Programming Methodology-Lecture07

Instructor (Mehran Sahami): Alrighty. Welcome back to yet another fun-filled exciting day of CS106a. So a couple quick announcements before we start – first announcement, there are two handouts in the back. Hopefully, you already got them. If you didn't already get them, you can get them later.

Second announcement, just a little reminder, microphones, if you go ahead and pick up those microphones right now and keep them in your lap. They're warm. They're cozy. They're fun. Use them to ask questions. Just because, looking at the videos, got a, kind of, general reminder that the questions aren't actually coming through. Whoa, or I can just breath real heavily into my microphone.

All right. So one real quick point before we dive into the main meat of the lecture, it's just a clarification on something we did last time called The Cast. So remember last time where we had a cast, which was this thing that allowed us to say treat this one data point, or this one data item, this one variable as a different type for one particular operation.

And so last time I told you you could actually do something like int X equals – let's say we had some double up here, like YY, and maybe Y got some value like 3.5, and I said you could do a cast like this, and we put Y here, right? And so that'll take Y. It'll truncate it, right? It drops everything out of the decimal point and assigns it to an int.

And I, sort of, said, misspeaking, actually, that you could drop this part. It turns out that in Java, any time you do a cast of something, that loses information. Like, when you go from a double to an integer because it truncates, it drops everything after the decimal point, you have to put in the cast, so you have to make this explicit.

If you go the other way around where you're not losing information, like, if you were to have Y equals X at some point where you know that X is an int and Y is a double, you're not gonna lose information by assigning an int to a double. So there you're okay.

You can do the explicit cast if you want, and actually I would encourage you, actually, to use the explicit cast, but in this case, you don't need it. In this case over here you actually do. So when you lose information, you need to make an explicit cast. For most of you, it probably won't affect your day-to-day life, but that's just something you should know. Alrighty. So any questions about anything we've done up to now?

We're just gonna, kind of, press ahead with something that we've talked about last time. Remember last time we're talking about our friend the while loop? And the while loop was something that you used when you don't know how many times you're gonna do something. This is what we refer to as an indefinite loop – not an infinite loop, right? It's important that you have "de" in there. Infinite loop keeps going forever, but an indefinite loop is one that when it starts out, you don't know how many times you're gonna do something. You're just gonna do it some number of times.

Well, it turns out there's times in life where you're gonna do something some number of times. You don't know how many, but you figure you probably want to do it at least once, okay? This was, kind of, the case of my brother going to college. Like, he figured, "I'm gonna go to college." But when he got there, he didn't know how many degrees he was gonna get, and he just keep going and going, and five degrees later, he was, kind of, like, "Okay. I think I'm done now." So that was his while loop, right? He didn't know how many times, but he knew he wanted to do it at least once.

Now, the interesting thing is that in programming, this actually happens fairly commonly. So I'll show you a quick example of this. If we go over to the computer, there's something we call the loop and a half because in some sense, you want to half of the loop at least once, okay? Think about it this way. Let's say we want to take some program that reads in a bunch of numbers from the user until the user enters zero to stop, and it, kind of, computes the sum of all those numbers, so it adds them all together.

So we might have, basically, some sentinel value, right? We're gonna use our constant, our little friend last time that we talked about, the constant, to define what is that value that the user enters in which point they want to stop entering values. This, is in programming speak, is often referred to as a sentinel. It's, kind of, the last thing I'm gonna give you that tells you I'm done; I'm not gonna enter any more values. But until I give you that sentinel, I'm gonna give you a bunch of values that I want you to add together, for example.

So that sentinel we might define as a constant, and so we could have some loop that says well, if I'm gonna keep track of some total or some sum of all the values you enter, that home's gonna start at zero. I'm gonna ask you for some value, so I'm gonna read in some integer and store it in the little box named Val, right? So I just declare it a variable, and then I'm gonna have a loop, and what my loop says is if the value you gave me is not the sentinel, that means you don't want to stop yet, right?

So I have a logical condition in here that's checking against the constant, and that's a perfectly fine thing to do. So we're taking a lot of those concepts you saw last time and just, kind of, sticking them all together into one bigger program. If I haven't seen that sentinel yet, then take the value you just gave me, and add it to my total, and store it back in the total using the little plus equals shorthand you saw last time, okay?

Then what I need to do is after I've done this, I need to ask you for a value again. So I say value equals read it. Notice I already declared the value up here, right? I already declared this variable up here, and this guy's lifetime is until it gets to the closing brace at the level at which it's declared. It's not declared inside the while loop, so this closing brace doesn't make it go away. This closing brace makes it go away, which means it's actually alive the whole time in the while loop, and it's perfectly fine.

So val, I read in some other value from the user, and, again, I go back through this loop. Did they give me zero? No, add it in. Did they give me, zero? No, add it in. And when they give me zero, then I say, oh, the value is equal to the sentinel, so this test becomes

false, and I come down here, and I write out the total as total, and you might say, oh, okay, Mehran, that's fine. What's wrong with that?

Well, in computer science or actually in computer programming, we really hate duplicated code. If we can avoid duplicating code, even if it's one line, we generally try to do it, and you can see there's a duplication here, right? What's happening is there's something we want it to do at least once, which was to ask the user for some value, but we need to keep doing that over and over in the loop, even after that first time. So we, kind of, run into this conundrum where we say are we gonna have to repeat code?

And there's a way around this. So the way around this is we pop out that piece of code, and we're gonna pop in our friend the loop and a half, okay? And so what the loop and a half says, the first thing that looks funky about it is we say while true, and you see that and you go, "Oh, my God." Right? Any time you see 'while true' you're thinking bad times, right? "I'm never gonna break out of the loop. Every time I come here and evaluate the condition, it's always true. Isn't this an infinite loop, Mehran?"

