Programming Methodology-Lecture03

Instructor (Mehran Sahami):Couple quick announcements before we dive into things. There's one handout, which, hopefully, you should have gotten. It will contain the Karel example we did in class last time, the steeple chase, as well as some more examples that we're gonna go over this time. But I encourage you to actually pay attention to what we do in class rather than sort of looking at it on the handout, because one thing that's always true about programs is once you see the solution to a program, it's easy to lull yourself into thinking oh, I could have done that, it seems so easy.

But when you're actually sitting there staring at sort of a blank page on your monitor, it's a lot harder to generate the program. And we'll actually generate a couple of programs together in class today, so I'd encourage you to sort of take part in that that we do in class, and then you'll have the actual listing for the code to take home with you so you don't need to worry about scribbling everything down quickly.

A couple quick announcements, the class website, one more time – every time I look on Access there's more and more people enrolled in this class, which begins to frighten me after a while but that means there's people who haven't gotten any of the previous handouts.

So if you need to know where to get all the previous handouts if you just joined the class, CS106A.Stanford.edu is the website for the class, you can get all the handouts there, and there's a whole bunch of stuff you should read right away.

Section sign-ups. How many people have already signed up for a section? Oh, yeah, that's a good time. If you haven't at this point, you should have looked around at all those hands that went up and just realized that sections are just filling up to the left and right of you, literally.

Section sign-ups are open, and if you go to CS106A.Stanford.edu/section, you can sign up for a section. The section sign-ups close on Sunday at 5:00 p.m. You need to sign up for a section in addition to enrolling on Access to actually be enrolled in the course – important thing.

Another thing just real quickly is just for your convenience, I mentioned last time sort of the conditions of the [inaudible] about the world. For your convenience, we actually posted them up off a link on the announcements on the CS106A website, so if you're programming somewhere and you forgot your little Karel book and you're like what are all those things that Karel can check in the world, like is front is clear, or is facing north or whatever, you can just go to a link off the website and you'll find them all listed there.

One other thing I would encourage you to do is we haven't been picking up the audio on questions that are asked, so one of the things I'd encourage you all to do right now is you'll notice there should be microphones on the fronts or on the backs of the seat in front of you.

Pick up the microphone right now, just pull it out of the slot and just keep it in your lap. Don't worry, it won't damage anything on you personally. It's just kind of fun. You can keep you there, it'll keep you warm, and that way if you ask a question it'll remind you to use the microphone, all right?

So with that said, I wanna take a quick poll, because last time you got your first assignment and the download instructions and everything. How many people have already downloaded Eclipse? Oh, rock on, good time. How many people have gotten the first assignment filed that you needed to get? And how many people have started on the first assignment? Wow, I love that. Oh, that just warms the cockles of my heart.

How many people are done with the first assignment? Oh, yeah, just [inaudible] and you're like oh, yeah, I got Karel to run around, and it's just a good time. So you've still got a week to do it, but that's a good thing to keep in mind.

So a little bit of additional background on Karel before we dive into the real meat of things, okay? One thing you may have noticed is all the Karel code that you've either started writing, if you're already working on assignment number one or all the code that we write in class is all in file that ends with dot Java. If you haven't noticed that before, you'll – now you know, it ends with dot Java.

Because in fact, Karel is all implemented in Java. So one of the things you might be wondering is hey, I know a little Java; can I use some Java in conjunction with Karel? And the answer is no. For the purpose of the Karel assignments, you should just use the constructs that you've been shown in class and in the Karel reader, just keep it to those constructs. That still gives you plenty of stuff to do with Karel, it's a good time, but keep it to that, and actually starting on Monday we're gonna, like, sort of leave Karel behind and say bye-bye, Karel, and I'll be, like, bye-bye, I love you, call me. But we'll get into Java, so if you know Java now, just use the Karel stuff in the Karel assignments that we actually do.

All right, so with that said, any questions to start off with before we dive into something new? Um-hm?

Student: How do you stop the program? Like, I had a problem, it goes up into the corner and just kind of like it says an error message.

Instructor (Mehran Sahami):And it kept looping there forever?

Student: Yeah, and I tried making, like, an empty program called Stop, but that didn't really work.

Instructor (Mehran Sahami):No, just go up to the little icon – or the little thing in the top of the window that allows you to close the window and just close the window. Karel will be okay, he knows how to deal with that.

As a matter of fact, that's – I am so glad you asked that question. It's just a wonderful thing, because it actually leads into the first topic that I want to talk about, which is common errors. And so there are some common errors that may come up, and these are good things to kind of know about. And one of them is the thing you just ran into.

But before we actually talk about that at sort of the level of Karel, I wanna ask you a question: [inaudible] another one of these strange questions. How many people have actually ever read the instructions on a bottle of shampoo? A few folks – oh, wow. Man, that's more than – I was, like, you've really gotta get out more often, right?

But I'm surprised that many. And what do those instructions say? Rinse, right, that's where you rinse your hair, and then you lather, and then you repeat. And if anyone actually followed these instructions, you would still be in the shower now. As a matter of fact, you'd be in the shower for the rest of your life. Why? Because you rinse, you lather, and you repeat, which means you rinse, you later, and you repeat.

And you just keep doing this, and it's like Karel just taking a shower, right? And he's like you keep telling me to rinse – to linse, rinse and lather together. Rinse, lather, and repeat, right? And this is what we refer to in program-speak as an infinite loop. Which as you can guess is a loop that keeps going forever. And this may come up in your Karel programs. Here is an example of an infinite loop. You might be standing somewhere and you might say while front is clear.

Oh, well, if your front is clear, I want you to turn left, because, you know, I have this idea that eventually you'll turn and you'll be facing something and your front won't be clear anymore. And that might be a great idea to have, but what happens is if here's our friend Karel, happens to be standing in the world and guess what? There's no walls around him. What does he do? He rinse, lathers, and repeats, right?

He just keeps sitting there, turning left on this corner, and his front will never become blocked so he'll never get out of that loop, okay? That's one of the places where the syntax of the program is perfectly valid, but in terms of our intention for what we wanted the program to do, it didn't work as intended.

