Transcript

Lecture 11: Programming

Hour 1

(00:00:00)

DAVID MALAN: Welcome to Computer Science E-1. My name is David Malan. This is Lecture 11, "Programming." We are nearing the end, and as we do, the topics become a little more of a segue to perhaps additional courses in computer science.

Recall, after Exam 1, we passed out that handout "Life After E-1," which offers you some pointers in which directions you might go after this course. Programming is certainly among them. And programming—even though you won't exit this course a programmer, per se—by the end of tonight, you should have a sense of what programmers do, what programming is all about. And, daresay, a better understanding of why some things go wrong in your own computers, why your computer crashes, why your computer hangs, ultimately as a function of mistakes that humans have made.

Speaking of mistakes, what is correct about this advertisement? It's terribly small, I know. So you can go out on a limb and take a guess. What did they get right in Bed, Bath & Beyond's winter issue of their Christmas specials, and such?

STUDENT: (inaudible response)

DAVID MALAN: Yeah, they finally got the URL right: www.bedbathandbeyond. Recall on Exam 1, we had a photograph that I had taken, in all my photographic skills, of a window display of Bed, Bath, & Beyond, and it was erroneous in that it had an ampersand, and not the word "and".

So some crazy demos today. We have a phonebook, it would seem. We have a small child. We have some digital demos. So let's dive right in.

So programming is ultimately about implementing algorithms, and solving problems. An algorithm is just a procedure, it's a way of doing things. And it's in algorithms that a programmer gets to exercise a bit of discretion, a bit of intelligence; gets to think about the design of a program.

And perhaps before we dive in too deep, let's just put some words on this. What is a program? Should be a softball, right? What's a program? Yeah?

STUDENT: (inaudible response)

DAVID MALAN: Good. So it's just a set of instructions to a computer. And we interact with programs by way of double-clicking them usually. But what is that Word.exe on your hard drive? Well, it's just a collection of bits. It's a file. But those bits are laid out in a way that the specific CPU that's inside your computer understands. And, at the end of the day, we've said that those bits simply comprise instructions: Do some addition. Do some subtraction. Do some printing.

But ultimately a program is just a set of instructions. So a programmer, therefore, is just a person who actually writes those instructions, and writes them in a whole bunch of different languages, some of which we will glimpse tonight.

A programmer is a software developer; might be a software engineer. A lot of these terms are synonyms, for the most part, depending on where one works and what one calls him- or herself.

So, let's consider a problem. Here is a phonebook with roughly 1,000 pages in it. If I wanted to look up a phone number in this phonebook, and I knew the name of the company, how would I likely go about solving that problem? Physically, literally, what do I do?

Here's the phonebook.

STUDENT: (inaudible response)

DAVID MALAN: Turn to the business section. Let's assume the whole thing's the business section. It's pretty much all yellow pages. All right?

STUDENT: (inaudible response)

DAVID MALAN: All right, so let's look alphabetically by name. Give me a company name that's likely to be in here.

STUDENT: (inaudible response)

DAVID MALAN: Leo's Diner, all right. So I'm going to open to the business directory, which starts with the *A*s. All right, *A* is clearly too early, so I'm going to turn the page, going to turn the page, I'm going to fast-forward, lest we get bored too soon tonight.

And eventually I'm going to get to the *L*s. Oh, just passed the Escort section, apparently, I noticed. Now we're in the Health section, the Hotel section, Insurance, Lawyers, so we're at least in the *L*s. A lot of lawyers.

STUDENT: (inaudible response)

DAVID MALAN: Oh, we're going to... We'll go to restaurants, right. So we're not... I'm an idiot is the takeaway here. There is not a section called "Leo". So why don't we go to the Restaurant section. Good takeaway for tonight.

All right, and so here in the Restaurant section, the section itself is alphabetized, presumably. And so now I can do a similar process, starting at the restaurants starting with *A*, and sort of work my way from left to right, through the alphabet, until I hit *L*, then Le, then Leo, and hopefully Leo's Diner would then be in this phonebook.

So if this phonebook has, say, 1,024 pages, which it actually happens to, roughly, how would you describe the running time, or the duration of that algorithm as I just executed it, which again, to be clear, was to open to the *A* section, and make a few mistakes, but then eventually work our way to the *R* section for Restaurants, and then home in on Leo's Diner?

How would you quantify the amount of steps that that would take if, for instance, we say that one step is the act of me turning one page?

STUDENT: (inaudible response)

DAVID MALAN: Good. So you'd have to know pretty much how many pages are there between the start of the book and the letter *R*. Well, in the worst case, maybe you wouldn't have asked me about Leo's Diner, but rather Zoe's, down on Mass. Ave. So starting with *Z*, that restaurant listing, if it's in here, is going to be at the end of the Restaurant section.