And it would be except for one caveat, and here's the caveat. We're gonna ask the user for a value, right? Notice we declared the int val inside here. We could've declared it outside if we wanted to, as long as we didn't do the read into outside. We could've just said int val and declared it out here, but we're only gonna use it inside the loop, so we're just gonna declare it here.

We read an integer from the user. We ask if the value is the sentinel. If it is, we break. What a break statement does is it will break you out of your closest encompassing loop. What does that mean? It means it finds whatever loop is the loop you're currently in and jumps you out of just that loop. So if you hit a break statement, it will jump out to, essentially, the place right after the closing brace for that loop and keep executing.

So what it allows you to do, essentially, is to check the condition to break out of the loop in the middle of a loop, which is, kind of, a funky thing, rather than at the very beginning or every time you iterate through. So we get the value from the user. If the value is the sentinel, we break, which means we never execute this line or the rest of the loop. We jump down here. If it is not the sentinel, then we add it to the total. While is true, so we execute the loop again and go and read another value from the user, right?

So notice that this read int line is no longer repeated. We only do it once, but because of the structure of this, we're always guaranteed that this portion of the loop up here is always executed at least once because we had a while true, and we're checking the condition to break out of the loop in the middle, okay? Now, one caveat with that is in terms of programming style, it's generally a bad time to have multiple of these kind of breaks inside a loop because it makes it very difficult for a programmer to figure out what condition has to be true to break out of the loop.

If there's only place you see a break, then there's only one condition you need to check to break out of the loop. If you have multiple statements, the multiple if, you know, blah-

blah-blah break in your loop, that means the programmer has to keep track of multiple possible places you could've broken out of the loop in their head, and that gets pretty dicey. So you generally want to avoid that when you're doing it, okay? So any questions about the loop and a half? Uh huh.

Student: Is it okay to redeclare a variable in a loop like that over and over again?

Instructor (Mehran Sahami): Yeah, it's fine to actually declare that variable inside the loop. You don't need to worry about, like, efficiency issues or anything like that.

So let me show this to you in action in a larger program, right? So here's that loop and a half in the context of an actual run method. It prints something on the screen. It says total to be equal to zero, and then it comes here, while true will always execute, right, because the condition's true.

So it asks the user for some value. We enter one; one is not equal to the sentinel. So we add that to the total, and you can see here it's keep track of value and total. Here's just my two declared variables and the little boxes for them, and they get updated as I go along.

So while it's still true, I read another value. Let's say the user gives me two. I add it to the total, which is now three, and I come back up here. I'll go through this a couple more times, okay? User gives me zero.

Now, notice when the user gave me zero here, I hit the break because my value is zero, and that's equal to the sentinel. So it says, hey, the if part is true, so do the break, and it immediately jumps out to the end of a loop. It did not do – even though it [inaudible].

So I feel a little like Roger Daltrey when I'm doing this. Anyone know Roger Daltrey, lead singer of The Who? Go look it up on Wikipedia, and watch the video. Man, I'm feeling old. All right. And then it says the total is, so this total plus equals value. The value just doesn't get added that last time, even though it would've been a zero and wouldn't affect the things.

It would be different, right, if we said the sentinel equaled a negative one. Then you would actually see that the total didn't get one subtracted from it, and we could've set the sentinel to whatever we wanted, but in this case, we used zero, okay? So that's our little funkiness, and it writes out the total of zero for actually doing the loop and a half.

Now, in terms of doing all this stuff, okay, you might think, "Well, hey, Mehran, you showed me about the for loop last time. You showed me about the while loop. It turns out I can write the same loop in equivalent ways using both of them." So you remember the for has the initialization, the test, and the step initialization happens once. Then the test happens every time through the loop, and then the step happens, kind of, every time at the end of the loop?

That would be exactly equivalent if I wrote it like this. The int happens once before the loop. I have some while loop that checks the same test. I have the same set of statements that would be in the for loop body, and I do the equivalent of the step in the for loop at the very end of the loop. So these two forms are exactly equivalent to each other, and you might say, "Hey, Mehran, if they're exactly equivalent to each other, why do I have these two loops, and when do I use one versus the other?"

Well, for loops, you want to think about what we refer to as Definite Iteration. That means we generally know how many times we want to do something, and that number of times we want to do something is how we, sort of, count in our test. When we don't know how many times we actually want to do something, that's an indefinite loop, or indefinite iteration as I just talked about, that's when we use a while loop, when we generally don't know how many times we're gonna do something, okay?

So that, kind of, why we give you both forms, and most programming languages actually have both forms, but to a programmer, it's more clear between seeing the difference between for and a while, it's, kind of, the meaning of it, right? Are you counting up to something some number of times, or are you just gonna do something until some condition is true? That's, kind of, the differentiator there. So any questions about loops?

So let's put this all together in the context of a bigger program, okay? So I will show you a bigger program here. So we'll come over here. Here's a little program; let me just run it for you real quickly. Come on. So we run along, and what this program's gonna do is draw a checkerboard, right? Just like you're used to in the game of checkers or chess, a little checkerboard on the screen.

So this is gonna be a graphics program. It's gonna draw a bunch of G rects, which are just rectangles – in this case, they're gonna be squares, to draw a little checkerboard, and you should think, "Huh, this might have some similarities to, maybe, some of the stuff you were doing in your current assignment. Yeah, so let's go through this code and see what's actually going on, and we can put a bunch of things we learned together like constants, and loops, and blah-blah-blah.

