

Programming Methodology-Lecture02

Instructor (Mehran Sahami):Alrighty, welcome back to CS106A. If you're stuck in the back, just come on down, have a seat. Originally, I thought maybe we would have slightly fewer people today than last time, but that appears not to be the case. So while we're waiting, everyone loves babies, so I decided to put – that's Karel, the Robot, the early days.

No, this actually – my son, and yeah, I know. He's a little bit older now but, like, he's got these little robot pajamas and he runs around all the time. So I'm like, oh, it's Karel, The Robot, and my wife looks at me like, it's your son. But that's a whole different issue.

Anyway, a few administrative announcements before we start. A couple things, there are four more handouts today, just because we like consistency. Four last time, there's four this time. They're all in the back. If you didn't already pick them up, you can pick them up after class, but they have all the information about downloading Eclipse, which is the environment that you're going to use for programming in this class, both for Karel and for Java.

There is also a special handout in there just on using Karel, so it talks about how Karel works in the Eclipse environment, and so you can get all set up with that. And so after today, you'll know how Karel works and you'll have the environment to use it. So surprisingly enough, you also get your first assignment today, which is actually in two parts.

So the assignment, the real assignment is the programming part, which is due on Friday of next week, October 5th. There is also an email part to it, where the email part is, we ask all of you kind folks to send an email to Ben, the head TA, myself, and also your section leader. So Ben and I get a whole bunch of mail, your section leader will hopefully get a little bit less mail.

But you won't actually know your section leader until after you go to your first section next week, which is why that part of the assignment is not actually due until basically one minute before midnight on Sunday. So I didn't make it midnight, just because then there's always confusion, midnight, which night? So 11:59 p.m. on Sunday, October 7th is when you should send that email.

That's not really the critical part of the assignment. The critical part of the assignment is the programming problems, which are actually all due in class on Friday. Okay, and then there's one last assignment, or one last handout, which is about how you actually submit your work in this class. You'll submit your work both electronically and in hard copy. The electronic copy, so that your section leader can run it and verify it. The hard copy is so that they can actually write comments on it, and then you'll do interactive grading with them as well.

A couple other quick announcements. The website, in case you weren't here on Monday and you don't know what handouts I'm talking about, and you don't know where to get them, go to the class website, cs106A.stanford.edu. There will be electronic copies in PDF format of all the handouts there, so you can get caught up if today is your first day.

You also need to sign up for a section. As we mentioned last time, section signups start tomorrow, 5:00 p.m. Sign up early if you want to have the most flexibility in terms of time, because look around, there's a whole bunch of people in this class. It's going to fill up quickly. So if you have constraints, sign up quickly, and the place you sign up is CS 198, not CS 106A, but CS198.stanford.edu/section. The 198 folks are the folks who lovingly run all of the sections and coordinate the whole program.

Last, but not least, just a word on the readings that I didn't mention last time. On the syllabus, you'll notice that pretty much for every day, or most days in the quarter, there is some sort of reading associated with it. That's the reading assignment that you should have done by that day's lecture, because that's what we will cover in that day's lecture.

So for today it should be the first three chapters of the Karel book, or the course reader. And if you're like, oh my God, I'm already three chapters behind, don't worry, it's like 20 pages or something. It's pretty lightweight, but you should do the readings by that time in class. So any questions about anything we covered last time, which is mostly logistics, or any of the sort of administrivia?

All righty, then let's get started on the real content. So you remember from the time before, we talked a little bit about Karel, the Robot, and here is Karel in that world that I showed you before. And there were avenues that run north-south, and streets that run east-west, and little beepers in the world. And Karel can face different directions, and there's walls.

And so now, we want to think about how do we actually program Karel. How do we get this little guy to do something interesting in the world? Okay, and it turns out there are four commands that Karel understands, okay, and those are pretty straightforward.

So here they are. You're going to get all of Karel's vocabulary in, like, one minute. There's a command called move, and move basically moves Karel one spot forward in the direction he's facing. So if he's on corner one, one, and he moves one spot forward, he's facing east. So he'll move over to two, one, basically. So one corner forward in the direction he's facing.

Karel also started being a good democrat, knows how to turn left. So he turns left. This is a lower case T, this is an upper case L. So turning left changes his direction by 90 degrees in the left hand of whichever way he's facing. So if he's facing east now, and he turns left, he will be facing north. And then he can turn left again and he'll be facing west.

Okay, now, question?

Student:[Inaudible]?

Instructor (Mehran Sahami):There is no space here. These are all one word. Good question, and I love it when you make it easy. All right, so besides turning left, there's also B-person Karel's world, and if he couldn't do anything with beepers, they wouldn't be interesting. So he can pick up beepers, which interestingly enough is called pickBeeper, again lower case P, upper case B, all one word, and there is putBeeper.

And what these do is pickBeeper, if Karel happens to be on a corner that has a Beeper on it, he picks up that Beeper, and stuffs it in his Beeper bag, in which case the count in his Beeper bag goes up by one. Or if it's infinite, it remains infinite, because he's got a real big Beeper bag, and sometimes he can infinitely many Beepers in there. It's just – we can talk about the infinity of Karel's Beeper bags some other time, but if you're interested, it's fascinating.

And he can also putBeeper, which means he can go to a corner and be like, "Hey, I'm just feeling happy. I'm going to put a Beeper down, say, on that corner right there." Except you can't throw the beepers, right. Karel's kind of limited. He's got, you know, if you look at him he's got no arms. He's got legs. No arms, so basically all he can do is he sort of like drops the little Beeper where he's at. So this puts it on the corner that he's at.

