version that s3 references. So s4 will now reference that same instance that s3 references So with that, if we then say if s3 == s4, in this case, that will now return true. Now, as you might expect, the intern method has its own overhead, so you don't want to just start interning all your strings. You only want to use the intern method if you know there's a scenario where you're frequently comparing the strings. So with that in mind, the overwhelming majority of the time you're going to want to do equality comparisons using the equals method. But just be aware the intern method is available as an option. Okay, so now in our next section, let's take a look at some of the other capabilities of the string class..

# String Methods and String Conversions

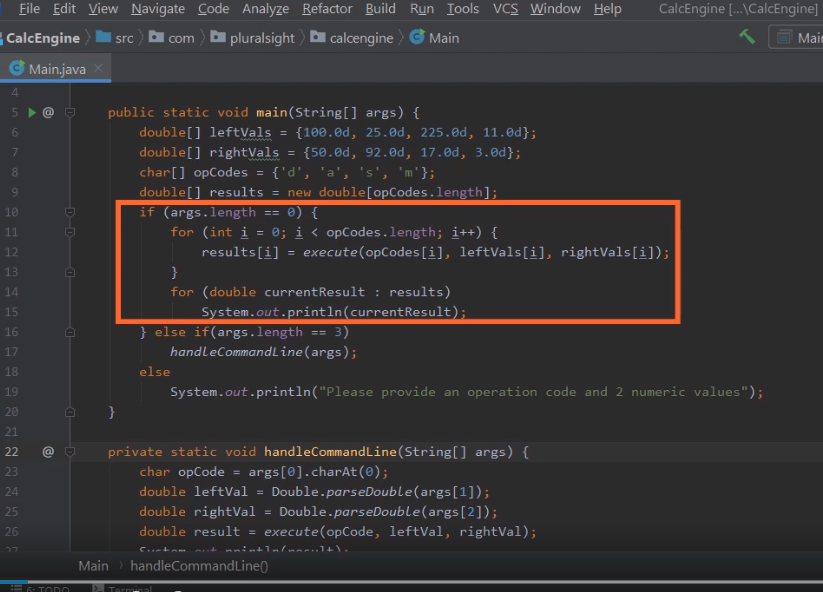
=>slides: Pg. 9

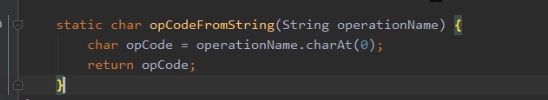
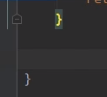
Let's take a closer look at some of the capabilities provided by our string type. And one of the key things that makes our string type different from the primitive types we've talked about earlier in this course is the string type is what's known as a class. We're going to talk about classes in detail a bit later. But one of the key value of classes as it relates to the string type is that classes provide methods. These methods allow us to operate on their data. So one simple method we have is the length method. So we can take any string, call its length method, and it will tell us how many characters are in that string. But we can do much more sophisticated things than that. For example, we have methods to help us create new strings from existing strings. So we can concatenate content onto a string, replace portions of a string, convert the case of a string, trim certain characters off the string, or even split a string into pieces. We can also extract a substring. So can extract one character from out of that string, or we can even take a substring from within that string. We can test a portion of the string. So we can ask a string if it contains a certain character sequence? Does it end or start with the sequence? You can find the location of a string sequence within that string. We can do comparisons. We've already talked about the equals method. There's also an equalsIgnoreCase. So it will check and see if they're equal, except for case. We can ask a string if it's empty, and we could even do greater than or less than comparisons. We can do rich formatting to create formatted strings. We're actually going to talk about that one in detail a little bit later. Then we also have the ability to create strings from non‑string types.