If you hadn't even asked me about a restaurant, but instead had asked, "Can you find me a local zoo?"—which might be a stretch, if there's even a zoo category—that certainly would be at the very end of this phonebook.

So if we wanted to just wave our hands slightly and say, you know what? In the very worst case, the number of steps that that algorithm, that procedure might take is maximally going to be what, in the case of this thousand-page phonebook?

STUDENT: (inaudible response)

DAVID MALAN: A thousand, right? It's a pretty naïve algorithm, but it's pretty darn correct. No matter what, no matter how long it takes, and no matter how bored we get, we know that if we did this brute-force approach, so to speak—starting at the beginning and then just looking alphabetically—we would eventually find our answer, if it's present.

But that algorithm, insofar as I was sort of walking a line, starting at one end and working my way to the other end, is what you might call an algorithm with a "linear running time," right?

If you were to plot—you think back to algebra, if you've got that *x-y* plot? Anything that's a straight line was a linear plot? And that essentially implied that, as you increase the number of pages in the phonebook, well, you increase the number of maximal steps that you might have in that algorithm.

So in the worst case, a thousand-page phonebook might take a thousand-some-odd steps. The question before a programmer, before a software developer, solving a similar problem, albeit with a computer, is, can I do better than that, can I do fundamentally better, right? It's one thing to just say, you know what? Spend an extra few hundred dollars and throw a 2 GHz CPU at it, instead of a 1 GHz CPU. But that's not really an interesting solution. Doesn't require any intellect, and doesn't fundamentally change the naïveté or the brilliance of your algorithm. All you're doing is throwing hardware at it.

And that's fine, but there's really no intelligence. There's nothing interesting about that approach.

So I put forth to you now the question, if I have this phonebook in the real world, and I wanted to solve that same problem—the looking up of a phone number—fundamentally more efficiently, could I do better than starting at the left and working my way all the way through to the right until I find the listing I'm looking for?

STUDENT: (inaudible response)

DAVID MALAN: I could do what?

STUDENT: (inaudible response)

DAVID MALAN: All right, so I could start in the middle. And, frankly, if you think about it—if you've even used the Yellow Pages in recent years, what with the Web and all—you would probably start roughly in the middle.

And then what would you do, looking, for instance, for restaurants?

STUDENT: (inaudible response)

DAVID MALAN: Sorry?

STUDENT: (inaudible response)

DAVID MALAN: So probably flip closer to the back because I've just opened to the *L* section, which is roughly in the middle. And you would expect that: 26 letters; we're roughly in the middle. So this is the Lamps page. So I clearly need to move to the right.

Well, let's perhaps apply that same approach, and dive into the middle of this right-hand side. But consider for a moment what I'm about to do. You told me to open to the middle of this book. Because I've opened to *L*s, and I'm looking for *R*, what can you tell me about this chunk of pages in my left hand?

STUDENT: (inaudible response)

DAVID MALAN: Forget it. And let's literally be dramatic here. Forget it! I've just chipped away at this problem, such that I've thrown half of it away, because my answer is clearly not in there if I'm looking for *R*s.

Well, if I apply this same logic—here's the *L* on my left, and the *Z*s are presumably on the right. Let's dive in roughly in the middle. All right, we're trying to do this... Well, that's great: Restaurants! And we're done! So that's okay.

So now let's say I'm on Restaurants, and I am currently on—I'm going to cheat slightly, because we're in the advertising section, not the main listings—and I am at the restaurants that begin with the letter *B*, currently. So where do I need to go next?

STUDENT: (inaudible response)

DAVID MALAN: All right, so I can again... That's the interesting part, right? It's clearly this way in the phonebook. But I get to now tear the phonebook in half yet again.

So in just two steps, what fraction is this remaining problem of the original problem?

It's a quarter, and that's a powerful thing. Because think about the original algorithm in the original phonebook. After just two steps, how many pages into the phonebook would I be?

STUDENT: (inaudible response)

DAVID MALAN: Yeah, on page 2, page 3, whatever. But still that leaves pretty much the whole damn phonebook. And this is a powerful thing. And what this algorithm suggests is something that's called a "logarithmic running time." If you think back to algebra, you might have recalled logarithmic graphs which, rather than looking like this, instead look something more like this, where they very, very, very gradually grow eventually. Whereas the linear plot keeps on going up.

(00:10:00)

And remember, if this is the number of pages on our *x*-axis, and this is the amount of time, *t*, that the whole algorithm is taking, which one in the long run is clearly the superior, the more efficient, the more intelligent algorithm?

Well, if your metric for success is speed, presumably the one that's logarithmic. And you might say that that's a log graph and this is a linear, just to slap the labels on it.

Well, consider the power of this, really. If we have a thousand-page phonebook, and we use a linear search algorithm—borrowing terms here from computer science—worst case? Remind me: How many steps might a linear algorithm take to look up a phone number in a thousand-page phonebook?