And these things are what we refer to as methods. Okay, methods are basically some instruction that we can call, okay, or use, like move, or turn left, or pickBeeper, or putBeeper. And what we say is Karel responds to this method. What we're doing is we're invoking, or we're calling a particular method on Karel, and he takes some action, which is basically what that method specifies to do.

Okay, so if we kind of think about this program, or if we kind of think about this world, maybe we want Karel to do something. So here's the initial configuration of the world. Maybe what we want Karel to do is pick up this beeper, drop it off at this corner, and end up at this corner facing to the east. And you can think about how we might do that with some of the instructions we have.

So what we want the final configuration of the world to look like, and I'll just put Karel at blinding speed. You can actually control Karel's speed here with this little slider. So I'm going to just show you the end configuration by running Karel so fast you can't even see it. It's just – he's blindingly fast. He's just that good, okay.

And so that's what we want to get to. Notice he's still facing east, but he's picked up that beeper that was on that corner at two, one, and moved it over to the corner at four, two. Okay, and so how might we do this? Right, so one thing we could consider is, what was the initial state of the world again?

So I'm just going to run this program again, and here I have a clip set up, which is your handout number five explains how to get this. But if I want to run a program, there is

these two little running person icons, or as you might notice, we even have our own menu in Eclipse, because Stanford's just that much fun, okay.

So under the Stanford menu, we have these two options for run. Import project is explained in the handout, is what you'll use. We'll give you some initial stuff for Karel that you'll start with, and you'll use import project to get it into Eclipse. But once it's there, we can run it, and so what we're going to do is say run this particular Karel program. So it's get – and we'll go through all this – it's get and go.

And this is our first Karel program that we want to run, and it's sort of, here's the initial world again, all big and happy. So I'm going to shrink this down a little bit, so we can see it over here on the side, while we think about what commands Karel would need to execute to kind of get to that final state we just talked about.

And over here, happily enough, let me resize a little bit, is our first file, which is empty right now, that we're going to write our first Karel program. So what commands might we consider Karel actually doing from the list you have here, to sort of affect what we want to happen in the world?

Student:[Inaudible].

Instructor (Mehran Sahami): Yeah, so we want to move. Okay, and then what do we want to do?

Student:[Inaudible].

Instructor (Mehran Sahami): PickBeeper, then move. Notice at that point we could have actually done something else. This is part of the art of programming. There's actually many ways to solve the problem. We're just going to happen to pick one. So we've moved. We've picked the beeper. We've moved again, and now what do we do?

Student: Go to the left.

Instructor (Mehran Sahami): Turn left. What happens if we turn left?

Student:[Inaudible].

Instructor (Mehran Sahami): Right, then what are we going to do?

Student: Move.

Instructor (Mehran Sahami): Move. And now what do we want?

Student:[Inaudible].

Instructor (Mehran Sahami): Yeah, conceptually what we'd like to do is turn right, right? At this point, Karel's like, "No, man. I'm just, you know, I'm left wing. What can I do? I turn left. That's what I do." And you kind of think about it, and you say, "That's all right, because really the world is just one big spectrum, and if you go around far enough to one side, you end up on the other." So if we make you turn left three times, that's equivalent to essentially turning 90 degrees to the right.

So here, we turn left, turn left, and turn left. All right, now what do we do?

Student: Move.

Instructor (Mehran Sahami): Move. So we sort of go up the step. PutBeeper and move to the last spot. Now, at this point we'd like to think, oh good times, we can just run this and it's a Karel program, and life is happy. In fact, this is not a valid Karel program. What this is, is an algorithm.

This is a recipe for doing something, and we'll talk in a lot more detail about different algorithms on Friday. But the way you can think of distinguishing between an algorithm and a program is an algorithm is essentially the recipe for doing something. The program is something that is valid syntactically according to the rules of the language.

And so Karel actually has some specific rules to its language that we have to apply to these statements to make them look a little bit different, so they follow the valid syntax of Karel. And that's what we're going to do now. We're just going to kind of go through this step by step, okay.

So one of the things we'd like to be able to do is, first of all, we want these to be valid commands. Right now, they're not valid commands. To turn a command into a valid command, or what we would refer to as a method call in Karel, after the name of the command we put an open paren and a close paren, and then a semicolon.

Okay, so move, open paren, close paren, semicolon is actually the valid move command, or move method invocation for Karel, and we do that for all these things. So to save a little bit of time, I'm just going to copy this and just go through and paste until the cows come home. Let's do a little paste there. Oh, it didn't copy. All right, let's try that again. We do a little paste, a little paste.

So we're going to go and turn all of these into valid Karel commands. Okay, so now you might think, okay they're all valid Karel commands. Are we ready to actually run this Karel program? And it turns out no, we're not yet ready to run – this is not yet a valid Karel program.

What makes it a valid Karel program is we need a few more things. One of those things is, we need to tell Karel where to start running. And you look at this and you're like, what does that mean? Like, doesn't he just start at the top? Does Karel start somewhere in the middle?

No, what it really means is that we're going to do is encapsulate a set of instructions inside something that tells Karel, hey, this is what you're going to execute. And the way we do that is we create something called a run method, and the syntax for that looks a little bit archaic, but basically it says, public, void, run, open paren, close paren, and then we put a brace. And that first brace says everything that you will now see until the closing brace is all part of what we refer to as the body for this thing called run.