=>slides: Pg. 10

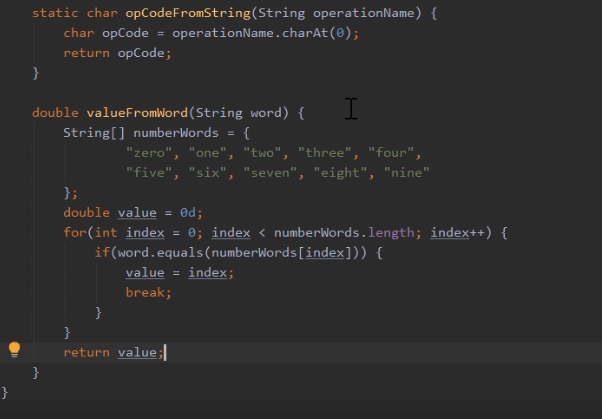
So let's take a look at this idea of converting non‑string types into strings because virtually all types can be converted into a string. And as we mentioned, that's where the String.valueOf method comes in. So if I started with an integer variable, iVal, if I simply try to assign iVal to a string variable, the compiler wouldn't allow that because remember that Java is a strongly typed language. But I can convert this integer into a string by using that String.valueOf method. So what'll happen here is that the valueOf method will look at the integer and create the string representation of that. So the integer 100 string representation are the characters 1‑0‑0. Now although there are situations where we do need to use valueOf convert a type into a string, in many cases, the Java environment can infer that we want to do the conversion and will do the string conversion for us implicitly. So let's look at another example. Say I have two integers here, i with a value of 2, and j with a value of 3. And then I multiply i \* j and assign it to another integer, result. But now let's say I want to produce the string that shows the operation I performed. So I want to produce a string that says 2 \* 3 = 6. Let's see how we would do that. I'll declare a String variable output. Now if I take that string variable and, as we mentioned, if I simply tried to assign i to it, the compiler wouldn't allow that because it can't assign an integer directly to a string. Now if I start adding some additional characters, like say I do i and +, the Java compiler is still not going to allow that because it would still be doing integer arithmetic. But if I now say i + the string space \* space, the Java compiler now goes oh, I know what you're doing. That's a string value, and you want to do concatenation. So the compiler will automatically convert the integer i into a string representation and concatenate that with that literal string space \* space. And we can do this with pretty much arbitrary sophistication. So if I do + j + the string space = space + result, each of those integer values will automatically be converted into their string representation, and then we'll concatenate those together to produce the string 2 \* 3 = 6. So now to help us get a better understanding of working with strings, in our next section, let's jump into our CalcEngine project, and we'll use strings to enhance our application's capabilities.

# Adding String Support to CalcEngine

Here we are back in STS, and what we want to do now is use our knowledge of strings to further enhance our application. Now as you recall, the application currently has two modes. This one mode will loop through a predefined set of arrays performing calculations. The other mode allows you to put in an opcode into numeric values, and the application will perform the operation indicated by those command line values. And so let's add a third mode that's more interactive. This will allow the user to start the application and then enter in an operation and the values. But let's make it a bit more interesting than that. Right now the user has to understand our opcodes. Let's go ahead and allow them to use full operation names. So, for example, if the user wants to do multiplication, they can simply say multiply, rather than have to specify the letter m, and let's them use words for some of the numbers. Now for simplicity, we'll just stick to the 10 basic digits 0 through 9. Let's scroll down a bit, and we'll start adding that code to our application. So now we're down here towards the bottom of our code, and there's a few steps we'll have to follow to do this work. You want to follow good programming best practice of breaking out each task into its own individual method. That will make our code much more maintainable. And some of the tasks we have to do is translate these words that the user's going to type in to values that our program already understands. So one of the things we'll need to do is translate the words like multiply into the opcodes that we already have support for.