STUDENT: (inaudible response)

DAVID MALAN: A thousand steps. Easy, right? So you look on the *x*-axis to a thousand—maybe it's here; and a thousand on the *y*-axis ends up being here, right? A thousand pages, a thousand steps, a thousand units of time.

Well, now let's talk in binary, just because the math will work out a little cleaner. Suppose the phonebook actually has 1,024 pages, just so we have a nice power of two. And now I apply not my linear algorithm, but my much more clever logarithmic one, the one that does, if you recall, divide and conquer: split the problem in two, throw half away; split the remaining problem in two, throw the other half way.

That begs the question: If you've got 1,024 pages, how many times can you split that value in two; in two; in two, until you're left with just one page?

STUDENT: (inaudible response)

DAVID MALAN: Good E-1 answer: "Heck of a lot less!" Let's put a number on it, because this time it's impressive. How many times can you divide 1,024 by the value 2?

STUDENT: (inaudible response)

DAVID MALAN: Not quite. Too many.

STUDENT: (inaudible response)

DAVID MALAN: "Almost eight." That would give us 256.

Ten. Literally you can search a 1,024-page phonebook if you apply a logarithmic algorithm—and that makes it sound scary, right? All I'm doing is common sense, really. You can search that whole phonebook, assuming it's alphabetized, in simply ten steps. And contrast that, obviously, with the linear algorithm: just as correct, just as obvious an algorithm, perhaps, but so much more expensive—100 times more expensive.

Put it even more in perspective. Suppose that the phonebook actually had one million pages, and you applied a linear algorithm. Worst case, how many steps is a linear algorithm going to take?

A million, right? Easy one.

What about the logarithmic one—the divide-and-conquer approach?

STUDENT: (inaudible response)

DAVID MALAN: Not that bad.

This begs the question, how many times can you divide a million by two?

Twenty. 2^{20} is the same thing as $2^{10} * 2^{10}$, which…

We just said that 2^{10} is 1,024. So 1,024 * 1,024; 1,000 * 1,000 gives you roughly a million.

If that math went over your head, it's okay. You can work it out later. But the point is, twenty steps to search a million-page phonebook.

Now consider the relevance to today's world, right? Millions, billions of Web pages on the Internet. Google, and search engines like that, search it. Clearly, Google and the like are not using linear search algorithms. And daresay they're not even using just these sorts of logarithmic-type search engines. But the point is, they are presumably doing something that's more than naïve, something that's more than obvious. And even with some slight changes; even by taking a few more minutes up front, and thinking, how can I do this algorithm, how can I write this program better, you can imagine the savings that someone like that gets per unit of time.

And even we, though it might have taken us slightly longer, as it did—maybe a couple of minutes to get to the discussion of our logarithmic algorithm—consider if we use that algorithm day in and day out, be it in a digital world or this physical world, just how many thousands of steps you ultimately save. And that's a powerful thing.

This logarithmic running time: We can see it in another way. And I need you to humor me, where we all partake in a demo.

Note, on screen: slide #4

If you would, I need you all to stand up in place. You don't have to come down. But go ahead and stand up. And we're going to execute an algorithm together.

In this algorithm, we're going to follow Step 1, which is "Stand up." Nicely done. Step 2: "Assign yourself the number 1." So think to yourself, "I am the number 1." Step 3: "Find someone else that's standing up." In other words, find a buddy. Say hello if you've never yet met. A little awkward if you're across the room. Don't be left out. And Step 4: "Add your number to that person's number."

A sanity check: Who does not have the number two?

STUDENT: (inaudible response)

DAVID MALAN: All right, one of you might not, if we have an odd number. But hopefully all of you have the number two. Now is the race to sit down. Number 5: "One of you in each pair should sit down."

Never saw some people sit down so fast. Step 6: "If you're still standing, go back to Step 3." And let me ask that those of you still standing, repeat this algorithm. Go ahead, as directed.

And by repeat, I mean, find someone else, add your numbers.

Don't look at me. I don't count. I'm not standing, per se.

Keep on counting. All right, we have two... Kind of standing, yes? Still standing? Just you guys? So, all right, so you guys are presumably on... You just finished Step 3, so add your numbers together.

Okay, so one of you gets to sits down. One of you! Whoa! All right, what's the total number you have in your mind?

STUDENT: (inaudible response)

DAVID MALAN: You would have just added Eric's number to whatever number you had a second ago.

STUDENT: (inaudible response)

DAVID MALAN: "Three!"

You may sit, because this is about to embarrass all of us, I think. There are three students in the classroom room right now, based on this algorithm.

If I wasn't clear, this is an algorithm for counting students. And I will offer that it was correct—and don't worry if you got three, because it means a whole bunch of other people screwed up before you.