So we come down here to the bottom and we put a closing brace, and everything between the open brace and the close brace is referred to the body, or what's actually executed when we do a run. Okay, so get to know where the brace keys on your keyboard are. They will come in very hand in programming, because you'll use them all the time.

One thing we also do with this is we're going to actually insert some tabs to make it easier for the human to read this program. Okay, so when we put in tabs, suddenly everything that's inside the body of run gets tabbed in, and when I look at it visually, it's much more appealing. I can just tell all the things that are actually inside run.

Okay, so we tab those in there and now we have the body of run is inside the braces. What we've actually done is created a new method. Well, run is here, is we defined for Karel a new method called run. And run is a very special method because that's where Karel begins running.

What he says is when he starts up, he comes into the world, he's like a brand new baby. He's like, "Hey, I'm here. I'm in the world. What do I do?" And he's preprogrammed to go find the method called run and start executing from the beginning of it. So he does executed in a top down manner, but he needs to have this run thing.

And so you might see them say, "All right, we're ready to go. Fire up Karel, right. I'm slipping a shrimp on the barbie." I'm sitting here barbequing while Karel's going and moving the beeper. It's like, no, we're not quite there yet. You're like, oh, what else do we need?

Okay, so one thing that we need is we need to take Karel. Karel is actually implemented in Java. If you've done some Java programming before, some of the syntax already looks familiar to you. If you haven't done any Java before, it's perfectly fine. You don't need to know anything about Java. What you do need to know about Karel is that Karel is defined in what we refer to as a class.

Karel is sort of a class of robot. There's kind of like the Karel 1000, kind of comes of the assembly line, and what we'd like to do is take those Karel 1000s and sort of mold them into our own version, which does some set of instructions that we'd like it to do. So in order to sort of say, "I want to create my own version of Karel," what we do is we create a class, and again we say public, and we say class now, instead of void.

So far, all you need to worry about, the stuff that's in purple, public void, and public class, is just kind of the standard syntax of Karel. And later on, all of this stuff will make

sense what it actually means, but right now just think of it as standard syntax, and just put it there, and life is good.

Public class, we now need to give this Karel program, in some sense, a name, and what we're going to name it, for lack of any better name, is we could call it our Karel program, because we did it together. It's kind of a collective process. So our Karel program, and at this point we're not done.

We need to actually say – you might have noticed after I type this why suddenly all this red showed up on the side of the editor. And it's like, oh, bad times, dogs and cats sleeping together, like what's going to happen? Well, what happens is we're creating our Karel program and it says, "Okay, you're creating Karel program," but it doesn't even know anything about the basic Karel 1000.

It needs to somehow know that this Karel program is creating a version of the Karel 1000. We'll call it the Karel 1001, and the way we specify this is we say that our Karel program extends Karel, which is sort of the basic Karel that actually exists and has these predefined instructions in it.

So we say public Karel, our Karel program extends Karel, and now we need to, again, encompass what our Karel, our particular class Karel is going to be running. So we put this inside braces and we say, "Yeah, this whole run thing is part of our Karel program." So we put another brace at the end, and to make it more readable, we go through one more time. Don't want to put a tab there, and tab everything so that it's nicely readable by a person.

Okay, so all this stuff now gets tabbed over one more spot, and even this brace over here gets tabbed one more spot. So now you can see, everything here is part of the run method, because it's nicely tabbed in. We can tell that, and all this stuff here is all part of the body of run, and all of it's encapsulated inside our Karel program.

Student:[Inaudible]?

Instructor (Mehran Sahami):Why is there a space, you mean like down here?]

Student:Yeah.

Instructor (Mehran Sahami):Just for readability. We don't actually need it. There doesn't need to be a space, but that's a good question. Oh, nice bank. I like that. Feel free to bank. If someone – you're on the rebound and I'll set you up for the rebound, just because you were the first one to do it. All right, so, and you might say, "Okay, now we've got our Karel program. We've got our run method. Are we ready for Karel?"

Okay, and you think we're ready for Karel, and someone's calling, saying, "You're ready for Karel. Do it." What I would actually encourage you to do is take a moment and turn

off your cell phones, unless your doctor. I actually would tell this to my students all the time.

I had one student who was a doctor once, and he says, "Actually, your class is when I'm on call, and if my phone rings, that means, like, someone's dying. So can I keep it on?" And I was like, hmm, yeah, okay. So he got an exception, but other than that, I'd encourage you to turn your cell phones, unless you have a really cool ring. Then you can leave it on.

All right, so at this point, you might say, "Oh, we're ready to run it," but even now, we're not ready to run it. And the reason why we're not ready to run it is, where does this Karel thing come from to begin with, right? Someone, somewhere had to say, "Hey, I'm going to build the original version of Karel that you're now extending."

And that actually comes from a place called Stanford.Karel. So what we do is we add a line called import, Stanford.Karel.*, and we put a little semicolon at the end of it. And what this tells us is, hey, go and get everything that Stanford's previously defined about Karel.

And when we get to Java, this'll become much more clear, but for right now, all you can think of that import's meaning is, go get me all the standard Karel stuff that Stanford's already done for me. And that defines this thing called Karel, which now we're extending in our Karel program. Uh huh?

Student:What is the semicolon for?

Instructor (Mehran Sahami):The semicolons always come at the end of the line and Karel's syntax, what they mean is this is the end of one statement. So at the end of the import, for example, that says import the stuff, that's the end of a statement, or for move, or pickBeeper, or turn left, that's the end of a statement. So we put a semicolon. Uh huh?