So what's going on over here is, first of all, we start off with a couple constants, right? And the constants we're gonna have tell us how many rows and columns are gonna be in our checkerboard. We want to draw a standard checkerboard, which is an 8x8 checkerboard. So we have private static final int, end rows is eight, and private static final int, end columns is also eight.

Notice that these two constants are not defined inside a method. They're defined inside the class but not inside a particular method, which is generally where your constants are gonna be defined, right, in a class but not inside a particular method, and then for the run part of the program, first thing we need to know is we need to figure out – because we want this thing to be general, right? No matter how big our window is, we want to draw a nice little checkerboard.

So we need to figure out how big do those squares on the checkerboard need to actually be so they appropriately fill up the screen. How do we do that? We get the height of the graphics window – remember our little friend, the method Get Height, which tells us how many pixels high the graphics window is, and we divide that by the number of rows.

Since we divide that by the number of rows, this is gonna give us integer division, right? This is gonna give us back an integer, and we're gonna assign that to square size, which is an integer. So everything's perfectly fine because this is some number of pixels. This is an integer, and so this is an integer value divided by an integer value gives you an integer division.

So that tells you basically how many squares can you fit in, sort of, the height of the screen, and that's gonna be the size of one of the sides of our squares, and, as you know, squares have the same size on both sides, same length on both sides, so we're perfectly fine.

Now, what we're gonna do here is, right, we want to get the structure that's like a grid. In order to get a grid, we're gonna have what we refer to as a pair of Nested Loops. All that means is one loop is inside the other loop, okay? So we have two for loops here, and you can see with the indenting, one is, kind of, nested inside the other one.

So one is gonna count through the number of rows. So we're gonna have one loop that's gonna go through the number of rows we want to display and another loop that, within each row, is going to, essentially, count the number of columns in that row because we're gonna put one box for every little square grid on there, which means for every row, we need to do something for every column in that row.

So we're gonna have one for loop that's going to have I as what we refer to as its index variable. So remember when we talked about for loops, and we talked about counting, we said, "Oh, yeah, you say something like int I equals zero I less the num rows." And that'll count it from zero up to num rows minus one, which that iterates num rows times or N rows times.

That's great. The only problem is this variable I, remember, is alive until it gets to the closing brace that matches this brace, which means that I is alive all the way to down here. Well, if that I is alive all the way to down here, if we have some other for loop in the middle, if it's also using I, those I's are gonna clash with each other, and that's bad times.

So what do we do? We just use a different variable name. What we're gonna do is what's the next letter after I? J, right, so I, J, K you'll see are very popular names for loops, and those are the one place where descriptive names – you don't need to have some big name like, "Oh, this is the Counter Through Rows variable," or something like that. I and J are perfectly fine because most programmers are used to I, J, K as the names for control variables, or index variables, and loops is how we refer to it because this is what the loop is using to count through, so we think of I as the loop index.

So we have one here for J, and notice all the places I would've had I are now just replaced by J. So J gets initialized to zero. We count up to N columns, and J plus plus is how we increment J. So a common error with nested loops is you have the same variable up here as you had down here, and that's generally bad times, okay?

Now, once we're counting through all the columns, the thing we need to figure out is where is this square that we're gonna draw gonna go? What's its XY location? Well, it's XY location says, well, first of all, if you want to figure out its X, that's, sort of, going horizontally across the screen, right? Going horizontally across the screen depends on which column you're currently in.

So I take J, which is the index variable for this loop over the number of columns here, and I multiply it by my square size. What I'm essentially saying is move over in the X direction J many squares. That will tell you where the X location is for the next square of the tracks you're gonna write out.

Similarly, Y, which is the vertical direction, I need to figure out which row I'm in. Well, I is what's indexing my rows, right? So I'm just gonna take I and multiply it by the square size. That tells me how far down the screen I need to go to get the Y location for this square. So now I have the X and Y location. Any questions about how I computed the X and Y location? Don't be shy.

All right. Are you feeling good about X and Y? If you are, nod your head. All right. I'm seeing some nods in there, some blank stares. If you have no idea what I'm talking about, nod your head. All right. That's better. There's always this indefinite ground where it's, sort of, like, no one nods, and they don't give you any feedback. So I'm begging you for feedback, all right?

Student:[Off mic].

Instructor (Mehran Sahami): We could've put this outside of the J loop and just computed – or we could've put the Y location outside of here and just computed it once because that Y location will remain the same for the whole row. So that would've been an interesting little optimization we could've made, but we didn't make it here because I just wanted to compute X and Y together so you would see them computed together, and because our forms are pretty similar to each other, okay?

So once I know the X and Y location for that square, I need to say, hey, get me a new square at that location. So I say new G rec. That gets me one of these new square objects. Where is the upper left-hand corner for that square? It's at the XY location I just computed. How big is the square? Square size width and square size height, right, because it's a square; it's got the same width and height. So that gives me a new square.

Now, I gotta square at some location that I want to display that square out, right, because I computed the right X and Y. The only problem is at the checkerboard. Some of the squares are filled; some are not filled. How do I figure out which squares should be

filled? This is where I do a little bit of math and a little bit of Boolean logic, and it just makes your life, kind of, beautiful.

So rather than having a whole bunch of ifs in there and trying to figure out, oh, am I on an odd square, an even square, which row am I in, blah-blah-blah? You do a little bit of math, and you figure out your math, and you say, hey, you know what, that very first square that I draw in the upper-right-hand corner, okay, let me see if I can bring that little square back up again. So I'll run this one more time.

