Transcript: Methods Declaration

>> Now let's consider a few more details about methods. A method declaration specifies the code that will be executed when the method is invoked or called. And when, when a method is invoked, the flow of control jumps to the method and executes this code. When complete, the flow returns to where the method was invoked, and continues on.

The invocation may or may not return a value, depending on how the method was defined. Let's take a look at method control flow. Note that the call method. If it's in the same class, only the method name is needed. Here we've got a, a class. And it's got a two methods, compute and myMethod, and compute is executing.

And so, it gets down to myMethod and then when myMethod is invoked, it actually jumps to myMethod. Executes that code, and then returns to where it was called and continues on. Now in the debugger, we would be stepping down to myMethod. And then to jump to the method to see the code execute, we would do a step in or click the Step-In button.

And that would go to the call method, and we could step through that. And when we get to the end of it, it would return to where it was invoked and continue on. Now, let's look at the method header. This is the first part of the method declaration.

And here is an example. Here we've got public care calc with three parameters. And the name is calc, and the return type is char. And the access modifier on this method is public. Recall, it could be public, private, or protected, or be, absent. But public is what we've got here.

And then we've got the parameter list. We've got three here, int num1, int num2, and String message. In the parameter list, it specifies notice the type and name of each parameter. And we call these formal parameters. Because this is where they're defined. When a method is invoked and we pass values, those will be the actual parameters.

But the ones you see here are formal parameters. Now let's look at the Method Body. The method header is followed by the body and it's just in a block. And in this case we're going to take in the, we have the parameters num1, num2, and message. Notice they're used in the body, we declare int sum = num1, num 2, and then we declare a char result.

And then we call on message, which is a string, we call the string method, charAt. On sum and we get the character at wherever sum is. And then we return that as the, as the return value. Notice that sum and result are local data. These are local variables.

They come into existence when you enter the block and they go away when you leave the block. And the same holds true for the parameters, num1, num2, and message. Are the arguments that they actually come and go, as the method is, when the method is invoked and, and then it ends.

The return expression, here we're returning result. It must be compatible with return type. And notice, the type of result is char. And, up here in the header we're, we're claiming that we're gonna return a care value. So, these are compatible of course. Now the return statement, the return type of course indicates the value that we're gonna return.

The return statement that specifies the value, actually can contain any expression that evaluates to the, that, compatible type that's gonna be returned. Now, if a method has a void return type, that is not returning a value, you can still have a return statement it just can't have the expressions, since it's not returning anything.

And, if you do have a return statement it's gonna transfer control, out the method before it reaches the end. And, sometimes this a very useful thing to do. So we have parameters and when we actually call a method the values of these actual parameters, also called args. In the invocation are copied into the formal parameters and then passed sort of passed in by value.

So here's an example of a method call, we've got char = obj.calc (4, count, and "War Eagle!"). Count must have some value and of course it must be an int. And so these are passed in, the values, or copied in, you might say. So num1 will be 4, num2 is gonna be whatever count was, and String message is gonna have the value War Eagle!

It'll actually be a pointer to that string object, War Eagle! And then the, the method gets executed, a, as you would expect there. So lets look at this example, over in jGRASP. Here I have queued up, here's MessageCalc. This is what we've been looking at. And then here's a method example, and these are in the examples folder there.

That actually calls message calc. And then notice this is in a class of message calc, and then the method is calc. It returns a char. And so since this is an instance method, as in, it's not static. We're gonna actually need to create a an object a MessageCalc object.

So we do that right here at the beginning and I'm gonna run this in the Canvas. I've already got a Canvas up here and we'll just sort of look at it as it goes. So, here's a message calc object being created. And then we set int to five.

A int count equals five, and then here's the call, we're gonna step in here. And when we step in, notice the arguments 4, 5, and War Eagle came in. As num 1, num 2, and message. And and we have this here, now we're gonna create a a local variable sum and add up num 1 and num 2 and I'll just step.

And then here's char result, or declaring an a new local variable result. And we're gonna call message.charAt (sum). In this case, sum is, 9, we added up the 4 and 5, got 9. And we can see that the charAt 9, is an exclamation point. So this is what we expect to get back here.

And sure enough we see result equals, exclamation point and the value 33 is its ASCII value. That's the underlying numeric value if you will. And so we step and we return and then char, our local ch here gets assigned to the exclamation point, which has this underlying value of 33.

And we can print those out. Now notice here, since, since we're talking about chars, this is a something you should be aware of. But a char has that underlying value 33, you can actually assign a char to an int. And so, here we're gonna make that same call.

I won't step in this time. But we'll see that i gets assigned the 33. And so, since that's a widening inversion from a care, to an int, 32 bit int, that's okay to do. Here's another example, just sort of in line here. I'm exsigning int j to the character m.

And we see that m, rather, yes, an m is a a 109. And then I can print that out. And there it is, and in this case, I'm gonna, take, 112. This is decimal 112 I'm gonna cast to a care, and assign it to a care. So this is a literal int, if you will, a 32 bit int, but I'm casting it to a care and assigning it there.

And we'll see what 112 is, 112 happens to be a lower case p, and we can print that out. So just sort of making the point here that the type care, is, is actually a numeric type underneath, and you can assign it to an int. And if you sign an int to a care, you do have to cast, like we did here in line 30.

And we cast this N here to a care, and and we're good to go there.