Student:Do the stars mean anything?

Instructor (Mehran Sahami):Do stars mean anything? Yeah, that star at the end of Stanford.Karel.* is actually important, and in a couple weeks you'll understand why. For right now, it just means get everything associated with Stanford.Karel. Uh huh?

Student:So then, is the run program [inaudible]?

Instructor (Mehran Sahami):It's all part of run. So when Karel starts up, and you'll see that in just a second, we start – we go to run and we just start executing the statements from run. So let's actually run this and see what happens, and I'll take a couple more questions in a second.

So we go up to our little run icon here. Actually, before we do that, let me cancel this, because we already have a running program over here. So let's quit out of the running

program and come here, and say run. And sometimes if you actually happen to write multiple Karel programs, and you hit the run, then this guy sort of looks like he's running with smoke all around him, which means he's running really fast. That means run the last program I was running.

The guy without smoke around him means, run one of the programs that I tell you to run, and it might give you a list if you have multiple programs. Here we're going to pick first Karel program, okay. So yes, we'll go ahead and save the resource because we forgot to save our file, and here is life.

And we'll run it slowly, and so we start the program, and there goes Karel. Woo-hoo! Congratulations. You are now all computer programs. Okay, because you got Karel to do what you wanted to do, and you're like, "Oh, that's so good. Let's do it again." And so you're like, "Yeah, let me start the program again." Wah-wah-wah, thanks for playing.

What happened? It tried to execute the same program again, but the world was in a different state, right? Karel was facing that wall, and the first thing we said is, "Hey, Karel, move." And he's like, "Okay." He's a little happy, scrappy robot. He doesn't know anything about the world, and he just, wham, into the wall. Thanks for playing. Dead Karel.

All right, no, because Karel's blocked, okay, and you get this little bug on top of Karel. It's like Alfred Hitchcock, the birds, he's just being, like, harangued by this little bug. But what that means is Karel tried to do something in the world that the world would not allow him to do, and this is what happens if you try to walk through a wall, or try to pick up a beeper from a corner where it doesn't exist, or you try to put down a beeper and your beeper bag happens to be empty, that's the kind of error you'll get.

So if you want to start this program again, you need to load world – and we happen to have first Karel program, you need to reload your world and then you can start it again from that point. Okay, so any questions about our first Karel program? Uh huh?

Student:[Inaudible]?

Instructor (Mehran Sahami):Yeah, he's moving basically from one grid point to the next grind point. So he just moves one step at a time. Uh huh?

Student:Is Karel case sensitive?

Instructor (Mehran Sahami):Karel is case sensitive, yes. Everything you type will be case sensitive. My throwing is case sensitive as well, and evidently, that was the wrong case. All right, so something else we'd like to be able to do with Karel, and a few people already pointed out is, "Hey, we created this thing called run. That's kind of cool. Can I create other stuff?" And as a matter of fact, you can, because that's just how snazzy Karel is.

And one thing, you might look at this and be like, "Ooh, ooh, ooh, I know what I want to create." What would you want to create if you could create a new command?

Student: Turn right.

Instructor (Mehran Sahami): Turn right. Right? That's a social, right? Everyone wants to turn right. All right, except the person who actually said, "Turn right." So what we can do is we can have a similar syntax that allows us to create new methods in Karel, like the run method actually is the place where Karel will always start. But we can create new methods that we could use, and so maybe we want to create a turn right.

So we could say void, sorry, here we'd say private void. Voice – sometimes typing is bad, especially when your hands are a little chalky. So run – you'll notice run is always public void run. For all the other commands that we're going to do as far as Karel's concerned, we're going to use private void, and the name of the command.

So we'll call this turn right, and this command will have a body, and what is the body of the turn right command going to be?

Student: Turn left.

Instructor (Mehran Sahami): Turn left three times. So I'm just going to copy from here and paste into – and the indenting even works out right. Rock on, right? And what I've now done – I'll go ahead and save this – is I've created a new command for Karel called turn right. So anywhere else in my program, I can now say, "Turn right," like here are these three turn lefts. I can replace them. Let me just get rid of them, and I'll replace them with a turn right, and I'll appropriately indent that.

And what that means is, as you're executing the program, you move, you pickBeeper, you move, you turn left, you move. When you get to turn right, it says, "Let me pause where I saw that turn right. I will go to the turn right method body and run all the instructions that are in that body, and then I will return back to where I left off.

So this program is exactly equivalent to the program we wrote before, now in terms of something called turn right, and just to prove that to you, we'll go ahead and run, and there it is, Karel does the equivalent of turn right and life is good. Okay, so we've created a new command called turn right. Question?

Student: [Inaudible]?

Instructor (Mehran Sahami): The location's not important. You can put it anywhere. I like to put it at the end, because that's convention in the book, but you can put it anywhere. Uh huh?

Student: [Inaudible]?

Instructor (Mehran Sahami): Why is run always public? Because that's where things start, and for Karel to know where it is, like private's all the stuff you're sort of like, eh, I'm keeping it to myself. But Karel's like – he's, like, looking around. He's like, "Where do I run? What do I run? And so you got to say, "Oh, it's public. It's available to the whole world." And in a couple weeks when we get into Java, you'll see public and private rear their ugly heads again and it'll all be clear.

Uh huh?

Student: [Inaudible]?

Instructor (Mehran Sahami): Pardon?