So let's add a method named opCodeFromString whose return type is char, and our method will accept a single string parameter, and let's go ahead and call the parameter operationName. We'll go ahead and give our a method of body. Now remember, our codes are type char, so let's declare a char variable named opCode. Now at this point, we could go through and translate the words in a number of different ways. If we wanted to, we could do exact matches on the words that we expect, or we can do something simpler. So let's go with that simpler path. Let's just go ahead and grab the first letter from whatever operation word the user types in. So let's use operationName.charAt, and then we need to specify which character we want. Well, the character positions, just like array indexes, are zero‑based. So if we want the first character, we want the character at position 0. So that way if the user types in multiply, we'll get back the character m. If the user types in add, we'll get back the character a. So let's just go and return back opCode. So that takes care of translating the operation name into an opcode.

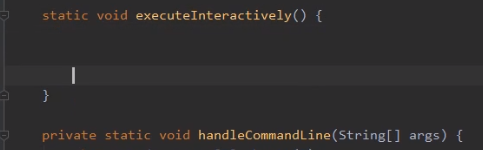


Now we need to do a translation work for the numeric words. So let's declare a method named valueFromWord. Its return type is double. And we'll give it a single string parameter named word. And again, what we want to do in this method is use the word that represents a number, translate that into the double that our application already understands, and return that back. So let's go ahead and give our method a body. Now when it comes to doing the word matching, remember that we're only matching the words for zero through nine. So a really easy way to set this up is use a string array that contains all the words we want to match on. So let's declare a string array named numberWords. Since we'll be providing a starting set of values, let's put opening and closing brackets. Then within the array, we can simply put the words in that we want. So that gives us the words for the 10 digits zero through nine. Now remember how arrays are indexed. Remember, they're zero‑based. So the index for the word zero will be 0. The index for the word one will be 1. So basically, we can just loop through this array, find the word we're trying to match on, and that index is its numeric equivalent. So to get started, the first thing we'll do is declare our local variable named value of type double because remember, our return type is double. Then we'll go ahead and use a for loop to count to the array. So now the for loop will allow us account through our numberWords array. So now we can use an if statement to check to see if the word the user typed in matches any of the values. Now remember that word is a string, so we want to use its equals method, and we want to know if the current element of numberWords matches it. And if it matches, there are a couple things we want to do. First, we want to take our index, and we'll assign that to our local variable named value. So now once we have the value, we're actually kind of done in this loop. So we really want to exit the loop. And there's a couple ways we could do that, but one way I like to use is use the break keyword. And as you recall, we talked about the break keyword when we were talking about the switch statement. Remember, the break keyword allowed us to process the statements in one case and then jump out of the switch when we hit the break keyword. Well, it works the same way in the loops. When we match the word we're looking for, we'll assign the index to the value, and when we hit the break keyword, we actually jump out of the for loop to whatever statement follows the for loop. So what we'll do to the for loop is return back our value local variable. So with that, our valueFromWord method will give us back the numeric value that corresponds to the word that the user typed in. So now that we can translate word names and our opcodes, we're ready to start gathering input from the user, and we'll see how to do that in our next section.

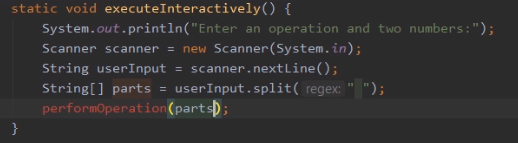
# Making CalcEngine Interactive



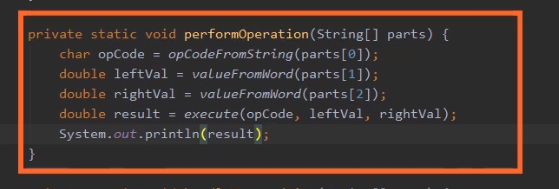
Here we are back in STS. We're ready to continue our work evaluating interactive capabilities to our application. Now before we carry on, remember that I mentioned that at this point, each time we declare a method, we want to be sure that that method is marked as static. You'll notice that our valueFromWord method is not yet marked as static. So let's go ahead and add in our static modifier. And now that we have the method marked as static, we can use it throughout the rest of our code. So with that done, we're ready to start doing the work of actually gathering the user's input. So I'll scroll up near the top. So we're here now near the top of our source code, just after our main method.