This guy up here, I can think of that square all the way up there as the 00 square. That guy's not filled, but if I were to move one over that is filled, or if I were to move one down that is filled – so if I think about the index location that I'm currently at, and I take my X and Y coordinates, or I should say my I and J variables because that's what indexing my rows, and my columns and add them together, if that puppy's even, then it's not a filled square.

If it's odd, then it is a filled square because the very first one is 00. That one's not filled, so even's not filled, but if I move one in either direction, I get something that should be filled. If I move two in either direction, I get something that should not be filled, et cetera.

So the way that translates into logic for the code is I come over here, and it looks a little hairier than it actually is, but I say take the I and J, and add them together, and mod it by – or I should say – remainder it by two, right? If I remainder by two, if that remainder is zero, that means it's even.

If the remainder is one, that means it has to be odd.

The remainder can't be anything other than zero or one when I'm taking the remainder and dividing by two. So if that remainder is not equal to zero, that means it's an odd square, right, because my remainder divided by two is not equal to zero; it means it was equal to one. What I'm gonna do in that case is I'm gonna set filled.

So this looks very funky. You might be inclined to think, hey, shouldn't you have an if statement here, and say if this thing is true, then set filled? Well, think about, right, if this thing is true, set filled will become true, and I'll get a filled square. If this thing is false, set filled will become false, and I'll get an unfilled square.

So it works for me the way I want it to work in both cases, so rather than worrying about all this controlled logic for an if statement or all this other stuff, I just take my condition here, which is this big honking Boolean expression that evaluates the true or false and just stick it in for the value for – that set filled is actually expected, okay?

And this will give me the alternating effect for squares, and when I finally do this, I need to say, hey, you know, I got that square. It's filled the way I want it to be filled. I know its location. Throw it up on the canvas for me, so I add it. And then I just keep doing this

over and over, and I get all of my – for every row, I'll get a line of squares across and all the columns. Then I'll come down to the next row, and I'll just keep doing this until I get eight rows of squares. Uh huh?

Student:[Off mic].

Instructor (Mehran Sahami):Use a mic, please. That'd be great. Thanks.

Student: How do you designate is as a Boolean expression? Does it just automatically assume that is going to be true or false, or – Instructor:

Right. So that's a good question. The key that's going on here is this little operator right here, the not equal operator, right? The not equal operator is a logical operator, which means it's gonna have two arguments that it works on, and what it's gonna give you back is true or false.

So the way it knows that it's Boolean is that this particular operator, the not equal operator, always gives you back a Boolean value, right? Just like greater than or less than, all those operators are Boolean operators; they give you back a Boolean value. Whereas, the stuff I'm doing over here, right, is just math. This is giving me back an integer, and I'm comparing that integer with another integer using a Boolean operation. That's why I get a Boolean. Uh huh?

Student:[Off mic].

Instructor (Mehran Sahami):Does this only – well, this will work in any case as long as the window is wider than it is tall because the way we compute the squares is based on the height as opposed to the width. Uh huh?

Student:[Off mic].

Instructor (Mehran Sahami):No, that's different, and we'll get into that now. Well, the private part's the same, but I'll show you what all of those things mean in just a second. So we're gonna do methods in just a second, and then it'll become clear. Uh huh?

Student:[Off mic].

Instructor (Mehran Sahami):Because everything's in terms of pixels, and the pixels are always integers. So we just want to compute in terms of integers because all of our values mean our integer values. So let me push on just a little bit. Uh huh, question right there?

Student: Why are we doing it this way if we're supposed to be doing top-down programming; is it the opposite?

Instructor (Mehran Sahami):Well, in this case, top-down programming is how you break things up into smaller functions, right? Here we only have one function. So we

could actually think about, "Oh, should I do one row at a time?" And that would actually be an interesting decomposition is to have a function or some method that does one row for you, and I iterate that some number of times, and the reason why we didn't do it that way is we haven't done methods in Java yet, which is the next topic we're gonna get into. So thank you for the segue.

So the next topic we're actually gonna deal with is our friend the method, but in the Java world. So hopefully you've already seen in Carroll, right? You saw how to break things down in Carroll, and we did decomposition in Carroll, and now it's time to bring that into the Java world, okay?

So when we think about methods in Java – you already saw a couple of these things, right? Like, read ints, for example, is some method that you've seen, and so we might say, you know, int X equals read int, or print lin is some other method you've seen, right, that just prints out a line.

Now, one way we can take these notions and make them a little bit more familiar is, first of all, we can say the idea of a method is just like in Carroll. You want to use methods to be able to break down programs into smaller pieces that you can reuse; that's critical. But, in Java, they get a little bit more complicated, and here's where the complexity comes in, and it's easiest to draw the notion of how they differ in the Java world by thinking about mathematical functions, right?

So somewhere, someday, which you're never gonna have to worry about again as long this class is concerned, is you learned about this thing called the sign function. You're like, "Oh, Mehran, you told me there was gonna be, like, no, like, calculus in here." Yeah, don't worry. This is just for this example, and then the sign function goes away. You never have to worry about it again, at least for this class, right?

But the way the sign function worked is it took some value that you computed the sign of, right? So there was something here like X that you took the sign of, and what you got back from the sign function was some value that was essentially the sign of X. So you not only had something that we called a parameter that was what that function was expecting in terms of information going in, you also got some information coming out, which is what the value of that function returned to you or gave back to you, okay?

That's the one bit of additional complexity that comes up with methods in the Java world as opposed to the Carroll world, and the Carroll world, you didn't have any parameters that went in, and no values came out. In Java's world, you do, okay?