This is the more common thing that you're actually going to need to deal with fixing in your programs as opposed to syntax-type problems. As a matter of fact, I won't mention the name of the company but there was a company years and years ago that put out this advertisement that was, like, our processors are so fast they can execute an infinite loop in 2.5 seconds.

And you kind of looked at that and you were like ah, ha, ha, funny computer scientists. But the thing that was actually funny about it was the company that put out these chips, the chips actually had a flaw in them where if they executed things really fast they would warm up and then they would melt. So in fact, they would execute an infinite loop in 2.5 seconds, and then the processor would catch fire and it would stop.

So sometimes, that happens. Hopefully, that won't happen to your machine. Don't worry, even if you get Karel going real fast, it's okay. If your processor melts, come talk to me, it's never happened. But it'll make for a wonderful story in class, and it'll cost you about $1,800.00. Fine.

So another kind of error that comes up fairly often also has a little affectionate name to it. And this is something that we refer to in programming speak as an off-by-one bug. Or if you take the first letter of that, it becomes an OBOB. And sometimes you'll hear people refer to OBOB, and if you're name is Robert they're not saying, like, oh, Bob. No, that's just referring to the off-by-one bug.

And what that means is you forgot to do something one more time than you really needed to, even though logically it didn't seem like you needed to. So the best way to understand that is with a little concrete example. So let me actually give you a concrete example by going over to the computer.

So here's Karel sitting in his world, and one of the things we might wanna do with Karel – whoa – Karel in his world is to say that we actually want Karel to lay down a line of beepers, right? Just whatever row he's in, just lay down a line of beepers till you hit the wall. And that's a fairly common thing that you might wanna do, okay?

Now one way we might be inclined to actually do this if we were to write some code is to say something like hey, I'm gonna, you know, write a new class, I'll call it fill row that extends super Karel. I import all the Karel stuff, so I'm doing all the crazy stuff that [inaudible] told me to do, and hey, I want to keep putting down beepers until I fill the row. So wouldn't it make sense to say while front is clear, right? There's no wall in front of me. Put down a beeper and move to the next corner.

And as long as my front is clear I'll put down another beeper and keep taking another step and doing this over and over, and this seems like – hey, that seems like reasonable code to lay down a line of beepers. What happens when I do this? Yeah, so let's actually run it. Let's bring up our buddy Karel. Where'd you go, Karel? Here you are. And run that program. And so what Karel does, he puts down a beeper, his front is clear, so he moves, puts down a beeper, front is clear, puts down a beeper, front is clear, puts down a beeper, front is clear.

He moves here, front is no longer clear. Before he puts down the beeper, he says hey, my front isn't clear anymore, I stop. He's off by one because there's one more thing that he needed to do, which was to put down another beeper in this last spot. This is another very common logical error that comes up – you're off by one in terms of what you're thinking, and so lo and behold if we go into the program over here and we look at this comment, and I'll tell you about comments in just a second, it says this is buggy, it's off by one. You need to add a final put beeper.

And so what we do is we say hey, after you got out of your loop, your front is no longer clear, but you haven't put down a beeper on this corner yet. Put down a beeper, right?

And if we save that puppy off and we close this off and just so we know that it's in fact running, doo doo doo, we run our little program and rock on, all right? No more off-by-one bug, life is good. So any questions about the OBOB or the off-by-one bug? That's another common thing.

These are just a couple of common things I wanna show you, so, you if they come up in your programs you don't feel like oh, you're out there adrift, all alone. Like, these have been done millions of times by other very qualified programmers. Don't worry, it happens to everyone.

All right, so one of the things I just mentioned is this thing called the comment. This thing that's up here in green. What's that all about? So as I mentioned to you last time, one of the key software engineering principles is to write programs that are understandable by people, not just by machines.

And so what a comment is is a way of being able to put something in your program that another human being reading your program can actually read and understand, and it actually has no impact on the execution of the program at all.

And so the way the comment works is it starts off with a slash and a star, and now everything you put – it can even span multiple lines – until you put a star and a slash is a comment. It's there just for the person to read, and it doesn't affect the program at all.

And so what I would encourage you to do as part of good programming style is all of your programs up at the top should have a comment that says what the name of the file is and has a little bit of an explanation about what your program actually does.

And if you're wondering how much of an explanation or how in-depth it should be, be it as in-depth as you want, but in the handout you got today you see examples of a bunch of programs, and they're all commented, so you can see examples of comments in there.

There is also shorthand you can do. There is a comment that's just slash, slash, and that just means everything on the remainder of this line is a comment. So it actually has no close, in some sense, for the comment like this does explicitly. Everything else to the rest of the line, when you hit a return, that's how it knows it's done with the comment.

So sometimes if you wanna have a one-liner comment somewhere, you can just put a slash, slash.

Student:[Inaudible.]

Instructor (Mehran Sahami):Yeah, when you get assigned to a different section leader, you'll actually put your assignments in with your name and your section leader, and that's how it'll get differentiated. So wait until you get your first section, and then you'll see how submission stuff works. Um-hm?

Student: What about when you write private programs [inaudible] public ones, do those need comments, too?

Instructor (Mehran Sahami):Oh, when you write, like, other methods? Yeah, you should comment those as well. So methods should also be commented as well, and I'll show you an example of that today, actually, so you can see an example of how we do that.

So that's actually – and I love it when everyone just, you know, it's like ringers in the audience. You just lead me into the next topic, and that's a perfect example to say hey, let's actually look in, look at a program that we did last time, which was the steeplechase program, now with comments, okay?

So up at the top – sometimes you need to click this little plus sign to expand the comment, because Eclipse has a tendency to sort of minimize the comments automatically. So just click the plus sign and that'll stand them out.

Steeplechase, so we have a nice little comment here at the top of our program. Notice this is a Java file. As I mentioned before, it ends with dot Java, because Karel's actually implemented in Java. But you should use no features of Java other than what is in the Karel book. And if you're like hey, [inaudible] I don't know Java, hey, nothing for you to unlearn. You're good to go, because you just know Karel, all right?

The other thing that's going on here is you'll notice that inside the program we have comments as well. So this method, the run method, it says to run a race that's nine avenues long. We need to forward or jump eight hurdles eight times. And that makes it explicit to the person, right?