So let's add a new method that will handle the details of getting input from the user. Let's call that method executeInteractively, and we'll give it a return type of void. Now since this method will be interacting with the user, probably one of the first things we should do is print out a message that tells them what to do. So now once we do that, we want to get input from the user. Now when we've sent information out to the user, we've been using System.out. So it makes sense to get info from the user, we would use System.in. But working directly with System.in will require a fair bit of housekeeping for us. So what we're going to do is use another type that will take care of that housekeeping, and that's a type known as Scanner. Now you'll notice that when I type Scanner, STS says, Oh, wait a minute. I'm not quite sure what you're looking for here. So to help resolve this, it wants me to press Alt+Enter. And when I do that, it shows me a few different options for the type Scanner.

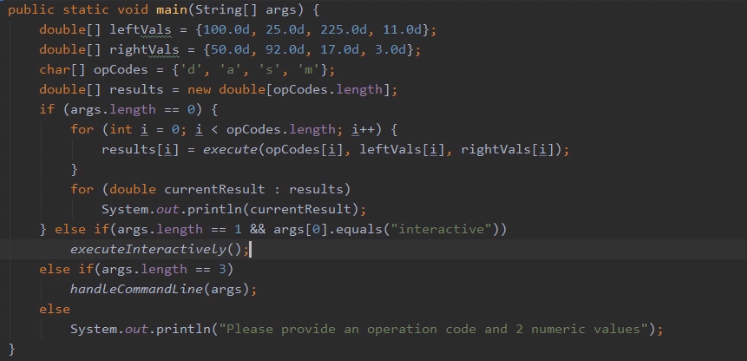


Now I do want this first one, this one marked java.util. So I'll go ahead and choose that. But you'll notice when I chose that, the error indicator for Scanner went away. Scanner's no longer red. But nothing else appears to have changed. Well, it turns out STS did actually make a change. It just made it towards the top of the source file. So let's scroll up to the top of the source file. So now up here at the top of our source file, you'll notice we have this import statement here. Well, this import statement is what STS just added for us. The issue relates to something we talked about early in the course. Remember that we mentioned that data type names are qualified by the packages they're contained in. So that's where the import statement comes in. By saying import java.util.Scanner, that tells the Java compiler that any time I use the data type Scanner, I actually mean the one Java.util.Scanner. So it gives me a shorthand for dealing with the type. All right, so let's scroll back down. We'll continue our work of getting input from the user. So now we're back here in our executeInteractively method. Let's go declare a scanner variable named Scanner. And I'll create a new instance of Scanner. And you'll notice here when I created the new instance of Scanner, I passed in System.in. So what will happen is this instance of Scanner will now take care of the details of getting the input from the user. All I have to do is ask the scanner instance for that input. And I'll do that by calling scanner.nextLine. So what scanner.nextLine will do is it will read all the input from the user until they hit the Enter key. And then it will give me that input as a string. So let's assign it to a local variable named userInput of type string. So what this will do if the user types in something like multiply three five, that value will be assigned to our userInput string variable. So once we have that input, the next thing we want to do is break it into its individual parts. So what we'll do is use the string method split. So we'll call userInput.split. Now the split method accepts an expression to identify what we want to use to split the string into parts. We simply want to split it based on spaces. So we'll pass in the string literal space. So what this will now do if the user types in multiply three five, split will return back three different pieces, multiply, three, and five. And it returns it back as a string array. So we'll assign the result of split to a local variable named parts of type string array. So in this case, if the user types multiply three five, multiply will be in parts[0], three will be in parts[1], and five will be in parts[2]. So now that we have those individual parts, we're ready to start actually performing the operation. And let's do that in a separate method that we'll create called performOperation.