So, first of all, just to wrap up on the little math example, a couple people asked last time about some mathy functions, and it turns out that there's this thing you import called Java.lang.math. They're a bunch of mathematical functions you get in the Java world for free, and some of those are, for example, oh, a function you might be using on this next assignment like square root.

So if we have double, let's say Y – or I'll say XX, XX equals 9.5, and then I might have some other YY here, and I want Y to be the square root of 9.5. I would say math.sqrt, which is the name of the method for square root, and then I give it the value, or I give it a variable whose value I want to compute the square root of.

So this guy takes in something of type double, computes the square root, and gives you back that value that you can now just assign to some other double, for example, okay?

And there are some other ones in there. There's a power function. Someone asked about power last time. That's where you also get the function; it's also in there, but power gives you a nice example of you can have multiple parameters. So power you actually give it an X and a Y, and what it does is computes X to the Y power. So it's essentially computing something that would look like that if we were to write it out in math-ese, and it gives you back that as a double. So we could assign that somewhere else.

But notice we have two parameters here, and they're separated by a comma, okay? And that's, generally, what we're gonna use to separate parameters is commas, just like you saw when we created, for example, objects over here, right? We created a new G rect, and we had to give it four parameters. They were just separated by commas, same kind of idea going on over here.

Now, the question comes up, what's the whole point of having these kind of functions exist in the Java world or methods in general? And the critical part about methods in Java, there's the whole notion of decomposition and top-down design. That's part of it. That's not the most critical part.

The most critical part has to do with a notion we think of as information hiding, okay? So what is information hiding all about? The real idea – the way you can think about this is that everything in the world is a function or some method. So, for example, anyone know what this is?

Student: CD player.

Instructor (Mehran Sahami): CD player, you're like, "Mehran, a CD player, come on. You've gotta be kidding. Like, what, were you, like, on some 1990s archeological dig or something?" It's like, oh, okay. I think I've found the CD player. This is a CD player, right? It's also a method. What does that mean that it's a method?

Well, guess what? It takes something in, and the thing it takes in happens to be CDs, and we have a little CD, Alice in Chains, always a good call, Apocalyptica. Any Apocalyptica fans in here? Yeah, a few, it's Metallica done on the cello, and Tracy Chapman just to date myself.

But what do you do? You take a CD. You put inside here. You hit close, and you hit play, and out comes music, and the music that comes out of this thing depends on which CD you put in, but the interesting thing about it, if you think about it, is all of the electronics

in here are completely reusable, right? I can use this CD player on, virtually, any CD, right?

Someday we might get to the day where you go to the store, and you buy a CD, and all the electronics to actually play the CD are inside the box, and you just plug your headphones into this, and you just listen to your CDs, and then we toss away those electronics when we get rid of that CD, and we get a copy of all the electronics again. That would be a huge waste, right, if you kind of think about landfill space and all that stuff.

So why don't they do that? Why don't all the electronics to listen to a CD come with the CD? Because I can create them once and generalize them in a way so that if I pass in the right parameter, which is the CD, I can use the same thing over and over for all different kinds of values that go in, and, as a result, produce all kinds of values coming out, okay?

So that's the thing you want to think about in terms of methods is you want to think about methods and defining what's gonna go into them and what's gonna come out of them in a way that's general enough that they can be used over and over again, and that's what the whole beauty of software engineering about is thinking about that generality, okay?

So with that said, how do we actually define a method, okay? So we need a little bit of terminology to think about how we define methods. Yeah, it's time for that dreaded terminology again in the object-oriented world, okay? So when we actually call a method, right, you've seen this a little bit, just like we did over there. We have what we refer to as the receiver, the name of the method, and then some arguments, which are the parameters, right?

So here the receiver happens to be math. The method name is power, and then there's some arguments X and Y that get passed in, okay? The way we think about this thing is we say that we are calling or invoking a particular method; here is the name of that method. So a lot of times you'll hear me say, "Call the method." That's what we're referring to is invoking that method.

What we passed in terms of these arguments is the parameters, what that function is actually or that method is actually expecting, and you'll hear me sometimes use the terms function and method interchangeably, and they basically mean the same thing. And this guy can, potentially, give me back some value, just like square root and power did over there.

Now, sometimes we also refer to this, and you may have heard me say this in the Carroll world, right, as sending a message to an object, right? So if we have our friend the G label, and I declare, let's say, I'll just call this lab to make it short, which is short for label, and I ask for a new G label, and that G label, I'm just gonna give it the words high at location 10, 10 just to keep it short and fit it on the board, and then I set its color by saying label – or in this case just lab.set color is color.red, okay?

So when I've done that, lab is the receiver. The method name is called Set Color, and, alternatively, I would say that I'm sending the set color message to lab, okay? That's just the terminology that we use. You should just be aware of it. So set color is the message that we're sending, okay? That's the receiver of the message, same kind of thing as the name of the method and the receiver of that method, or making a method called or a method invocation; those names just get used interchangeably.

So how do I actually create a method in Java, okay? Let me show you the syntax for creating a method. It's a little bit different, just slightly different than Carroll, but actually very similar, and so the way this looks is I first specify the visibility. You're like, "Huh? What is that?" You'll see in just a second.

So we have the visibility, the type that the method may potentially return, what value it may return, the name of the method, all these have squiggly lines under them because these are things that you will fill in momentarily, and some parameters that get sent to that method, and inside here is the body, which is just a set of statements that would get executed as part of that method.

Now, what do all these things mean? First of all, let's start with the visibility, which I'll just abbreviate as vis. The visibility, at least as far as you need to know right now, is either private or public, and you're like, "Hey, yeah, I remember those from the Carroll world."