If we didn't have that comment, someone would come along and say hey, the race is nine avenues; why are you only iterating this eight times, right? And that's a natural question to ask. And so you put in comments to clarify things in the program, which are not obvious, okay?

Another thing related to that is what's referred to as pre-conditions and post-conditions. What did you expect to be true before you called a particular method or before a particular method is invoked, and what's going to be true afterwards?

So our friend Jump Hurdle here – remember Jump Hurdle from last time? We ascend the hurdle, which means we go up, we move to cross over the top of the hurdle, and we descend hurdle to come back down the other side?

Well, what we needed for that to be true for this to actually work right is the precondition is you're facing east at the bottom of a hurdle. Which means the hurdle was right in front of you and you were facing east in front of it. That way, we know which way to turn and how to ascend the hurdle and how to get over it, and when we're done you're going to be facing east again.

So you can assume, after this method is done, that you're facing east again and you're at the bottom of the world in the next avenue after the hurdle. So it makes clear, right, because we don't know what-all stuff is going on inside ascend hurdle and descend hurdle, but a programmer now doesn't need to trace through the execution of your program to figure out what's going on.

This tells him what needs to be true before and what needs to be true after, and it helps you with debugging. Because if you're debugging your program and you call this jump hurdle method and you haven't satisfied the precondition, it's probably not gonna work right and that allows you to work backward to figure out what you need to do to fix things before you even call this method, okay?

So any questions about that?

So as we go further down, you'll notice – and this is all in your handout – all of the methods have pre-conditions and post-conditions that are commented in here. This is a good habit to get into, at least for your Karel programs, okay? Even if you don't explicitly put pre- and post-conditions, you should at least have some comment for every method, explaining what that method does. So that's the level of commenting that we actually want you to have.

Notice it's also perfectly fine to have very short, like you know, three-line or in some cases even one-line methods is perfectly fine, because it helps – like here's a two-line method over here – it just helps break the program down, okay?

And that's a process we refer to as decomposition. And so what's decomposition all about, right? We need to think of this in sort of more concrete terms that you may understand from your daily life. So besides reading the bottle of shampoo, another thing that we could talk about is what you did this morning.

So if I were to say hey, what did you do before you got up for your – or before you went to your first class this morning? How did you get yourself ready this morning? What did you do? Anyone wanna volunteer what they did? I know it might be a little scary for some of you. Um-hm? What did you do this morning? Use the microphone, please. Pardon?

Student: Basically I basically rolled out of bed, and then –

Instructor (Mehran Sahami):All right, so you woke up. Wow, did you even hit the snooze bar at all?

Student: Twice.

Instructor (Mehran Sahami):Oh, good times. That's always good.

Student: That's why I set it early.

Instructor (Mehran Sahami):She was good. I had a – my old roommate in college actually would play the snooze bar game, no joke, for two and a half hours. And I would be sitting there, I'd be in the room, like, I'd be awake, staring at the ceiling, and [inaudible] I'd know his snooze bar, and I'd be like Joe, why don't you just not set your alarm? And he'd be like no, man, I gotta set my alarm.

And every day – anyway, I'm glad it's only twice for you. You can tell there's just, like, you know, like I got this little twitch from, like, 15 years ago from the snooze bar. But you woke up, and then what did you do?

Student: Okay, and then I brushed my teeth.

Instructor (Mehran Sahami):Okay, you brushed teeth. What else?

Student: Go to the bathroom.

Instructor (Mehran Sahami):Bathroom. I won't ask you for any more detail on that.

Student: Yeah.

Instructor (Mehran Sahami):What else.

Student: That'd be awkward.

Instructor (Mehran Sahami):Shower? Was shower in there, maybe?

Student:I shower later in the day.

Instructor (Mehran Sahami):Oh, okay, that's fine.

Student: After [inaudible].

Instructor (Mehran Sahami):I'm just gonna stick shower in here.

Student: Okay.

Instructor (Mehran Sahami):Just for good measure. But it's perfectly fine whenever you wanna take them. What else?

Student: And then clothes, [inaudible] clothes.

Instructor (Mehran Sahami):Clothes. And then presumably you probably went out of your room or ate breakfast or – did you get up for breakfast?

Student: No.

Instructor (Mehran Sahami):Yeah, I know, it's brutally early, isn't it?

Student: Yeah.

Instructor (Mehran Sahami):Like, I think, like, I ate breakfast like once the whole time I was an undergrad, and it's because, like, I just stayed up the night before and I was, like, oh my god, there's a thing called breakfast, and I can actually go to it, so I went.

But so this is an interesting thing, you give us sort of a breakdown of what you do. Thank you very much, what's your name, by the way?

Student: Jasmine.

Instructor (Mehran Sahami):Thanks, Jasmine. I'll try to see if I can get to you, but – denied every time. So here's, you know, a standard set of things that we might think about doing. Now it turns out this is a very good list of things that you might want to consider doing. Good hygiene for the morning. But in fact, when you think about something like brushing your teeth, brushing your teeth – we all understand what brushing your teeth means.

But if there's someone, like when I, you know, try to get my 1.5-year-old son to brush his teeth, I say hey, William, you need to brush your teeth, he has no idea what I'm talking about. Because brushing teeth for him involves getting the toothbrush – brush – putting toothpaste on it, and sort of moving it on your teeth, right?

So there's a level at which I need to break this down. As a matter of fact, moving it on your teeth, if you've ever talked to a dentist, this is a whole involved process by itself. Like what angle you put it at and how much you move it and how much time [inaudible]. It's like whoa, man, I never thought there was that much to brushing your teeth, but evidently, there is.

And so what happens is we start with a high-level description, and then each one of those steps in the high-level description we break down. And we keep breaking it down until we get to a level of detail, which is sufficient for us to be able to understand minutely. Or if we're doing this on the computer we get to a level of detail at which the computer understands it directly.

What that means is we eventually get to a level of detail where it turns into things like put beeper or move or while something is true, right? So when we get to that level which we call primitives, that's when this process eventually stops. But this is essentially a process that's called stepwise refinement.