So, there are not three people here. Something went wrong along the way. But let's think about what was happening on each step, and see if we can salvage this demonstration, and see where we were going with it.

So, Step 1, you all stood up and had the number one. In Step 2, you all sat down. No, half of... well, practically... half of you sat down. With that in mind, how many iterations of this algorithm did you all go through by the end of the demonstration? In other words, how many times could half of the classroom sit down; half sit down; half sit down?

Well, we have... roughly, I counted... Well, let's see... 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17. I count eighteen students. So we're just only slightly off by fifteen.

So eighteen students. You can divide that by two, and you get nine students standing. Divide that in two and you get, like, four students standing, giving or take; then two students; then one student. So in theory, it should have only taken four passes through this algorithm until there was just one of you left.

Contrast this now with my teacher-like approach to taking attendance: 1, 2, 3, 4, 5, 6, 7, 8, and so forth. Mine is clearly what we would call what?

STUDENT: (inaudible response)

DAVID MALAN: A linear algorithm. Yours, in spirit, was a...?

Logarithmic algorithm. Yes, there's more steps. It's sort of like apples and oranges, because this is one step; whereas, you guys have six steps to go through, or three steps on each pass. But the spirit is certainly the same.

And only because that was really wrong, can you humor me once more? Let's try to get this right, and see… If we get eighteen, then there will be a big surprise somehow, at some point.

All right, so if you'll humor me. Stand up once more. And, actually, we can take something away. What is a "bug" in a program?

STUDENT: (inaudible response)

DAVID MALAN: Something that doesn't work the way you planned. So case in point.

STUDENT: Lack of direction.

DAVID MALAN: Yes. Lack of direction! Oh, programmer error!

STUDENT: (inaudible response)

DAVID MALAN: Well, yeah, I mean, add your number to that person's number.

STUDENT: (inaudible response)

DAVID MALAN: Ah, all right! So, buggy hardware. All right!

So, Step 2: Everybody's the number one. Step 3: Find a buddy. Okay? Add your number. Everyone's number two. One of you sit down. And repeat.

There's three of you. That's okay. Just two of you pair off, and we'll get rid of one of you.

STUDENT: (inaudible response)

DAVID MALAN: What, you have twelve?

STUDENT: (inaudible response)

DAVID MALAN: How many do you guys have?

STUDENT: (inaudible response)

DAVID MALAN: So you have four. That's sixteen students. How many do you have?

STUDENT: (inaudible response)

DAVID MALAN: See, that's twenty students! Okay, I'll take twenty. All right, all take twenty.

So that's not bad. So there's clearly a tradeoff sometimes with the complexity of algorithms, daresay. And this isn't quite fair to compare to my algorithm. Because, whereas I arguably am just one CPU—one brain, so to speak, or lack thereof—counting you, you are arguably are eighteen, or three, or twenty CPUs.

So really, you could say that the way you executed this algorithm would be an example of, like, a multiprocessor algorithm, a parallel algorithm, in which multiple brains are each doing something at once.

And we've certainly talked about that idea in terms of dual-core processors being able to do two things at once. This is an extreme case. This is like a computer having twenty-some-odd processors, but the idea is the same.

(00:20:00)

In theory, by throwing hardware at a problem, sometimes you can just double the speed. But by throwing CPUs at *this* problem, I'm able to fundamentally change the running time of the algorithm.

So the benefits, therefore, really depend on how you're using that hardware, and if you're just turning a dial and speeding things up, or if you're fundamentally changing the solution. And that's the fun part, when you get to design more clever, better algorithms for solving problems.

And again, to come back to Google, that, in effect, is what they did up front. Right? They solved the search problem better. And efficiency was certainly among the challenges they faced.

All right, so where could we be going with this? Well, it's one thing to talk about things in terms of algorithms, and sort of spoon-feeding algorithms to you. It's another thing to begin writing programs yourself, implementing algorithms yourself, because precision becomes ever more important. And by precision in this context, I mean being so precise, and essentially acknowledging that that computer, no matter how fast or expensive it is, it's just a machine, and it will only do what you tell it to do. And therefore, if you don't tell it to do something, it's not just going to figure out what you meant, or infer from other instructions you've given it. It's going to do nothing, or do something incorrectly, or just not work at all.

And so in programming, as you'll see, as you work on your Problem Set 8, being precise and being specific, and being thorough becomes ever more important. Because the downside of not being so is that your software just doesn't work.

How many of you have ever been running some program on your computer, and all of sudden it hangs, it freezes, it slows to a crawl? Well, that's because somebody somewhere along the way screwed up. It's not the computer's fault. The computer is doing what it's told. And so that error must fundamentally, you know, 99 times out of 100—if we ignore hardware issues and cosmic rays, striking your computer—was human error. Even if you've never met that person before, they at least contributed to the software.