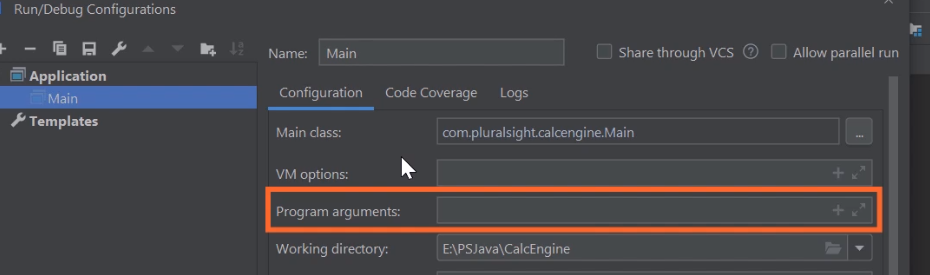


And we'll pass in our parts array as a parameter. Now we'll let STS create the method for us, so I'll press Alt+Enter. We'll choose Create method. And then we can tab through the return type and the parameter information. So now we have our string parts. We simply need to translate those into data types our application already understands. And we've already written methods to take care of those details. And we have our opCodeFromString and valueFromWord methods. So let's start out with the opCode. So let's declare a local variable named opCode of type char, and we'll call opCodeFromString passing in parts[0]. Then we'll declare a local variable named leftVal of type double. And then we'll call valueFromWord passing in parts[1]. And then we'll do the same thing for rightVal and parts[2]. So now we have everything we need to do the actual operation. So let's declare another local variable named result of type double. And we'll call our execute method passing in opCode, leftVal, and rightVal. And then once we have our result, we can print it out. So now with that, our executeInteractively method takes care of the details of getting input from the user and breaking that input into its individual parts. And our performOperation method converts those parts from strings into the appropriate data types, performs the operation, and prints the result out to the user. So we're just about ready to run our program, and we'll do that in our next section.

# Running CalcEngine Interactively



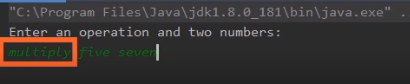
Here we are back in STS. We're just about ready to run our program with our new interactive capabilities. Now remember, we do all the work of running interactively in this method, executeInteractively. But of course, in order to do that work, we need to call that method. Now look at the way our program is currently written. Here in our main function, we have two distinct modes. So if the user provides no command‑line arguments, we work through a list of predefined arrays. But if the user provides exactly three arguments, then we use those three arguments to perform an operation. Let's go ahead and add in a third option here. Let's go ahead and add in an option where if the user provides one command‑line argument, and that command‑line argument is the word interactive, then in that case, we'll run interactively. So let's go here where we check to see if the args.length is 3. Let's move that down to the next line. And then just above that, we'll add an else if to see if the args.length is equal to 1. Now, if it is equal to 1, that means we have exactly 1 command‑line argument, so we want to check and see if that argument is the word interactive. So we'll use the conditional & operator, and then we'll check to see if arg sub 0 equals the string literal interactive. So if we know we have one command‑line argument, and that command‑line argument is the string interactive, then what do we want to do? We want to call our method executeInteractively. So now with that, we're ready to test out our program. Now, one way we could test the program is to follow the same procedure we used in the previous model. We built the project manual here in STS, went out, opened up a command prompt, and then manually launched a program, passing in the command arguments. We can imagine having to do that over and over again can get to be a lot of work. So STS can actually help us with testing this out. STS will actually allow us to specify command‑line arguments that will be passed into the program each time we run it. So to do that, we have to modify our project's configuration.



So we'll go up here to our Run menu, I'll click on that, so then I'll head down here to where it says Edit Configurations. You notice that when I do that, this dialog pops up, and one of the fields is program arguments. Any value I type in here will be passed into the program, just as if I'd passed them as command‑line arguments running the program from a command prompt. For example, if I wanted to test out our 3 arguments scenario, I could specify a 100 200. So if I clicked OK and ran the program, the program would receive three command‑line arguments But in our case, we're ready to test out this interactive feature.