Because what we're doing at each step in the process is we're coming up with some steps, and if those steps are not the primitives that we understand we need to refine them into more steps and refine them into more steps until we eventually get to the primitives, okay?

So we start at the high level and we break things down. And the notion of breaking things down into their sub-pieces is something that we refer to as decomposition. Decomposition. Which just means taking something big and breaking it down into smaller pieces, which is something you should do with your program. You wanna break them down into smaller pieces, okay?

And this whole process, the whole process itself is something we refer to as top-down design. And you'll hear this phrase referred to a lot, and what top-down design is is you start at the highest level, the top, and you go down from there. And that's to be contrasted with something that's referred to as bottom-up design.

And what bottom-up design says is hey, I need to go do something in the world. Well, let me start at the primitives. I need to make Karel, you know, go to the end of the wall and put down beepers. What's the first thing I need to do? I need to put a beeper, and then I need to move. And it starts with the low-level stuff and this is actually the place most programmers, when they start out, this is what they begin doing is bottom-up design.

People have actually done psychological studies. It takes about 100 hours of programming proficiency. In the average case, I think everyone at Stanford's above average so it'll take you less, to go from thinking bottom-up to thinking top-down.

Guess what? It's a 10-week quarter. We might do about 100 hours of programming in here. Hopefully by the end of this class, everyone's doing top-down design. But if you can start from the very beginning doing top-down design, you're golden.

In some cases this makes sense, but a lot of times this just makes your life easier, to think about the most abstract level and break down from there. And so if you think about what we did when we actually did the steeplechase program, this was top-down design, right?

When we actually wrote this program together, when we went through it together, we didn’t talk about Karel doing individual moves to begin with. We said hey, we need to run a steeplechase, and that either involves doing the move or jumping a hurdle. At this point, we've done enough step-wise refinement that we've gotten to the level of a primitive. We don't need to go any further because Karel immediately understands move.

Jump hurdle, Karel doesn’t understand. So once we've written this, we need to break it down into lower-level steps and say what does jump hurdle involve? And that's where we actually go down and define jump hurdle, and guess what? When we define jump hurdle, there might be some other things like ascending and descending a hurdle, which are the next steps down in the process that we need to define, and we just keep doing this, okay?

So with that said, let's actually do top-down design together. Let's start another program from a blank slate and see if we can do top-down design together to see how this process might actually go for writing code, okay?

So the problem that we wanna try to solve is we wanna – oh, and don't look at the code – we wanna try to get – and this is kind of a funky problem – it's Karel does math. If you didn't think Karel could do math, in fact Karel can do math. Karel's a lot smarter than we sometimes give him credit for.

So this is a little problem called double beepers. So I don't know if you can see the number here, but there was a five here, which means there was five beepers on this corner. And when this program is done running, what we want to have happen is Karel is guaranteed to start in position 1-1 and there's guaranteed to be a pile of beepers in front of him and a space after him, okay?

When he's done, he should return back to the same position and the pile of beepers should have exactly twice as many beepers on it as it had before, which means Karel begins with a beeper bag that's full of as many beepers as he needs, so it has infinite beepers in there so we can double the number of beepers.

So if I speed up the program and just run it, you can see the final state. So let me just crank up the speed here and start the program, and it looks like nothing happened, but in fact this number changed from a five to a 10. It doubled the number of beepers.

So two things should be going off in your head right now – one is thinking hey, that's kind of slick. How do I do that? What's the recipe for doing that? And the recipe for doing that, the sort of general approach you wanna take, is something that we call an algorithm, okay?

So the notion of an algorithm – and this is just a fancy computer science term for an approach to something – actually comes from – anywhere know where the word algorithm comes from? Nineteenth century Persian mathematician named, if I can pronounce it right, Al-Khwarizmi. Al-Khwarizmi, which sounds like algorithm. There you go.

Yeah, add 1100 years to that pronunciation and it sounds like algorithm, all right? Because this is from the 19th century. It's your basic approach. As a matter of fact, there was a band at Stanford years and years ago called the Algorithms, where this was, like, rhythm, like hey, I got rhythm. Yeah. I won't say how good they were. Not very.

All right. So you wanna think about your general approach and then you turn your general approach into thinking in terms of step-wise refinement. How are you gonna solve this problem from the top down?

So let's actually start with a blank slate. Let's close this puppy out and we'll start with a blank slate, which we'll call, oh, Our Double Beepers. So I just gave you the boilerplate stuff to begin with. There's a little comment up at the top that says file double beepers. That should be called Our Double Beepers, actually, and Karel doubles the number of beepers on the corner directly in front of him in the world, he returns to his original position and orientation.

So we have the public class stuff for Our Double Beepers, we have an [inaudible] super Karel. Now we're gonna run the run method, okay? So what do we wanna do to get Karel to double the number of beepers at a high level? What might we think about doing this? Hm. Um-hm?

Student:[Inaudible.]

Instructor (Mehran Sahami):We need to figure out how many beepers there are. In order to do that, maybe we should move to the pile of beepers, right? It's one avenue in front of us. So we do a move. And now here's the part that almost seems like magic. I'm going to say move, hey, you're on this pile of beepers now. Why don't you, hey, double the beepers in the pile? And you're like, uh, [inaudible] can I do that? Sure, why not, right? You're gonna write the program, you can do whatever you want.

So I would like to say move, double the beepers in the pile, and then actually what I wanna do is not move forward again but I wanna move backward, right? I wanna move – almost like Karel could go in reverse, that'd be kind of cool, to keep the same orientation but goes back.

And you're like but [inaudible] you didn't tell me about move backward. Can super Karel do move backward? And I'm like, no. And you're like oh, okay, well, all right, well, I guess I need to write some of these, right? Let's actually do move backward, because that's the easier one.

If I'm sitting on a particular corner and I'm facing east, what do I need to do to move backward one step and face the same direction? Yeah, I need – so let me define move backward here. Move backward. I need to turn around, which is a primitive that I can use. Then what do I do? Move. And then what do I do? I turn around again so I have the same orientation when I'm done as when I started. And now, guess what? Move backward is something I understand how to do, because I've broken it all down into primitives.