So let's try something that always gets me into a bit of hot water here… We are going… because of the pain that this doll sometimes suffers. So we are going to change a baby's diaper. We are going to implement an algorithm that you are going to craft verbally. I will be the computer, the dumb computer that only does what he is told. And our goal in this exercise is to change this baby's diaper. And clearly the stakes are high, since we're changing a baby's diaper. And I—as a few of you saw who got here earlier—I have never done this before. It took me ten minutes just to get the first diaper on the child.

So I need step 1, from a father, from a mother, from someone with more intuition about this than I. How do I begin to change a baby's diaper?

STUDENT: (inaudible response)

DAVID MALAN: "Place the baby on a safe surface."

So this is where I would start getting a little reckless. But I'll be gentle at first. All right. Safe surface. Step 2. Someone else?

STUDENT: (inaudible response)

DAVID MALAN: "Take the clothes off. Unsnap the..." Let's roll back. Step 2. So it's step 2? "Turn over..." I did turn over the baby.

STUDENT: (inaudible response)

DAVID MALAN: Got to be precise. Tell me what to do.

STUDENT: (inaudible response)

DAVID MALAN: "Find the snaps." Okay. I mean, here, too, you're not being very... Here there are. Step 2.

STUDENT: (inaudible response)

DAVID MALAN: I want something along the lines of "Be gentle with the baby," or "Lift the baby. Look for snaps." Anything!

STUDENT: (inaudible response)

DAVID MALAN: You're making this too easy!

STUDENT: (inaudible response)

DAVID MALAN: All right, "gently." I'll take that. All right, gently.

STUDENT: (inaudible response)

DAVID MALAN: The point clearly has not come across yet. So we're going to keep doing this until it does.

STUDENT: (inaudible response)

DAVID MALAN: I like that, I like that. All right, I've found them on the back of the clothes.

STUDENT: (inaudible response)

DAVID MALAN: None in between the legs. Just on the back. It's like a sleeper.

STUDENT: (inaudible response)

DAVID MALAN: "Without moving the baby." Okay. It's technically Velcro, but that's okay. The Velcro? Okay, done.

STUDENT: (inaudible response)

DAVID MALAN: Okay, not bad: "Without harming the baby, gently remove the clothes to expose the diaper." All right.

STUDENT: (inaudible response)

DAVID MALAN: All right.

STUDENT: (inaudible response)

DAVID MALAN: All right, "Without crushing the cranium." All right. You didn't say that the first time. All right, so now we pretty much have... And just... Okay. Notice I've never let go of the baby. All right? Step 7?

STUDENT: (inaudible response)

DAVID MALAN: "Gently put the baby back down." All right, that's not bad.

STUDENT: (inaudible response)

DAVID MALAN: "Without moving the baby, remove the diaper." "Carefully."

STUDENT: (inaudible response)

DAVID MALAN: Something tells me this was not intelligent. But it's not done. Step 9?

STUDENT: (inaudible response)

DAVID MALAN: "Throw diaper..." All right, "Throw diaper away into..." All right, step...

STUDENT: (inaudible response)

DAVID MALAN: "Pick up new diaper." All right, we're moving along, nearing the end.

STUDENT: (inaudible response)

DAVID MALAN: "Prepare diaper." We're calling another function all together. What does it mean to prepare a diaper?

STUDENT: (inaudible response)

DAVID MALAN: "Open it up." All right. Done!

STUDENT: (inaudible response)

DAVID MALAN: All right. "Gently wash the baby." Let's assume that just happened. Next?

STUDENT: (inaudible response)

DAVID MALAN: "On baby." "Under baby." All right, and actually, I'm going to ask... The things, the straps. Where do these go?

STUDENT: (inaudible response)

DAVID MALAN: I'm not even joking around. Front or back?

STUDENT: Back.

DAVID MALAN: Back? All right. All right. Whoops!

STUDENT: (inaudible response)

DAVID MALAN: All right, I'm sliding it under the baby. All right. And?

STUDENT: (inaudible response)

DAVID MALAN: "Pull the straps..."

STUDENT: (inaudible response)

DAVID MALAN: "Pull the bottom tab up around the top."

STUDENT: (inaudible response)

DAVID MALAN: "On both sides." Oh, good, and?

STUDENT: (inaudible response)

DAVID MALAN: "Gently dress the baby." All right. I think that deserves a round of applause for someone. All right!

(students applaud)

All right, so, excellent. So the lesson hopefully eventually came across that, even when we try to tell another human how to do something or what to do, even we make assumptions. That's perfectly reasonable in a society.

But when you start to deal with what fundamentally is a dumb device, but, granted, a device that's much faster than you at doing certain things, you can't make these assumptions. And you have to think through what could possibly go wrong? What situations might arise that this program needs to know about preemptively?