Student: If you made that a public [inaudible]?

Instructor (Mehran Sahami): It would still run. It would be fine. It's just good programming style. See, there's all these things that we kind of will eventually get to, but we got to start somewhere. So we sort of start right in the middle and we're going to build up from there, and eventually we'll go back and fill in some of the pieces underneath.

Uh huh?

Student: Do you have to [inaudible]?

Instructor (Mehran Sahami): Run is where – has to be called run. That's where Karel always knows where to start, but everything else could be something else. Like, instead of turn right, I don't know, what's the French word for turn right, or the French two words for turn right? Anyone speak French?

Student: [Inaudible].

Instructor (Mehran Sahami): Exactly. We could have called it that if we wanted to, and then rather than turn right we would use that, and it would be fine. So all the other things besides run, we can name them whatever we want as long as we're consistent throughout.

All right, so we could also – you might say something else might be kind of fun to do is to have a command called turn around, for example, which turns Karel around 90 degrees. So we'll just do a little copy and paste magic and have turn around here. We might not actually use it in this program, but just to have, and turn around is actually just two turn lefts.

It turns Karel around, and if you really wanted to do it, you could do it as two turn rights, but that's kind of six turn lefts, and that's not really a good idea. Okay, so because turn right and turn around are such common things to do, we actually have a souped up version of Karel. It's sort of like Karel with the training wheels taken off, and he's bad,

and he's buff, and he knows everything Karel knows, all four methods, plus two more, which is turn right and turn around built in. And that's super Karel.

So if we get rid of these and we say, "Hey, our Karel program, instead of extending Karel, is going to extend the super Karel model." Okay, oh, I know, it's dangerous, but it's okay. Super Karel, we've gotten rid of the turn right definition and the turn around definition, but if we run this now, start the program, Karel actually will do the right thing.

All right, let me quit out of this, unless my computer decides to freeze, and sometimes it decides to freeze, and that would be bad. All right, so at this point you're Karel programmers. You're rocking. You know all the Karel's methods. That's all he knows, at least for the time being, and you know how to actually make new commands by creating new definitions that have a body and the syntax around them to create new commands.

All right, so now what we're going to do is I want to ask you a philosophical question. I know, I guess you're like, this is a computer science class, why do we have to do philosophy in here? Because it's good for you, and the philosophical question is what is the downfall of the modern college student? Anyone know? You're like, what does that mean? You're like, I haven't fallen down yet. That's good. Hopefully it won't be that way.

But there is something, probably in your everyday life, that's just sitting there niggling. Uh huh?

Student:Procrastination.

Instructor (Mehran Sahami):Procrastination? It's related to that. What do you do to procrastinate in the morning?

Student:Sleep.

Instructor (Mehran Sahami):Sleep. Oh, sorry, yeah, and how do you prolong that?

Student:[Inaudible].

Instructor (Mehran Sahami):Snooze. And the thing about snooze is, it's amazing how quickly you can do math in the morning. I don't know about you, but my alarm has a nine-minute snooze on it. Why it's nine, I don't know, but it came out of the factory with a nine-minute snooze. And so you're, like, lying there.

I remember when I was a student, like, the dreaded, like, 10:00 a.m. class, 9:00 a.m. class. Heaven forbid you ever have a 9:00 a.m. class, and you're sitting there, and, like, the alarm goes off, and you're like, yeah, not going to brush my teeth today. Snooze. And then you think about it a little bit and nine minutes later, it goes off, and you're like, okay, it must be, like, 8:09 a.m. now.

And then the math kicks in. You're like, if I don't take a shower, that's like two more whacks on the snooze bar, snooze, snooze. And now you're up to like 8:27 a.m. You're just doing this all while you're, like, have asleep, right. It's like calculus when you're fully awake, yeah, impossible. Snooze bar, you could have, like, your brain removed in a lobotomy and you could still figure it out.

But what the snooze bar actually gives you is a way of doing something a certain number of times when you want to be able to do it a certain number of times. And so you might say, "Hey, Karel, sometimes rather than just, like, telling you turn, left turn, left turn, left three times, can I just tell you to turn left three times? Is there some way of doing that?"

And in fact, in Karel there is, and it's something called a for-loop, and the syntax for this looks like this, for, just the word for, paren, int I equals zero. And so this might look a little bit strange, and if you've done some programming before, it won't look strange, and if you haven't done any programming, this is just the syntax. Just use it and feel good about it.

This is int I equals zero, semicolon, I is less than the number you want to count up to. So if I want to do something three times, I'd have a three here, but in general this could be any number that you want. Then another semicolon, and then an I followed by a plus and a plus with no spaces, and then a close paren.

And then after all that, you have a brace, an open brace, and there's a close brace down here, and everything that goes here is the body of what we refer to as the for-loop, and that body gets executed however many times you say here. It can be one instruction. It can be more than one instruction. So turn left could actually be defined – or turn right could be defined as for I equals zero, I less than three, I plus, plus, turn left.

And what this will do is it will execute this three times. I could have multiple instructions in here if I want to, and it would execute that whole set sequentially three times, but this is the equivalent of turn right. Okay, question?

Student:[Inaudible]?

Instructor (Mehran Sahami):So the spaces are, sorry, there is a space here. Well, you don't actually need a space there, but there are no spaces in here. So you need to have this space for sure. The rest of these spaces are basically optional, but we like to kind of space them out for it to make sense. You definitely should not have any spaces in the I plus, plus. There's no spaces in there. Uh huh?