So I'll specify the program arguments as the word interactive. So now my program will receive one command‑line argument, and it will be that string interactive. So I'll go ahead and click OK. Then I'll go up here and just run the program as we normally do from inside of STS.



Now you'll notice that when I do that, the Run window now has my prompt, Enter an operation and two numbers. So I'll head down here, click on that window, then I'll go ahead and type in the string that I want to use. So let's type in multiplying, space, the word five, space, the word seven. So now if the program works correctly, we'll translate that word multiply into the opCode: m, the word five into the number 5, the word seven into the number 7, so we'll multiply 5 times 7. OK, and hit Enter. And we get that proper result of 35. So our program has worked perfectly. We now have interactive support for allowing the user to type in the operation they want to perform, and they can use full words for the operation and for the numbers. So in our next section, let's take a look at something known as the string builder.

# StringBuilder

=>slides: Pg. 11

Let's take a look now at the StringBuilder class. The StringBuilder class provides a mutable string buffer. Remember when we first looked at strings, we mentioned that strings were immutable, and that meant that any modifications we make to a string don't actually change that string but create an entirely new string. And that was something we had to keep mind if we had to build up a string piece by piece because doing it that way wouldn't be particularly efficient. So what the StringBuilder class does is give us a way to efficiently construct string values piece by piece. So it has an append method that allows us to add content to the end of the string buffer, and it has an insert method that allows us to insert content within the string buffer. Now it's important to understand that the StringBuilder itself is not a string. Once you build up the content that you're interested in, you need to extract the string from the StringBuilder instance, and we do that by using StringBuilder's toString method. So let's take a look at some code.

=>slides: Pg. 12

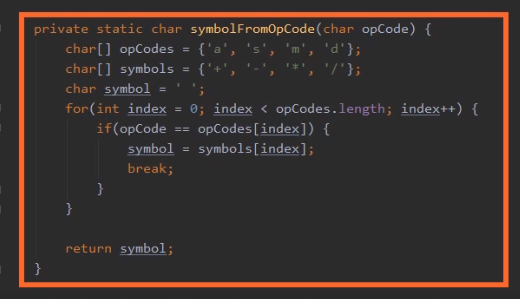
So we've got two variables here, a string variable location with the value of Florida, and an integer variable flightNumber with the value 175. And let's say I want to build up the string I flew to, and take whatever the value of the location is, on flightNumber, and take the value of our flightNumber variable. So I need to build this string up piece by piece. So that I can do that efficiently. I'll use a StringBuilder. So we create an instance of our StringBuilder class, so we have our variable StringBuilder sb = new StringBuilder. Now the StringBuilder class can dynamically resize itself based on the content you're working on. But if you have an idea about how long the StringBuilder content's going to be, you can provide that when you create the StringBuilder instance and that'll just help it to work a bit more efficiently. And once we've got our StringBuilder created, the StringBuilder will then have a buffer that it manages. And we can just start providing the content we want to go into that buffer. So I'll start out with sb.append and provide the literal string. I flew to, space, so that'll add that string to any content that we already have in our StringBuilder buffer, but since we just created the StringBuilder, there is no content, so that value will go right at the beginning of the buffer. We can then append the value of our location variable, so that appends the word Florida to the end of the buffer. We'll append the string literal on Flight #, so that goes, again, at the end of the buffer. We can then append our flightNumber variable, and that'll convert that integer into its string representation and add that to the end of the buffer. So now once we've built up the value that we're interested in, we need to get it out so we can use it as a string. So we have a string available here in message and we'll call sb.toString, so our message variable will have the value I flew to Florida on Flight #175. So that allows us to build up our string piece by piece in an efficient way. And just because we've gotten the string out of StringBuilder doesn't mean we have to stop using the StringBuilder. We can still continue operating on it. So let's say I want to add some time information.