I don't need to go any further in terms of the refinement for move backwards. But now there's the magical part, right? There's double beepers in pile, and that's something that might be a little bit more involved. So I come along and say okay, I need to define what double beepers in pile is. Double beepers in pile. At this point, I need to think about my algorithm. What's going to be my approach for doubling the number of beepers? What's a way that I could possibly think about that problem? Because Karel can't count, right? There is no way for Karel to just say hey, counting how many beepers are on this corner.

What's one way he might approach it? Um-hm?

Student:[Inaudible.]

Instructor (Mehran Sahami):That would be nice if we had a counter, but we don't have a counter, right? So Karel can't count. How is he potentially gonna do it? Um-hm, all the way in the back. Use your microphone, please.

Student: So he needs to pick up one beeper at a time and put it on the empty corner, and when he puts it on an empty corner, put two beepers for that one he puts down?

Instructor (Mehran Sahami):Yeah. So why don't we – I know it's not gonna happen. Yeah, I know, everyone's just starting to cower now. I do realize that, I've been teaching for about 15 years and my shot has never improved, so you're just, you know, at the random candy mercy.

Since we can't count, we need to do something beeper by beeper. So wouldn't it be interesting if we said hey, you're on that pile of beepers, pick up one beeper off the pile and take a step somewhere else and put two beepers down, and keep doing that over and over.

Pick up a beeper, put two down. And eventually, you'll exhaust everything on this pile over here but you'll have twice as many on the next pile beside you. And then take that pile of twice as many beepers and move them all back to the place where you started, and then you're done.

That's the algorithm. Let's write the code to do that. And you're like okay, [inaudible] let's write the code to do that. And I'm like yeah, rock on, this is gonna be easy, right? What we're gonna do is we're just gonna say you're on a pile of beepers. As long as there are still beepers for you to pick up – so I'm gonna have a while loop here – while beepers present.

Well, what I want you to do is pick up a beeper, yeah, you know how to do that – life is good. Beeper, I always misspell beeper, I don't know why. I want you to put two beepers next door – put two beepers next door – and you're like oh, this is always the part, right, where you're thinking bottom-up design, you get uncomfortable calling methods that don't exist, right?

You're just like oh, but that doesn't exist, right? And I'm like, it's cool, man, we're gonna write it. And then we hold hands and we sing Kumbaya and everything's good because at the end, Karel's gonna know what to do. So sometimes, you just write methods and they don't exist, and it's fine because you're the one that's gonna be writing them. So eventually, they'll exist, right?

It's not like you're sitting there, like, lighting candles, making some magic incantation, hoping the code just shows up. And you're like, is it there yet? No, it's not there yet, right? You're gonna write it. I had a friend once who actually thought that's how we did computer science. I didn't bother to tell him that that wasn't true.

But, you know, think about it. So we pick a beeper – yeah, I know, it's kind of mean, but that's all right. He does government now. We pick up two beepers and guess what? After we do that, when that's done we will have picked up all the beepers from that pile that we're standing on, and we would have put two next door for every beeper we picked up.

So when we're done with this while loop, we know that the thing that's true is that there's no more beepers on that pile because beepers present is no longer true.

So I have some pile next to me, say on the next corner, that has twice as many beepers on it. And so what do I wanna do? I just wanna move the pile next door back. And that's my function. You're like, oh, really? I can do that? Sure, why not? So what do we need to write now? Well, we need to know how to put two beepers next door.

So let's go ahead and write that. [Inaudible] private void, put two beepers next door, right? Now if I'm standing on a particular corner and I'm facing east, let's say I wanna put two beepers on the space in front of me and come back to where I am still facing east, so I'm left in the same position I was when I started, how do I do that?

I move. I need to put two beepers so I could put beeper, put beeper. And if you really wanted to you could put a four-loop here and say hey, [inaudible] you're doing something twice, ooh, ooh, can I do a four-loop and do four I equal into I equals, zero, I less than two, I++, put beeper? Sure, right? There's multiple ways of writing a program. I'm just going to put two put beepers because it's two lines. The four-loop would have been two lines too.

Now what do I do? Ah, yeah, rock on. I move backward, right? Which allows me to move back one step and keep the same orientation. So I end up, my post-condition is essentially the same as my pre-condition, except I've moved one beeper over to the – or I put two beepers onto the next spot.

And move backward, you're like hey, I already wrote that. So everything now is in terms of instructions that Karel will understand, so at this point I don't need to go any further. So I know how to put two beepers next door, so the only thing left to do in this program is to move the pile next door back to where I am now, okay?

So what that means is how do I move that pile back to where I am? So I'll call this, for lack of – well, I know what the function name is, I'll just – or the method name. I'll just steal it from up here, because sometimes you just get tired of typing.

All right, so I'll move the pile next door back. Well, the place that I call this, right, I call this up here. I'm not actually on the pile next door when I called it, right? My pre-condition is I am on the avenue before the pile. I'm facing the pile, but I'm not on the pile. So the first thing I need to do in terms of moving the pile next door back is I need to move over to where that pile is.

So now I'm standing on that pile of twice as many beepers, and I wanna move all the beepers back one. So what could I do? I could pick beeper, but there's something I wanna do multiple times. I need to have a wild loop, right? And I'm gonna do something – what condition do I wanna test, right? If beeper is present, right?

So as long as there's beepers present there's still more work for me to do because there's still more beepers I need to move back. So while there's beepers present, hey, why don't I move one beeper back? Move one beeper back. All right? Again, I haven't wrote that yet, that's fine.

But when this loop is done, what I know is I've exhausted all of the beepers that are on this pile because every time there was beeper present I picked it up, I moved it back one spot, and hopefully that method brought me back, should bring me back to the place that I'm standing right now, and I just keep doing this until I've transferred all the beepers.

So after I've transferred all the beepers, what do I need to do to satisfy my post-condition, which is I wanna go back to the place I just transferred all the beepers? I need to move backward. Because all the beepers were transferred one avenue back. So I just move backward to go back one avenue, okay?

Now I'm almost done. I'm almost feeling good, but not quite. I need to do the move one beeper back. And at this point, I've gotten to such a simple level that maybe I could just do this. So if I'm standing on a pile of beepers facing east, how do I move one beeper back behind me, put it down, and come back to the spot that I'm at? Well, first let me pick a beeper, right? Because I need to actually move the beeper.