Make this more real. Think about going to a Website. Well, a lot of Websites have forms you fill out, like when you buy something from Amazon. Well, among the things a programmer for Amazon has to think about is, how do I handle a user's buying an item?

Well, typically to buy an item, you add something to your shopping cart, and then eventually you provide your first name and last name, your street address, your credit card number, and so forth. You fill out a form.

But immediately should occur to you the fact that, what if a user messes up, or forgets a field, or chooses not to fill out a field? That's sort of an event you have to handle.

And error checking is a huge part of writing software, certainly software that has to interact with humans. Because where I was going with that was this: Suppose that the user forgets his zip code, or forgets the three- or four-digit secret code associated with his or her credit card number; clicks Submit. Well, you probably don't want to go ahead and forge ahead, and say, "All right, your item's en route to you in seven to ten business days."

Well, no, you want to check that the input was correct. You want to go back and check, allow the user another chance to input that.

Moreover, if someone types in alphabetical characters to their phone number, well, that might seem kind of stupid, right? Who would do that? But someone might. Might be an accident. They might be trying to hack your site; figure out, are there bugs that I can exploit to take advantage of some security hole in your system? You've got to handle that, too.

And one of the hardest parts, or certainly most time-consuming, or frustrating parts about writing software that interacts with humans is dealing with every possible case.

Because the downside of not doing so is that, either, one, your software breaks in some way; worst case, your software breaks in a security sense, such that because you didn't consider some scenario, someone's able to type in not just their phone number, but a line of, say, SQL code, which is a database language.

And because you were sloppy when writing your code, if someone happens to write programming code where their phone number goes, because you didn't think that someone might dare try something like that, the result, when they click Submit, is that your entire database is deleted. Because what they typed in for their phone number is something like,

(writes on board):
delete * from database;

And this is just an example of a query that one might type in the world of database programming.

(00:30:00)

But if you don't check that someone is actually inputting a valid phone number, and instead allow them to type crazy stuff like that, you're putting yourself at risk. So not only might the program break for the user, the program might also break in the backend in a very expensive, or certainly unpredictable way. These are very common situations.

And we talked about hacking, and so forth, in our Security lectures. Step 1 in a lot of hacking is just banging on a computer or on a server, and trying to pass it bogus input; really big input; no input at all, and just see what happens. Because usually if, by providing a computer with bogus input, you can compel it to crash, that's usually a sign of some bug running in that computer's software. And whenever there's a bug, there's a potential for exploiting it, and taking over the machine; accessing the machine; deleting data from the machine. It really varies.

And so a lot of hacking in the beginning is just futzing around, see what breaks. And once you see, "Ah, something broke!" then home in on that and see in what way is it breaking. Incredibly important: precision and thinking these things through.

Yes?

STUDENT: (inaudible response)

DAVID MALAN: Good question. So, legality, and so forth. If you started banging on Amazon's site just to see what would happen, could someone come knocking on your door and, you know, do something about it. I think the short answer is that it depends. A well-written Website should detect that you're banging on it and, for instance, making too many queries of it too quickly. And ideally that Website should respond by just ignoring you, or locking you out. And some Websites do this.

Even FAS: If you forget your FAS password and you try guessing every password you've used in recent years, but you try too many guesses, a system like FAS, like many good systems, will say, "You know what? You're locked out for fifteen minutes because we are suspicious of this behavior."

Now, that doesn't permanently lock you out. It might annoy you. But the upside is that now the system is protected against a brute-force attack. Because, yeah, someone can fifteen minutes later try another ten passwords, but they can't try ten passwords per second. And you've pretty much choked off an attack like that.

So what would happen with something like Amazon? Odds are, unless you really put some time into it, and were clever about how you were banging on a Website, probably the traffic would not be noticed. Unless you tried logging in, or doing something that is fundamentally related to security, odds are the site would break in some way, or it would just behave and give you an error message, which isn't necessarily breaking, but it's Amazon's way of just saying, "Eh, something went wrong. It's not a big deal. Here's an error message."

STUDENT: (inaudible response)

DAVID MALAN: How much do you have to know to use cookies on a Website?

STUDENT: (inaudible response)

DAVID MALAN: Mm-hm?

STUDENT: (inaudible response)

DAVID MALAN: Yes, so, to summarize: Many Websites out of the box these days, just by default, use cookies in some way. They put cookies on your computer, but they don't seem to use them in a way that you would notice.

That is for a couple of reasons. They might very well be using cookies, but in a way that you don't notice. And if you have cookies turned off, they are smart enough to figure out how to still interact with you in some way.

Other sites might simply have been using some off-the-shelf software—Apache, Microsoft IAS [Internet Authentication Service]—and they just turned on the Cookie option, which means, by default, every Web page you visit is being sent along with a cookie. They're just not using it. It's just the default setting, which doesn't mean it's good or bad. It just happens to be there.