Student:Is all of this syntax in the course reader?

Instructor (Mehran Sahami):It's all in the course reader, yeah. Yeah, so you don't need to worry about – if you want to take notes, that's great, but it's all kind of in the course reader if you want to follow along. So you can do something any number of times that

you want, right, and it'll just count up. So this could be, like, if you wanted to, you could say, "Hey, turn left six times," and that's still the equivalent of turn right.

Or we can do the equivalent of snooze, which is that. Right, so basically what does snooze do? Nothing. You just sit there, and you spin around a few times, and you're kind of like in the same spot, facing the same direction when you were done. Okay, but anytime you want to do something some number of times, you can actually do it using this for construct.

And this is the way you should remember it. You're always just going to use this syntax in Karel, and the only thing that's going to change is this number of times that you want to do something. All right, so besides that there's also sometimes, though, where you don't know how many times beforehand you want to do something.

And you might say, "Huh, that's kind of weird. When would I not know how many times I want to do something?" Well, let's say I'm Karel, and I'm standing here, and there's a wall over there. And I know there's a wall somewhere over there, because my world is finite. It has to end at some point, but one told me how many steps it is to that wall.

So if I say, "Hey, I'm going to take ten steps to that wall using a for-loop." One, two, three, four, five, six, and then I'm like, oh, dead Karel, right, and that's a bad time. So sometimes, we just don't know when the program starts, and what we want to say is, "Do something while some condition is true," like, keep going forward as long as there's no wall in front of you. And when there's a wall in front of you, stop going forward.

All right, so the way we do that is something called a while-loop. So at this point, hopefully you're intimately familiar with all of Karel's instructions. And the way the while-loop works, while, because that's what you want to do something.

You want to do something while there is some condition that's true, is the syntax looks like this, while, and then a paren, and inside here we're going to have some condition. Okay, so I'm going to put a squiggly line under it. So you don't put the word condition. We're going to put a condition in there in just a second, a close paren, and then we're going to have some body inside braces, open brace, close brace.

Now, where do these conditions come from? So one thing we can think about is the notion of moving until we get to a wall. And so Karel, it turns out, has some things that he can sense about his world, which are conditions you can check to see if there's something going on in this world.

One of the things you can check is front is clear, and that's an open paren and a close paren, and so there's actually two parens. Lower case F, capital I, capital C, all one word. And so what that asks Karel to do is say, "Where you're standing right now is your front clear," which means is there a wall right in front of the direction you're facing.

If your front is clear, so while your front is clear, what it does, it basically checks if the front is clear, and that's true, then it executes the body of the while loop. So we could say, for example, move here. And so what it's going to do is, while my front is clear I move and it's still clear, and it's still clear, and I just keep doing this.

And at some point, I move along, and my front is no longer clear at this point. Front is clear is no longer true, and so I stop. Okay, and basically at that point I stop executing this loop and I just pick up from right after the close brace. So I'm after I'm done executing a loop, whether it's a for-loop or a while-loop, I just pick up executing after that close brace.

Okay, so I could actually turn this, for example, into a new method, private, void, walk to wall, or we'll call it move to wall like that. And then anytime I basically want to tell Karel, "Just keep moving forward until you hit a wall," I just essentially invoke, or call the move to wall method, and it will just get me up until I get to a wall.

Now you might wonder, hey, where are all these conditions and what are all the conditions? So if we can get the overhead for just a second, I'll show you. Page 18 of your reader. So on page 18, there's the lovely table one. Oh, lovely table one, and these are all the conditions you can check in Karel. There's front is clear, left is clear, right is clear. So you can sort of check to your left and right.

There's no behind is clear. Important for babies, not for Karel. There's beepers present, which means is there a beeper, at least one on the corner that you're at. Are there beepers in your bag? Is your bag not empty? Are you facing north, east, south, or west? So you can check all those conditions.

You can also check the opposite of all the conditions, like front is blocked. So you can actually check, like while your front is blocked you might want to turn left, because you want to keep turning until eventually your front gets cleared up, and then maybe you go do something else. Or your left is blocked, or your right is blocked.

So these are all – I'm not going to go through them all, but they're all sort of obvious. They sort of give you the obvious information that you would expect about Karel's world, and they're all in your course reader on page 18.

All right, so given that you have these conditions you can check. Sometimes you want to do something while some condition is true, and as soon as this condition is no longer true, you stop executing. There's also sometimes that rather than doing something over and over, you just want to say, let me check some condition and if it's true, then do this one thing.

So I'm not going to do it as a loop, but I'm just going to do one thing. Okay, and that's something called an if statement, and the way an if statement works is it's if, space, open paren, some condition, again I'll put this condition in squiggles, open brace, close brace, and this is the body. And it says, if some condition is true, then do what it's in the braces.

And if it's not true, just completely skip over this and keep executing from right after the braces.

So you might want to say, "Hey, I only want to pick up a beeper if I know a beeper exists in the world." So I sort of want to have a notion of having a safe pickup of a beeper. So I could say if beeper is present, which means there is a beeper, at least one on the corner that I'm at, then it's safe to pick a beeper.

Because if I didn't necessarily check before, and I didn't know there was a beeper there for sure, I might try to pick up a beeper on some corner, and there isn't a beeper, and Karel blows up, and life is unhappy, and tears well in your eye, and the whole deal. All right, so if statement, that's just what we refer to it as kind of as an if statement. It's all lower case.