I could turn around first, that's perfectly fine. There's multiple ways I can do things, but I'm gonna pick the beeper first. Then what do I do? Yeah, rather than turning around, let me just use move backward. That'll take me back one spot. Then I put beeper. And I move to move back to the spot that I'm at, right?

Now I'm all in terms of things, what I refer to as primitives, right? Or instructions that I've previously defined. And there's nothing more for me to define, because every time I define some new method and I said hey, I'm just gonna call that, I went and wrote it and I just kept doing that over and over until I'd written everything I needed to do. Um-hm?

Student: Wouldn't you have to move backward twice [inaudible]?

Instructor (Mehran Sahami):On move pile next door back? No, because move one beeper back, I pick up a beeper, I move backward, I drop it off, and I move forward. So I'm on the pile that has all the beepers.

Student:[Inaudible.]

Instructor (Mehran Sahami):Yeah, and that's what I did up here, right? So after I doubled the beeper pile, then I moved backwards. So yeah, that's a good point, but you always wanna remember what other things are gonna happen after you execute all that, okay?

And so now the question comes – ah?

Student:[Inaudible.]

Instructor (Mehran Sahami):That's a good question. The only problem with doing that is Karel starts off with an infinite number of beepers in his bag, right? So if I pick up all five beepers at the corner that I'm at and then I take a step back, or I pick up all five beepers and now I wanna put them all down and double them, I don't know that there were five anymore, because after I pick up the five, I look in my beeper bag and I'm like oh, man, I had infinitely many beepers. I've got infinity plus five, and that looks like infinity, right?

As a matter of fact, if you take CS103, which I encourage you to do because I teach it in the winter eventually, we talk about infinities, and we're like oh, infinities, and it all comes back to Karel. No, it actually doesn't come back to Karel at all. But infinity plus any number is within reason – yeah, if you're a number theorist, you're gonna argue about that – is still infinity. So that's why Karel can't count.

If he had an empty beeper bag we could put all five in it and look inside and then we could put down five. But now we don't have five more beepers to double the number of beepers, which is why Karel had to start with infinitely many beepers, so no matter how many were in the initial pile he would always have enough beepers to double it. But that's a good point.

So let's actually go ahead and run our program now. Let's save, and we'll go ahead and run Our Double Beepers. Doo, doo, doo, doo, doo. Operate just to make sure it actually works. And we'll sort of do it at slow speed. So there's our world, let's start the program. There's Karel, and you can see he's going [inaudible] beeper – whoa.

He just didn't want me to explain that program at all. All right, Karel, I'll remember this. So for every beeper he picked up, he put two here, and now he's just transferring them all back. Go, Karel. And so now he's done with 10, and you'll notice he went back here after 10. Why? Because he needed to check to see if there were any more here, and there weren't, so he moved back and then he was done.

So we just doubled the number of beepers. And this program, actually, notice the initial state of the world is now exactly the same as the ending state of the world, except we doubled the number of beepers. So we could actually run this again if we wanted to – let me speed it up a little bit – and we'll go from 10 beepers to 20 beepers. And there is Karel sort of doing his thing at faster speed, but you can see there's 20 beepers over there, now they all start coming back and saying, you'll be like, hey, [inaudible], how many times can I run it? Can I just run it over and over and I'll get, like, you know, 40 beepers and 80 beepers and 160 beepers?

You just can't. I think eventually the program dies when you reach, like, 100 and some-odd thousand beepers, because it's just like no more, man. No more, all right?

But for all programs you write in this class, if you ever have more than 100,000 beepers on a corner, there's probably something wrong, so you don't need to worry about it, okay?

Now, that's – so congratulations, right? We just all wrote this together, and you just made Karel do math. As a matter of fact, Karel, in terms of raw computational power, can solve any arithmetic problem that any other program can. It just may take him a lot of beepers to do it. But theoretically, you can prove that. We're not gonna do it in this class, but if you're just kind of – you're like can Karel compute, like, square root and logarithm and he can integrate? Yeah, he can. It's just a whole lot of beepers, all right?

So I could show you another program. Here's another little program I happened to write in the office. This is a program that happens to be called Do Your Thing, okay? And I'll show you the full program, I'll scroll it down here so you can see the full body of the program. And all I will ask you, what does this program do? Doo, doo, doo, doo, doo, doo, doo, doo, doo, doo, doo, doo, doo, doo.

This does example the same thing the code you just wrote did, okay? This is really bad software engineering, right? There is no decomposition, there's no comments, there's no indentation, there's no notion of what's going on in this program. If you looked at this program and I said, hey, instead of doubling the number of beepers, I want you to say triple the number of beepers, you'd have to go through this thing, and you might, because you just wrote it, say hey, I know that there's two put beepers in a row, so I put another put beeper in there and it'll probably work, right?

Whereas say you're a software engineer and you just got your first job, and you can look at either that code, or you can come over here and look at the double beepers that's actually in your assignment, that has all the comments, like, hey, this doubles the number of beepers.

And over here, guess what? This is the version that's in your handout. This is all the same code we just wrote except now with comments. Like for every beeper on the current corner, Karel places two beepers on the corner immediately ahead of him, and this says place two beepers on corner one avenue ahead.

And you say hey, if I wanna triple the number of beepers, I change the comment from two to three and put another put beeper here, and I know exactly where it goes and it's easy to understand.

That's why software engineering's important, because that kind of modification to code happens far more often than writing code from scratch – about 10 times more often. So if the code is bad to begin with, it gets 10 times more costly to go and make fixes to it, and that's something that's sort of well documented. Some people actually argue that the number is higher.

But that's why it's important that when you write your program the first time, or subsequent revisions you make to it, you have good style because that's just way more

important than just getting the program working. And I've seen people actually turn in programs called Do Your Thing. That's where I actually got the name, was I actually knew someone once who write a program called Do Your Thing.

And I was, like, you've gotta be kidding, right? And it was just like this, right? It was just a whole bunch of straight-line code, no decomposition. They were like, well, it works. And I'm like yeah, but that's not what software engineering is about. And we got into this – yeah, I won't call it an argument, I'll call it a discussion. And they got really adamant, because they were, like, but it works, so I must get an A.