The short answer is it depends. But if you go through your Windows folder in C:\Documents and Settings...

STUDENT: (inaudible response)

DAVID MALAN: You'll see dozens of cookies from almost every Website you visited. And most of those sites are probably not using them again in a way that you would notice. Though it's becoming increasingly common, it's not really something to worry about. In years past, there have been more bugs in browsers that have resulted in cookies, as we said a few weeks ago, being a potential security hole. And theoretically, they still are. It's not something I would lose sleep over. There are so many more important threats—viruses, worms, spyware—that pose a much more imminent threat. Cookies, I think, are far more useful than they are dangerous. Though I might, for instance, turn off the third-party cookies, as I demo'd in class one day, just because. But that's about it.

All right, so, how can we now begin to approach programming a bit more procedurally? Dare I ask for a volunteer? This volunteer doesn't need to speak, but they do, at the risk of terrifying, need to sit here. Anyone at all? Nonspeaking role. This is a tough one, I know. Anyone? Oh, and you have to take your shoes off.

STUDENT: (inaudible response)

DAVID MALAN: Anyone? Dawne? Yes? Ah, Dawne is a trouper in this class. If we could?

(applause)

All right.

Dawne is saving one of you from coming down and taking your shoes off.

So in order to transition from this very high level discussion of programming, let's introduce something that's specifically known as "pseudocode." Typically, when writing a program, certainly for the first time, one won't just start by writing Java code, or C++, which are common languages. They'll rather sketch out ideas on paper: a framework, an outline, if you will, just like a high school essay might be outlined in English, or shorthand.

Well, pseudocode is something that typically looks very much like English, but it also has a mix of some programming syntax, some shortcuts that we'll highlight in the moments that follow, which really get across the idea of the algorithm you want to implement, without letting you get bogged down in some of the details, the syntactical details of the language, which might take a little more getting used to. Everybody understands English, though. Everyone should hopefully understand pseudocode.

You're looking very awkward or at least amused. So, um, we'll take either.

Here's an algorithm with which Dawne will be able to put on, hopefully, her socks.

Note, on screen: slide #6

We've numbered the lines for the sake of discussion, but a lot of programming environments will show you line numbers for your own sense of discussion, with other people perhaps.

Line 1 says, "let socks_on_feet = 0".

And notice I've used underscores instead of spaces to sort of imply that socks_on_feet is some self-contained entity. And that's going to be called a "variable." Right? In algebra, there's variables x, y, z. In programming, you can call variables x, y, z.

Variables simply hold values, whether it's a number, or a word, or a letter, or a sentence. But it's more useful in programming, typically, to name your variables not x, y, or z, but rather something that's a little more instructive.

I figure we'll both be on camera, rather than just one of us here.

So, Step 2: "while socks_on_feet != 2".

What do you think that's getting at? What does that mean?

STUDENT: (inaudible response)

DAVID MALAN: Perfect. So how do we begin this next step in our algorithm, this next action? Well, do so by realizing that socks_on_feet does not equal...

!= is programming shorthand for "not equals" because there's no key on the keyboard that's an equals sign with a line through it. So that means "not equals 2."

And this makes sense, right? Because if we're putting on Dawne's socks for the first time in the day, she presumably doesn't have socks on yet. And in fact, if you wouldn't mind removing the shoes and socks. And then we'll be perfectly set to go in just a moment.

So it would make sense that socks_on_feet at the first thing in the day is equal to zero. And so it's certainly not equal to two.

So in Step 1, we declared a variable and initialized it to zero: "socks_on_feet = 0".

Step 2, we begin what's called a "loop," which is a fundamental programming construct...

I know this is difficult. It's a longwinded lecture, isn't it?

DAWNE: No, I'm okay. I'm fine.

DAVID MALAN: Okay, so Step 2, we launch into a loop, which is quite intuitively a process that's going to repeat, hopefully just two times. Because she has two feet, we'll hopefully have two socks, and we'll be done. And you'll be out of this awkward moment.

Step 3: So notice the indentation. So this too is sort of a detail of many programming languages, where, to convey the idea that this next line should only happen if the previous line was true, or applicable, we indented, to sort of show that now the implication is that everything from line 3 to line 17 should proceed to execute.

And Chris, as we do this, if you wouldn't mind spending time on the board, too, that would be great.

So Step 3: "open sock drawer".

Here's your opportunity to shine. Pretend to open a sock drawer. And we'll need you to take the... Oh, are these socks, or are they, like, uh, hosiery?

DAWNE: Yeah.

DAVID MALAN: Okay, that's going to be awkward. Yes, no, so we're going to pretend that Dawne has socks to put on. That's okay, we'll take what we got.