=>slides: Pg. 13

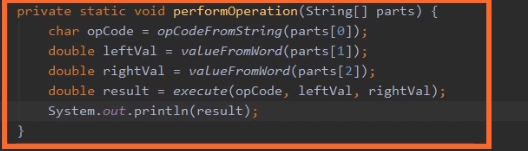
But this time information, rather than putting it at the end, I want to put it somewhere in the middle of the existing value. So one of the first things we need to do is identify where in that existing buffer I want the value to go. And let's say I want to put the time information where the " on" currently exists in the buffer. So by calling sb.index of will give me that position within the buffer. So once I have my insert position, I can call sb.insert, provide that position and the value that I want a place there. What StringBuilder will then do is move the content down that's already there and then add in that value that I provided. So I've got that string literal there, and now I want to place the time value just after that. So I do another sb.insert, indicating I want to be four characters further down, and then I have that time value, so our StringBuilder will again make room in the buffer and then place the time value right at that position. So once I do that, I can again call sp.toString, assign that to a variable, and now we'll get out the string I flew to Florida at 9:00 on Flight #175. So as you can see, the StringBuilder gives this efficient way to build up a string piece by piece. So to help us get a better understanding of all this, we'll jump back into our CalcEngine project and we'll use the StringBuilder class to improve our application output.

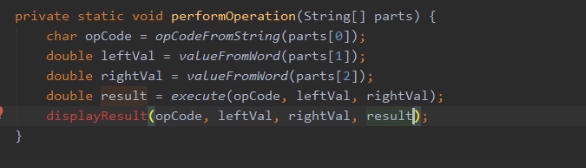
# Building String Output in CalcEngine

Take from materials.

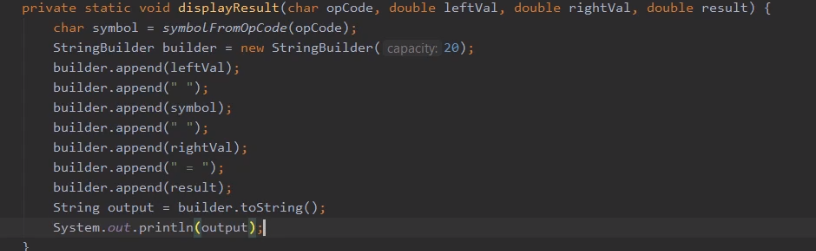


Here we are back in STS, and what we want to do now, use the StringBuilder class to improve the appearance of our application output. Because if you look here at this performOperation method that we created earlier in this module, we do all the work to perform the calculation, but then when it comes time to display the result, we simply print out a number. Well, it would be nice to give the user some context for the information that we're printing out. For example, if the user types in something like add three five, it would be nice to display back out to them 3.0 + 5.0 = 8.0, rather just putting the number 8. Now, in order to display a result like that, we're going to need to translate our opCode into the appropriate mathematical symbol. So I've provided this method here, symbolFromOpCode, that does exactly that. Notice that symbolFromOpCode accepts a char opCode as a parameter, and its return type is also char. And here inside the method, we have two arrays: opCodes, with a list of opCodes, and symbols, with a list of mathematical symbols. And these are parallel arrays. And remember, we talked about parallel arrays earlier in the course. When you have parallel arrays, the elements from one array correspond to the same elements in the other array. So, for example, the opCode a corresponds to the symbol +, and the opCode s corresponds to the symbol ‑. So what are symbolFromOpCode method does is it loops through the opCodes array to find the opCode that was passed in. When it finds the opCode that was passed in, it assigns it to a local variable symbol, breaks out of the loop, and then simply returns back that symbol. So with that method in place, we can go ahead and build our output.

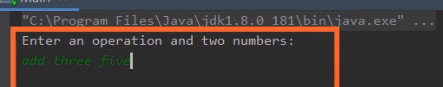




So here in our performOperation method, let's remove this line here where we simply print out the result, and instead we'll call a method we'll create named displayResult. So now in order to display the result, we'll need all the values involved in our operation. So we'll pass in the opCode, leftVal, rightVal, and result. So now once we have all the parameters specified, we can have STS create the method. So I could either click on that red light bulb or here on Windows I can simply press Alt+Enter. Then I'll choose Create method. Then I can simply tab through the return type and all the parameter information.