And I was like, you know, someday if you're teaching this class and you're willing to accept code like this, you can give yourself an A. I'm not, because this is just – anyone who sees this, not only in this class but outside of this class, will be horrendous code, and I don't want any people coming out of this class to go out into the industry and be like, hey, yeah, this is what [inaudible] told me to do, that's why I'm writing it this way, right?

Because not only will they be, like, no, I'm sorry, you're fired, but find that clown [inaudible] and fire him, too, right? So don't write code like this.

Is there any questions about this? Um-hm?

Student:[Inaudible.]

Instructor (Mehran Sahami):And let me bring up the good code again, just so you don't get, you know, think like that other code is the good thing to do.

Student: When you're naming the private [inaudible] methods, do you – should the first letter be capitalized, or not?

Instructor (Mehran Sahami):Ah, good question. So in terms of – someone asked before, all the words or methods that we define are all case sensitive. You'll notice that sometimes we use a naming convention. I'm actually bad because I mixed up the naming convention a little bit where the first name of the methods, we can either have a lower-case character or an upper-case character, it's up to you how you want to do it. Just be consistent.

Sometimes, this notation is referred to as camel case, and the reason why it's referred to as camel case, the first letter of every word that actually appears sort of all stuck together is upper cased, and it's kind of like a camel with humps, because every once in a while you get this capital letter with a hump. So that when you – if you hear someone refer to oh, we use capital case or camel case notation, that's what they mean. It's all strung together as one word with the first letter of each word capitalized.

Sometimes what people will do is they'll actually put in underscores. You can put in underscores between words and that's perfectly fine, too. When we get into Java, I'll actually spent probably like half of a lecture talking about the different conventions for

all kinds of things, because when we get into the Java world there's actually a bunch of things besides just method names you need to worry about, and we use all different sorts of conventions for them.

For right now, I'd just say pick one convention and stick with it, and the convention you can use is, you know, the two most common ones are either camel case all the way through or camel case except for the first character, which is lower case. Any other questions? Um-hm?

Student:[Inaudible.] Instructor:

Yeah, so it starts at run and every time it encounters some method that it doesn't know is a primitive method, it just says hey, did you define it somewhere? And it goes to that definition. And guess what? If it encounters one that you didn't define, that's an error. And usually it'll tell you that when you try to run the program. It'll tell you that this thing doesn't actually exist.

All righty? Any other questions about – um-hm?

Student:[Inaudible] as to what order you should put your [inaudible]?

Instructor (Mehran Sahami):No, you can put them in whatever order you want. Usually I just put them in the order as I need to define them, right? So just like we did in class, I just keep going down, which means run starts at the top, and everything comes underneath it.

A couple things related to that. One question that always comes up is how do I know how much to decompose, right? Like aren't there different ways I could decompose? And in fact, yeah, there are multiple different ways you can decompose. So let me just give you some sort of quickie guidelines about how you might think about decomposing, okay?

In terms of decomposition, the thing that you really wanna think about is every one of your methods should basically solve one problem, okay? The only thing is that that problem can be at different levels. So when you think about decomposition, what are some good guidelines?

One guideline is think about each method solving one problem, okay? That problem can be high-level, like double the number of beepers . That's a high-level single problem. When you get into doubling the number of beepers, you don't just wanna say oh, double the number of beepers again, because you're already at that level of decomposition. You need to go one step down and say, what are the things that make sense in terms of solving one problem at a time to break it down?

Now a lot of times, people wonder but [inaudible], what does that mean in terms of lines of code? Because sometimes it's different for me, at least conceptually, to think about one problem. And here's just a rough guideline. It's not a guideline that always holds, but a

rough guideline is most of your methods will probably be somewhere between one and 15 lines long, okay?

And you're like, one line? Yeah, that's perfectly fine. It's perfectly fine for you to have a method that's just one line long that calls some other method, and you might say what's the purpose of that? The purpose of that is giving that thing that you call a different name, right?

So remember when we did ascend hurdle, which basically brought you all the way down until you were facing a wall, and then it did one last turn at the end to get you facing the right direction? But basically all it was doing was move to wall. We could have actually defined ascend – or sorry, descend hurdle to be move to wall and then do the turn afterwards.

That would have been perfectly fine, too, and then descend hurdle would have just been a one-line method. But it would have changed the concept, the way the programmer thinks about it, from being hey, I'm moving to a wall to be hey, I'm descending a hurdle.

So that's another thing that kind of comes up in terms of the methods is you wanna think about them as having good names. The names should describe what the method is actually doing, what problem it's solving, because you want your programs to read like good English, right?

When someone – again, programs are written for people. Computers execute them, but they're really written for people to be able to go and look at and modify, so good names explains what the program actually does, and every one of the methods that you have in terms of decomposition should also have some comments associated with it that explains either at some more detailed level what the method is doing, or if the method is very short and it's fairly self-explanatory, pre-conditions and post-conditions are sometimes a good idea as well.

So think about these things for your assignment. When you're actually writing the code for your assignment, you wanna think about the sort of decomposition using the step-wise refinement process and make sure to comment – both a comment for your whole program and a comment for each method. Um-hm, question?

Student: The book mentioned procedural programming and decomposition under that. Is that what we're doing right now, or are we doing the –

Instructor (Mehran Sahami):Well, there's a difference between procedural and object-oriented programming. Decomposition actually applies to both of them, but what we are doing is decomposition whether it's procedural or object-oriented. They're just two different – come up to me after class and I'll set you up.

So I did wanna show you one more thing, and the one thing I wanna show you is yet another interesting problem, which is called Clean-Up Karel. And what Clean-Up Karel

was basically gonna do is Karel starts off, it's kind of like after – we should call it After-Fraternity Party Karel. It could be After-Sorority Party Karle, too. I don't wanna make any kind of claims.

But basically what ends up happening is that Karel kind of comes into his world and it's a mess. He's just like what happened? When last I was in this world I just doubled the number of beepers and shown you I can do math. And now you bring me into this world and there's just crap everywhere, right?