Yeah, and they're exactly the same. The way you want to think about it is right now all you need to worry about is your run method is public, and, pretty much, all the other methods you write, as long as that method is only used in a single class, which is generally the class that you are writing that method in, they will be private. So, for all intensive purposes right now, all the methods that you're gonna write, except for run, are all gonna be private, and run has to be public. Uh huh?

Student:[Off mic].

Instructor (Mehran Sahami): The way things are set up, you just need to make run public because it needs to have a larger visibility the way we've, kind of, defined the CS106 stuff right now. At the very end of class, like, when we get to the end of the quarter, I'll, sort of, lift the covers on everything, and you'll see why, okay?

So that's the visibility, private or public. Run is public, most other things are private. All private means in terms of visibility is the only people who get to see this method are other methods inside the class that you're defining, okay? So that's why it's visibility. It's who else can see it, okay?

The other thing you want to think about is the type. What's the type? The type here is specifying what gets returned by this method. So remember we talked about some methods, for example, that give you back something, okay? It is the type of the return value. What does that mean?

If I have some function that's going to compute the sign of something, and you give me a double, and it's gonna give you back a double, it's return type is double, and you might say, "Hey, but, Mehran, when I was doing methods with Carroll, none of my Carroll methods actually returned any values." Right? Yeah, that was perfectly fine, and guess what word you used when they didn't return a value?

Student: Void.

Instructor (Mehran Sahami): Void, yeah, that's a social. Good times all around, right? So void is a special type, which basically just means I'm not gonna return anything, right? So its return type is void. It's like you're sinking into the void. You're getting nothing from me. You know, it's, kind of, standoffish, but that's what it means is I'm not giving you back a value. You need to know that I'm not gonna give you back a value, so I still specify a return type as void.

Then the name, you know about the name, right? That can be any valid name for a method just like we talked about in Carroll, and there's some parameters. The parameters are information we pass into this method that it may potentially do some computation on. So I'll show you an example of that in just a second, and what parameters actually are, and how we set them up, okay?

Some functions may actually have no parameters, like the functions you wrote in Carroll, right? There was no information that got passed into those functions, in which case, the parameter list was empty, and all you had was open parens, close parens, which means there's no parameters for this function. When you call it, you just say the name of the function and open paren, close paren, okay?

So, with that said, let me show you one more thing and then some examples. So you've seen a whole bunch of syntax so far of while loops, for loops, variables, declarations, all this happy stuff, and now I tell you, "Hey, there are these methods, and this is, kind of, how you write a method." And you're like, "Yeah, that has a lot of similarities to Carroll." But there's this value that you're returning. How do I return the value?

Well, surprisingly enough, you use something called the return statement, which just says return, and then after return what's over here is some expression which just could be some mathematical logical expression, and when you get to a return statement somewhere inside a method, that method stops immediately. It does not complete the end of the method, and immediately returns whatever the value of expression is.

A lot of times, the return will come at the end of a method, but it need not come at the end. Your method, at the point that it's currently executing, will stop and return with that value that is expression as soon as you hit it, okay? So that's, kind of, a whole mouthful of stuff. What does this actually mean? Let me just show you a bunch of examples, and hopefully this will make it a little more clear.

We want some methods; give me methods. So here's an example of a simple method. This is a method that converts feet to inches, right? It's, kind of, like, "Oh, Mehran, yeah, this is, you know, yet again, an entirely useless thing for you $2,000.00 computer to do." But it shows you a simple example of all the pieces you need to know to write a method.

What's the visibility of this method? It's private. It's only gonna be used inside this class. What type is this feet to inches thing gonna give back to me? It's gonna give me back something that is of type double. The name of the function is Feet to Inches. What parameters does this function take? It takes in one parameter. The type of that parameter is a double, so what I'm expecting from you is something to double. It could actually be an explicit value that's a double.

It could be a variable that holds a double U inside it, but that's what I'm expecting is a double, and the way I'm going to refer to the double that you give me in this function is with the name Feet. So all parameters have a type and a name, and the name is the name you're gonna use inside this method to refer to that parameter.

So what do I do? You give me some number of feet as a double; I multiply it by 12, and that's what I'm gonna return to you. So when I get to this return statement, the method is immediately done, and it returns the value over here in the expression. Often times, you're not required to, but I like to parenthesize my expressions to make sure it's clear what's actually being returned. So let me show you another example. We'll just go through a bunch of them. Uh huh?

Student: Is it common to not put a comma as a separator for the type and then the name of the [inaudible] –

Instructor (Mehran Sahami): Well, the type and the name come together, so there is no comma there. Let me show you another example that actually has more than one parameter, okay? We'll get to that right now. So here is a function that's the maximum function, right? Again, it's private. It's gonna return an integer because you're gonna give it two integers, and it's gonna return to you whichever one has the larger value.

So it's name is Max. It's going to return an integer, and what it's gonna take is two arguments here, two parameters. Well, the first parameter, val one is an integer, and val two is also an integer. So the way I specify them is always they come in pairs, type and the name that I used to specify – type and the name I used to specify it, right?

So val one and val two are both integers. What do I do inside here? I just have an if-val statement. If val one is greater than val two, then val one is the max. So right here I'm gonna return val one. As soon as I hit that return, I'm done with the method. I don't go do anything else, even though there was only an else that I wouldn't have done anyway, I return val one. If val one is not greater than val two, then I return val two because val two needs to be at least greater than or equal to val one, so it's the maximal value, okay?