So now the first thing we'll do is translate the opCode into a symbol. So I'll declare a local variable named symbol of type char, and then I'll call our symbolFromOpCode method, passing in the opCode. So that'll give us the symbol that corresponds to the opCode. So now let's go ahead and create our StringBuilder instance. So I'll create a new instance of StringBuilder and assign it to a local variable named builder of type StringBuilder. So now once we have the StringBuilder, we can start building our content. So the first thing we'll do is append our leftVal, and then from there we want to go ahead and append the symbol as well. Well, let's go ahead and surround that symbol with spaces. Then the next thing we'll do is append the rightVal. Then we'll go ahead and append the string literal space = space, and then finally we'll append the result. And so with that, we've built our output. And notice as we're working with the StringBuilder, even though we're building a string, all the types we're working with do not have to be a string. For example, leftVal, rightVal, and result or all of type Double. Our symbol is of type char, but then all the literals we have provided are of type string. So StringBuilder class takes care of doing any conversions that are necessary for each of the types. Now, remember that StringBuilder is not itself a string. To get the string, we'll want to call its toString method. So we'll assign that result to a local variable named output of type string, and then once we have our output, we can simply print it out. And that's all there is to it. We now have the code in place that will display much more informative output to our user. So to test it out, we'll run the program. Now, just as a reminder, earlier in the module we specified the word interactive as a command line argument here in STS. So each time we run the program, our program will automatically run in our interactive mode. So I'll go ahead and launch the program.



So now the program is up and running, so I'll type in the input, add three five. And then once I do that I'll go ahead and press Enter, and as you can see, we now get the output, 3.0 + 5.0 = 8.0. So now our application works with interactive input, supports the use of words for the operations, as well as the numeric values, and displays much more informative output.

# 10.Summary

=>slides: Pg. 14

To wrap up, here's some of the key things you want to remember from this module. Remember that strings allow us to store a sequence of characters, and strings are stored in such a way that you can store pretty much any character sequence from any language you're ever likely to work with. Remember that string variables behave a bit differently than variables for primitive types we've talked about because string variables do not directly hold the string instance. Instead, they hold a reference to the string instance. So conceptually the string value exists in memory separate from the variable itself. Remember that strings are immutable, and what that means is that once an instance of a string is created, it can never be changed. So any time we make a change to a string value, even though in code it may appear that we're changing the value we're working on, we're actually creating a new instance of a string for each change that we make.

=>slides: Pg. 15

Then we looked at string equality, and as we mentioned, in the overwhelming majority of cases, when we want to compare two strings for equality we want to use the equals method because that assures that we're actually checking the content of each individual string. But then we talked about this concept of string interning. Remember that interning provides a canonicalized value, and that's a fancy way of saying that when you intern a string for any given string value, you'll get a reference to the same string instance. And by doing that, it allowed us to use the == operator to check to see if two strings are equal. So if we have two strings that we know have both been interned, we can compare them for equality by using that equality operator. And that improves the performance of frequently compared strings. But as I mentioned, you want to really be careful about using interning. In most interiors you're likely to deal with, the equals method is a better choice. Just know that interning is available for these certain high‑volume comparison scenarios. And then we finished up with StringBuilder.

=>slides: Pg. 16

Remember that StringBuilder provides a mutable string buffer, and by being mutable, in other words it supports changes, it provides a way for us to efficiently construct string values, so we can build up a string value piece by piece. But remember that the StringBuilder is not a string itself. In order to get the string that we've created, we have to call the StringBuilder's toString method, which will then return back a string that contains the value that we've constructed. All right that wraps up this module. In our next module, we'll take a look at Java's rich string formatting capabilities.

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