And we could call him Angry Karel, too, right? He's the one who's gotta live here. And there's, like, cups strewn all around and they're just on random corners, and the things that you know about Karel in this world is he starts at location 1-1 and every corner of the world can have at most one beeper on it. It means it can either be empty, like some of these corners, or have one beeper on them.

And what Karel needs to do is he needs to go through his whole world and clean it up, which means he needs to gather up all the beepers that are out there. And he can end up wherever he wants to end up, okay?

Now when we kind of think about a problem like this, one thing we might be inclined to do is you say hey, if there's a whole world that I wanna clean up, I might wanna break this problem down into a series of steps. One step – and something that is, like, oh, I can go and write it right now because I know how to write it is maybe to clean up one row at a time, right?

Because I probably know how to clean up one row, I probably just wanna keep moving until I get to a wall, and if every time I move there happens to be a beeper there, just pick it up. And so you get all excited and you go and you write the code to do that, and so you start off by having something like this – let me just focus on the top code up here – right, which is hey, I wanna clean a row.

This thing just does not wanna stay on me. It's like no, you will not talk into the microphone. So cleaning up a row I could say hey, you know, [inaudible] told me about the OBOB too, he told me about the off-by-one bug. So before I even do my first move, I'm gonna see if there's a beeper where I'm standing now, and if there is, I'm gonna pick it up.

And then I'll check if my front is clear, and if it is I know I can move and I should check to see if there's a beeper there, and I'll just keep doing this over and over until I've picked up all the beepers.

And that's a perfectly fine thing to do, right? So now you've written some method called clean row that can clean up a whole row, and you know you've sort of dealt with the off-by-one bug because you sort of do one extra thing outside of your loop, either before or after. In this case, you happened to do it before just so you can see a different kind of formulation of it than doing it after, and that's great.

The only thing is, now you need to figure out where clean row is gonna go in your program, okay? This is an example of sort of doing more, the notion of bottom-up as opposed to top-down design. It's easy to get excited, to say oh, there's something I know how to do, and just go write it. And if you happen to write the right thing that fits cleanly into the rest of your program, that's great. But sometimes it doesn't, and you might need to go back and modify it, so keep that in mind.

But if you do this, that's perfectly okay. It turns out that in this case, that's a fine thing to do, to clean one row at a time. And so if we wanna think about how that fits into the larger program, here's kind of a funky thing. And let me just explain the algorithm to you, because the algorithm here is somewhat related to an algorithm, for example, on one of your programming assignments.

I'm gonna clean up the first row, because I know that first row I always need to clean. So I'll just go clean up the first row, and now I wanna check to see am I done. Well, if I've just cleaned up a row, the only way I know I'm done is if the ceiling's above me, right? I've reached the end of my world.

And so I'm facing east, so I check my left to see if there's a ceiling up there. And if there's a ceiling up there, then I'm done. But if there isn't a ceiling up there, then I need to do more work. So while my left is clear, right, there's no ceiling up there, I'm gonna reposition for it to clean up a row to the west, because I just cleaned up a row sort of moving to the east – am I facing your east? Yeah.

I just cleaned up a row to the east, and I wanna say hey, no ceiling here, that means there's another row to clean up here. So what I wanna do is move up one row to be able to now head west and clean up the next row, because I'm gonna go back and forth, that's my algorithm.

So I'm just gonna write some function that says reposition for a row to the west, and go ahead and clean that row. Now what do I need to do again? I need to check my roof. But now my roof, because I'm not facing that way, isn't on my left anymore. I need to check my right to actually check my roof, so I say if my right is clear, then reposition for a row to the east. You're gonna go back the other way and clean that row, okay?

Now there's this else here, and we'll get to the else in just a second, but just to figure out what reposition to west and east are, they're actually very simple. I'll just show you those down here. Reposition for row to west means I just finished a row to the east, how do I get up to harvest the next row or to clean up the next row?

I do one turn left that gets me to face north, I move up one step so now imagine I've floated up one row, and then I do another turn left and I'm facing like this, and now I'm ready to plow over the other direction. And I do sort of the opposite for repositioning for a row to the east. So I'm standing here, I do a turn right, I look up, I move up one row, I do another turn right, and now I'm ready to go back that way.

So these are very simple, right? They're in terms of primitives. So this is the whole program. You've just seen the whole program now. The only question that comes up is what happens with this little turn around here. What's going on there?

So you said hey, [inaudible] you just finished cleaning a row and you just checked to see with your right hand, actually, if my right is clear, if there was a ceiling above you. And if there was a ceiling above you, right? Or if there wasn't a ceiling above you, you went ahead and got ready to clean another row. But if there was a ceiling above you, what do you wanna do?

And you say hey, if there was a ceiling above me, I wanna stop – just stop. Is there some way to get Karel to stop? For the love of god, Karel, stop. No, Karel's just like no, man, I'm going. I'm going. You're like, there's no way to just say stop, Karel? How do you get Karel to stop? You need him to finish executing what he's executing. What is he executing, right? If we just got here and we cleaned up a row and I checked my right and there's a ceiling there, if I don't do anything, what's Karel gonna do?

He's gonna go back around and say hey, is your left clear? Yeah, because my left is below me, it's clear. And it's gonna go and try to clean more, even though I just finished the last row.

So what I actually do is when my right is blocked over here I say hey, you know what, Karel? To get you to think that you're done, I'm gonna have you turn around. So now when you check your left, the ceiling is gonna be there and you're gonna break out of that loop.

So you wanna think about the conditions on your loop to actually get this to happen. So in our final 10 seconds together, I'll just run this to show you that it actually works. Doo, doo, doo. And you're like actually, [inaudible] you're in, like, our, you know, final minus four minutes together. I know, I'm sorry. Education's one of those odd things that the more – you get more of it for paying the same amount, and somehow you feel like you were cheated.

So let's start Karel. Look, if I let you out 10 minutes early, everyone would be like oh, yeah, rock on, but if I gave you a car that only had three doors on it, you'd be like what's up with that? Strange, how that works.

And there's Karel, just cleaned up his world. All righty, so I will see you on Monday. Have a good weekend.

[End of Audio]

Duration: 53 minutes