So, again, if you have multiple – you can have as many parameters as you want, sort of, within reason, right? And within reason I mean, like, sort of, a few thousand. You can have as many parameters as you want, and that's a lot of typing. They get separated by commas and they come in pairs, value and the name that you use to refer to it. Uh huh?

Student:[Off mic].

Instructor (Mehran Sahami):No, it will not print anything on the screen. All this will do is return that value to whichever place invoked this method. So if you think about something like read ints, right? Read ints gives you back a value, but it doesn't actually, I mean, that value happens to show up because these are typed it on the screen, but it wouldn't have printed it out otherwise, right? This doesn't print anything out; it just returns it back to where you, sort of, invoked this method from. Uh huh?

Student:[Off mic].

Instructor (Mehran Sahami):There is, and we'll get to it in about three weeks.

Student:[Off mic].

Instructor (Mehran Sahami):Yeah. So here's another one. This is something we refer to as a predicate method, and a predicate method is just our fancy, computer sciency term for a method that returns a Boolean, right? A method can return any type you want. This one happens to return a Boolean. Private Boolean is odd. What does this do? You give it an integer; it tells you back true or false. Is it odd, right? The name is pretty self-explanatory.

So it takes that variable, computes the remainder after we divide by two, and if that remainder is equal equal to one, this whole thing is gonna be true, and it'll return true. If it's not equal equal to one, that means it's equal equal to zero. In that case, this is false, and it'll return false.

So this whole expression here again because we have an equal equal, which is a logical operation, right, that's gonna give us something back that's true or false, and that's what we're gonna directly return to the person who called this function, okay? So any questions about that?

Alrighty. So let's look at something in a slightly larger context, right? Let's put this all together in a full program so you can get a notion of defining the class, and defining the methods in it, and you can, sort of, see the whole thing as we build it up. So here is something called the Factorial Example Program, and it just extends the console program, and all this puppy's gonna do is basically compute some factorials from zero up to max num.

So we have some constant here, which is the maximum number we want to compute factorials up to, and if you remember – let's take a slight, little digression to talk about

factorials. So if you remember in your math class in the days of yore, there was this thing that looked like this, all right? And you saw it and you went, "Oh, It's n!, right?" I hopefully caught your attention.

This was actually n factorial, right? Yeah, the explanation point – totally different now because a couple people were, like, sleeping, and "What? What was he talking about?" N factorial, and all n factorial is is multiplying all the numbers from one up to n!, okay?

So five factorials, strangely enough, is just 1 x 2 x 3 x 4 x 5, and one factorial is just one, and here is the bonus question; what's zero factorial?

Student: One.

Instructor (Mehran Sahami): Oh, I love it. It just warms the [inaudible] of my heart. Yeah, it's one, and there are some people who ask, "But zero, you know, like, I, uh – " and they get all wound up tight, and they're like, "No, man, mathematicians just said it was." Because if you think about it, it's multiplying – it's starting with one and multiplying zero additional terms, right? So I'm left with one. It's a good time.

So how do I actually compute that? Well, in my program I'm gonna have some run method that's going to count from zero up to max num and write out all the factorials. It'll say zero factorials this, one factorials this, two factorials this, by calling a method called Factorial passing in the value of the thing I want to compute the factorial of.

How do I compute the factorial? Well, I just add some method. This probably is gonna return an int. It's name is factorial. It's gonna take an n! of the thing I want to compute the factorial of, and how do I do that? Well, I start off with some result that's value is one, and I'm gonna have a for loop that's gonna count – here's one of the few times as computer scientists we don't count starting from zero; we actually count starting from one because it makes a difference.

We're gonna count from one up to and including n!. That's why this is less than or equal rather than just less than. So from one up to and including n!, so there's still n! terms I'm gonna count through. What am I gonna do? I'm gonna take my results as I'm going along and times equals it by I, which means the first time I multiply by one, store it back to result, then by two, then three, then four, all the way up whatever this n! value was, and when I'm done doing this loop, I return to my result, which is my factorial, okay? Any questions about the factorial method by itself?

Now, one thing that comes up, freaks people out a little bit, don't get freaked out; it's okay. You see an I here, and you see an I there, and you're like, "Oh, we just did that thing with nested loops, Mehran, where you told me that if one thing is inside another loop, like, I shouldn't use the same name. So why are you using the same name here?" And the reason why I'm using the same name is to make a little bit of an example, which is that the I here and the I here are different I's, okay?

When you think about a particular method, a method can declare its own variable. So this result, its lifetime or its scope, is until we reach the end of the method down here. This guy lives inside this method. This I is an I that only lives inside the factorial method. It doesn't interact at all with the I outside here.

So every method, and the interesting thing here is run is just another method. It just happens to be the special method that we always start executing from, but every method can have it's own declared variables, and this I here is just its own declarative version of I. It has nothing to do with the I up there, okay? Question?

Student: Okay, so is it possible to, like, give a variable from one method to the other method?

Instructor (Mehran Sahami): We'll talk about that in just a bit. So someone was wondering is there a way I can, sort of, give some variable from over here down to over here? And I'll show you how you have to, kind of, think about that in just a little bit, okay?

But any questions about this notion of the same I, that this I is actually a different I than that I. When you declare a variable in the method, you are getting your own version of that variable in the method. It has nothing to do with the same named variable in any other method, okay? Kind of, a funky thing. Uh huh?

Student:[Off mic].

Instructor (Mehran Sahami): That's just starting off the factorial, right? So if someone gives this five, we need to start off by saying we have some initial result, and then we're gonna multiply that by one, and then we're gonna multiply that by two. If we didn't have some initial thing we started multiplying, there would've been some unknown value there, and actually Java would've given us an error because it doesn't allow you to use uninitialized variables, okay?