Now, sometimes what happens in life is you don't want to just say, "If something is true, then essentially what you want to do is this." You want to say, "If that thing is false, then I want to do something different." Okay, so you might want to say, "Hey, if there's a beeper present, I'm going to pick I up. But if there isn't one there, oh, Karel feels a little sad and he's going to put a beeper down to, like, plant a new tree because he wants to fight global warming."

So else and then we're going to have another body here, putBeeper, assuming that Karel has beepers in his bag. So what this says is, if there's a beeper present, then pick it up. If there is not a beeper present, it doesn't try to pick a beeper, it skips this part and does the part that's referred to by the else. So we'll put a beeper. So it will only execute one of these bodies, and which body it does depends on whether or not there are beepers present. If it's true, it's this one, if it's false, if that's one.

Question?

Student:Is that a space or a tab [inaudible]?

Instructor (Mehran Sahami):These are spaces here. So these are required to be spaces, and the syntax is also all in book too. So you can see where the spaces are there as well. Uh huh?

Student:Can you make multiple [inaudible]?

Instructor (Mehran Sahami):For the time being, you should just check one condition at a time. Now, you can actually sort of, what we refer to as nest, these kind of statements. So you could say, "If beeper is present," you might say, "Hey, if there's a beeper present there, that's some marker that I should move forward if there is not a wall in front of me." So what I want to do is, if there is a beeper present, what I now want to do is check to see if I can move forward.

So if front is clear, move. And this is what we refer to as a nested expression. Notice how I sort of tabbed it in a little bit, so that it's kind of nested inside this other guy. What it means is check to see if beepers are present. If they're not present, then it skips this whole body and it just puts a beeper down.

If there is a beeper present, it's going to execute this body. Well, what is the body? It's another statement that says, "Is your front clear?" If your front is clear, you say, "Hey, I was on a beeper, and my front is clear. Now, I'll take a step forward." But if there was a beeper present, and my front was clear, or was not clear, I'm not going to take a step forward, in which case it doesn't do the if part. It finishes up and there's nothing else for it to do here. So it's done with this whole part and it's done with the whole if-else statement.

Okay, any questions about that, what's actually going on there, right? If beepers are present, you know you're only going to execute this part. This part is now a goner, because that's the else part, and beepers present was true. So to do this part, you come in here and you say, "Is front clear?" No, front's not clear. I have some wall in front of me.

Oh, okay, well I'm not going to do the move. Is there an else over here? No. This else is not attached here. This else is attached to this if that says, oh, there's no else. Okay, there's nothing else I should do. Are there any other statements over here that I should execute? No, I'm done with the whole body for this part. So it's done with the entire statement and it continues executing from down there.

Okay, so any questions about that? Uh huh?

Student: So is it every brace always comes in pairs?

Instructor (Mehran Sahami): Every brace comes in pairs. You always have an open brace and there's going to be some close brace that's going to define the body. Oh, that was the double. All right, so what we're going to do is we're going to put this all together in a big program.

Okay, so we're going to use all these things in a big program, and this program, I'll show you, is basically, Karel is going to run a little steeplechase. You're basically sort of like running hurdles, okay. So we're going to open up another program. Don't worry about all the details right now.

I'm just going to run this so I can show you what it looks like. So you can see what the world looks like and what we want Karel to do. So we're running, we're running, steeplechase. So what Karel's going to do here is they're the world that is always nine avenues long, and he knows that. It's always nine avenues long, which means there are, at most, eight steeples in the world, or eight hurdles.

The reason why we call them steeples as opposed to hurdles is because some of them are big. Some of them are just ginormous, and so what Karel needs to do is he needs to start here and end up over here, and run around every hurdle he encounters. So how are we

going to do that using the stuff that we've learned, and putting it together into a larger program?

Well, the first thing we're going to do is we need to start with our friend, the run method. Right, so in the run method, all, Eclipse doesn't jump out in front of me. Right, so notice we have all the kind of other boilerplate that we want to have. We import Stanford Karel. We're going to have some public class we'll call steeplechase that extends super Karel. Right, all the stuff you've seen before as boilerplate.

We're going to define the run method and what the run method is going to do is, if you have nine avenues you need to go through, that means intervening the avenues, there's at most eight steeples. So I want you to do something eight times. So we're going to have a for-loop that executes eight times. What are you going to do inside there

Well, you have a choice. When you look at the world, either you're facing a steeple or you're not. So if we look over here, right, here Karel starts out facing a steeple, but after he jumps over it, hey it's clear. So he could actually move forward before jumping the next steeple. So that's essentially what we check for in the program. At every step, what we say is, "If your front is clear, there is no steeple for you to jump. So just move ahead.

But if your front is not clear, you got some work to do buddy. So jump hurdle." "And you're like, jump hurdle. I've been sitting here for 45 minutes. You never told me about jump hurdle. Where is jump hurdle?"

And I'm like, "Yeah, I didn't tell you about jump hurdle because we're going to write jump hurdle," and you're like, "Oh, yeah, new instruction that we create." So we're going to break our program down into smaller pieces. So what's jump hurdle about? How do we jump a hurdle?

Well, think about what you want to do to jump a hurdle, and I'll put it in the simplest possible terms. You want to ascend the hurdle. You want to go up. You want to move past the hurdle, and you want to descend the hurdle to come back down. You're like, "Yeah, thanks, Ron, that really helped a lot. How do I ascend and descend the hurdle, right?"