So this is, kind of, a funky thing, but important to, sort of, think about. There is a method in the context of a larger program. Variables within methods are not the same as the same named variable in another method. That's the key take home thing, like, know it, learn it, love it, tattoo it backward on your forehead so every day when you look in the mirror when you take a shower, you see it, and you just know because that's one thing that trips people up. Uh huh? Was there a question up here?

Student: So when you return result, does it get mapped into I? Is that why [inaudible] –

Instructor (Mehran Sahami): Ah, good question, so when I return here, what's actually going on? When I actually could say I'm gonna compute the factorial of five, okay? This guy comes here – let's make it easy. Let's have him computing a factorial of two. So it multiplies one by one, then it multiplies it by two. I get two. It returns this two. What happens to the two?

That two goes back to the place at which this function was called. Where was that? That was up here. So this place up here where I said, hey, two plus your explanation point equals is what I would print on the screen. If I was equal to two here, right, two exclamation equals, and then what's the value that I get back when I call factorial of two?

Well, it computes factorial of two. It happens to be the value of two. It just sticks in a two for whatever you had as the calling method, okay? So you can think of when you call a particular method, the name of that method, think of it as getting replaced by the result that gets returned from that method, okay?

So if I run this program, just to make it clear, let me show it to you in its full glory over here. Here's a factorial example. I'll go ahead and run it, and you can see what its output is. Factorial example, we'll run it, and there it is, right?

So it says – if you can see up there, a zero factorial equals one because it writes out zero factorial, right, in the code over here. It would write out I has the value zero the first time through. So it writes out zero factorial equals, and then it calls factorial on zero, and what it gets back from factorial of zero is one.

So that one is just what gets displayed in the place of factorial zero because that value I get back from factorial zero is what's actually used here, and it keeps doing that all the way through the loop, right?

So down here it calls a factorial of nine, gets the value for factorial of nine, and returns that to the place where it's actually gonna print it out, okay?

So let me show you one more quick thing. Okay. Which is returning an object from a function, okay? So it turns out, interestingly enough, you can not only return, like, ints, and doubles, and Booleans from functions, you can also return whole objects.

So here is a method called Filled Circle, and what I'm gonna give to this method is some X and Y location for the center of the circle. Notice that's different than what G Oval Expecting to the upper left-hand corner. I'm gonna specify an X, Y location which is the center of the circle, a radius for that circle, and a color, and what I want this method to give back to me is an actual object that is a circle whose center is that XY. It has the given radius R, and it's filled in of that particular color.

Well, how do I do that? Inside here, I say, hey, I'm gonna create a new object that's of type G Oval, and I'm gonna call it Circle, and I'm going to get a new oval, right, and remember oval expecting the upper left-hand coordinate for the oval. I want the circle centered at XY, so what do I do?

I take that XY, and I subtract the radius from the X direction and the Y direction, which gives me the upper left-hand corner for the bounding box for that circle, and then I say give me something that has a width of two times the radius and a height of two times the

radius because that's what defines that circle, right? The diameter is two times the radius in both the height and the width direction.

So I get that oval. I've now called this Circle because it's my new object. I tell that circle to be filled because that's what I want to return is a filled circle. I tell that circle to set its color to be whatever color was passed in here. Notice the K sensitivity. Color with a capital "C" is a type. Color with a lowercase "c" is actually the name of the parameter that I'm gonna refer to, and that's what I use here is lower case "c."

So I set the color of the circle to be whatever color was passed in, and then I return this object. So that whole big box that contains this object that is some filled-in circle at that location, I say, hey, you, this is what you wanted because I'm giving you back a G oval. Here you go, and I give you back the whole thing, which means when you invoke this method, you better take the result and assign it to someplace that's a G oval, otherwise – or just add it to your canvas. You need to do something with it that you would normally do with a G oval, and you can't assign it to an int because an int and a G oval aren't compatible. Uh huh, question?

Student:[Off mic].

Instructor (Mehran Sahami): Okay, yeah, you can't assign it to an int; you have to assign it to an object of the same type. Uh huh, question back there?

Student: Why are X and Y set to doubles if the pixels are all whole numbers?

Instructor (Mehran Sahami): Ah, good question, so I could've actually made this with int and I probably should have made it with int, so it was my bad. Uh huh?

Student: In the red method, how would you add that oval? Like, what's this, like, syntax?

Instructor (Mehran Sahami): So the way you could think of it – let me just write it for you up here. So if I actually had the run method, so I'll blank, void, run, and then let's say I had this function filled circle, right? So I called the method Filled Circle, and let's just say I want to put this at location 10, 10, and its radius is two and the color that I want to give it is red, okay?

Now, the thing that this gives me back is a G oval object. There's a couple things I could do with it. I could declare a G oval out here, call this O, and assigned filled circle to it, and now I have the object O out here, which is actually that circle that this method gave back to me, and I can do whatever I want with it. Like, I could say add O, and it would add it to my canvas.

Alternatively, if I really wanted to, I wouldn't even need to set filled circle to some object that I want to keep track of out here. I'm just gonna say, hey, you're gonna give me a circle, I'm gonna add it directly to my canvas. So anything that I would be able to do in my run method with a regular G oval is the same thing I can do with this G oval that's

coming back from a method because it's just giving me back a G oval, and I can use it in the same way. Alrighty? I will see you on Wednesday then. We'll talk a little bit more about functions and methods then.

[End of Audio]

Duration: 52 minutes