So what we're doing is we're breaking the program down into more and more steps, and notice this is reading like English. And the important thing to realize in programming is programming is not just about writing a program that the computer understands. Programming is about writing programs that people understand. I can't stress that enough. That's huge. That's what programming style is all about. It's what good software engineering is all about.

Writing a program that just works, that someone else can't read and understand, is actually really terrible software engineering. And I've seen software companies do this where some hotshot programmer comes along and they're like, "Oh, I'm great programmer." And they go and write some program to do something, and then it breaks.

And then they come along and they say, "Hey, we need to actually get that program to do something slightly different, or we need to upgrade it. And it's written in terms of the code so badly that they can't do anything with it, and they throw it out completely, fire the programmer, and get a team of good software engineers to actually do it. I've actually seen that happen multiple times.

Okay, so write programs for people to read, not just for computers to read. Both of them need to be able to read it, but it's far more important that a person reads it and understands it, and that the computer still executes it correctly. But that's the first major software engineering principle to think about.

So how do we ascend and descend the hurdle? Move we knew about. Well, how do we ascend the hurdle? If you think about ascending the hurdle. Right, what that means is you're basically facing a wall, some wall that goes up some amount that you don't know.

Well, how am I going to get up that wall? I'm going to turn left, so now I'm looking up the wall. And guess what? If I put my right hand out against that wall and keep moving up along the wall, I can eventually get to a point where I've gotten to the top of that wall. So I turn left to face north, while my right is blocked, which is why that wall is there. I keep moving all the way up until my right is no longer blocked. That means I've extended all the way up the hurdle.

At this point, I'm still facing north. This guy's expecting me to face east, so I can move over the hurdle, right. If I'm facing north, I'm just going to step one more up. So I turn right at the end to now be way above the hurdle and be facing an open place where I can actually do that move.

So now, I do the move. I'm on the other side of the hurdle. I need to descend the hurdle. Well, how do I descend the hurdle? Ah, this is where our friend, move to wall, that we just wrote, comes in handy. So I'll show you both of these at the same time. If I want to descend, I'm at the top of the hurdle, I turn right.

So I'm looking south now, I'm looking down, and I just move all the way down to the wall. It hits the floor, and this is move to wall that we just wrote, right, while front is clear just moves. I move all the way down, and so when I move to wall and I get to the wall, I'm facing south. I want to turn left one last time so I'm facing forward to go again. Okay, so any questions about that?

Let me run Karel, so we can see if he's feeling up to the challenge. So here he is. We'll start the program. There he goes. Notice what he's doing. He goes over every hurdle. If his front is blocked, he goes up and over. Here's one where you can see it.

He goes all the way up until his right is clear. Then he moves and he comes all the way down until he hits the wall, and he keeps doing this. And if he does this eight times, he always ends up at the end because we know it's guaranteed to be nine avenues long. Uh huh?

Student:[Inaudible]?

Instructor (Mehran Sahami): Yeah, good eye. So you might say, "Hey, before when Karel turned right and he was making three right and left turns, what's going on now?" It's super Karel, baby. We're loving it. Super Karel not only knows the three turn lefts, just turn right, but he's like, "Hey, forget turn left, man. I'm just turning right." So he knows, or she knows, or it knows, or whatever, that amorphous form in the sky knows.

Uh huh?

Student:[Inaudible]?

Instructor (Mehran Sahami): Good question, because that's not the specification of the program, right. So one question you could ask is, like, "Hey when I bought my word processor, I bought my word processor to, like, write essay with. So how come when I got my word processor," and this is a valid question to ask, "Why didn't it come with all the essays I needed to write at Stanford?" Right?

I mean, it would be nice if it did, right. I'd go pay the \$400. I'd be like, woo hoo, super Karel, baby. I got super word processor. Right, because that's not the specification of the program. The specification of the program is to go over the hurdles one by one.

Now, just to see if – and you might say, "Oh, that's kind of interesting, but does it just work on this one world?" So let's consider another world, super steeplechase. So we can load a world, say you can just click load world, and there are some worlds that we'll provide to you in your starter project, which is all explained in the handout.

Super steeplechase. So here's super steeplechase. Yeah, that might look familiar. More bars in more places, right. So here's Karel, right. This is like a world that 50 high. That's Karel down there and you're like, "Oh, man, you're really making," yeah, we're making him work. All right, so go little Karel. There he goes. Oh, oh, come on, give him a little encouragement. Go Karel, go Karel, go Karel.

Aw, yeah, because you can just speed him up if your program's running slow, but there he goes, right. Huge steeples, he's done, same program works. So the important lesson that comes from that is to think about the generality of the program, right.

You don't want to just write a program that works in one particular world, unless that's the only world you need to worry about. You want to generalize your program in a way so that it's using, in some sense, general principles that will apply to any world that meets your specifications. Right, it still needs to meet the specification. Our world is still nine avenues long and we know what the specification of the steeples are.

All right, so any quick questions before we go? Uh huh?

Student: So if you had the [inaudible]?

Instructor (Mehran Sahami):No, because it's still just nine avenues. The for-loop counter stays the same, right. It's that while-loop that's actually going up and down the hurdles, because it doesn't know how big the hurdles are. Any other questions?

Student:[Inaudible]?

Instructor (Mehran Sahami):That it's nine avenues and the hurdles are just pools, basically, of any length.

Student:[Inaudible].

Instructor (Mehran Sahami):Yeah, all right, I will see you on Friday. If you have any more questions, come on down.

[End of Audio]

Duration: 48 minutes