So Dawne has just pretended to open sock drawer.

Step 4: "look for sock".

Your moment to shine.

All right, "if you find a sock then"...

Here's a decision point. Did you find a sock?

DAWNE: Yes.

DAVID MALAN: She found a sock, so Line 5 is, as they would say, "True." Notice the indentation. We do now everything that's indented below Line 5.

Line 6 says, "put on sock".

Excellent!

"socks_on_feet++". What could that be meaning?

STUDENT: (inaudible response)

DAVID MALAN: Both feet? Not quite. This, too, is a programming shorthand notation that you'll often see.

STUDENT: (inaudible response)

DAVID MALAN: Completed action. Kind of.

STUDENT: (inaudible response)

DAVID MALAN: Incrementing. Excellent! So it's a good inference. socks_on_feet was our variable. In many programming languages, when you write a variable and then ++, it just means "add 1 to that value." That's all.

(00:40:00)

So, socks_on_feet, our variable, has been incremented. And this is our programmatic construct for remembering in code how many socks are on Dawne's feet. So we don’t have to keep checking, for instance.

All right, Line 8 says, "look for matching sock".

All right, "if you find a matching sock then". Did we find a matching sock?

DAWNE: Yes.

DAVID MALAN: We did find a matching sock. We're going to go to Line 10: "put on matching sock".

Excellent. Line 11: "socks_on_feet++".

Line 12: "close sock drawer".

All right, and now, just to be clear, we don’t want to execute Lines 13 or 14 or 15, because what we began to do in Line 9 was an example of a branch, or a condition; a sort of fork in the road.

If Line 9 was "True," then we did the indented Lines 10, 11, 12, beneath it.

Else, had Dawne not found a matching sock, the idea would be that we would instead go to the "else" line, the other fork in the road, the other direction, and we would instead execute the two indented lines on Lines 14 and 15.

So now we find ourselves down where? Line 16—it doesn’t apply, because notice Line 16 is associated with which other line, based on the indentation?

STUDENT: (inaudible response)

DAVID MALAN: Line 5, right? But Dawne did find a sock. So we follow that first big branch, and not this latter one.

So now we're at the end of the program. But remember how we got here.

What was the line of code that launched us into those Lines 3 through 17?

STUDENT: (inaudible response)

DAVID MALAN: So, Line 2, recall, right? That was the loop that Eric said sort of launched this whole process into motion. So whenever you have a loop, that means that eventually, just like with our self-counting algorithm for students, you go back to wherever the line of code was that started that loop.

Well, that was Line 2. And Line 2 says, "while socks_on_feet != 2".

Is socks_on_feet equal to 2 at this point, or not equal to 2?

STUDENT: (inaudible response)

DAVID MALAN: It is equal to 2 because we incremented it twice, which meant she has two socks on her feet. So that means, because there's nothing else in this program, now the program is complete.

A round of applause, if we could, for Dawne, for humoring us.

(applause)

Excellent! And now let's see. Dawne is welcome to return to her seat, if she'd prefer.

DAWNE: (inaudible question)

DAVID MALAN: There doesn't need to be. It's okay.

Thank you.

Is there anything wrong with this program? Can we debug it?

It looks like it executed perfectly. You all recited the lines perfectly. Dawne executed the lines perfectly. It didn't look like we needed any... there were any assumptions that led us astray.

So is there any potential mistake or bug, as they say, in this program that would cause unintended behavior?

STUDENT: (inaudible response)

DAVID MALAN: Excellent! You've just offered a so-called "corner case." What if she only has—Dawne—one sock in her drawer, and she executes this program? Describe for me the symptoms that would occur if Dawne only had in her drawer, for whatever reason, one sock?

STUDENT: (inaudible response)

DAVID MALAN: Exactly. So we have in that scenario, it results in an infinite loop, a loop that seems to go on forever? Why?

Well, if she has just one sock in her drawer, well, then, we do, in fact, in Line 15, on the first iteration of this loop, find a sock.

Note, on screen: 5. if you find a sock then

And so Dawne puts it on.

Note, on screen: 6. put on sock

She increments the variable.

Note, on screen: 7. socks_on_feet++

She looks for a matching sock.

Note, on screen: 8. look for matching sock

But this time, in this scenario, she doesn't find one.

Note, on screen: 13. else

And so we execute these.

So Dawne removes the first sock from foot

Note, on screen: 14. remove first sock from foot

decrements the variable: minus, minus

Note, on screen: 15. socks_on_feet--

And then where does she go back to? Well, back to line 2: "while socks_on_feet != 2 ".

At this point in the story, is socks_on_feet equal to 2 or not equal to 2?

STUDENT: (inaudible response)

DAVID MALAN: It's not equal to 2, because it started at zero.

She incremented it to one; decremented it again; so it's back at zero, and so the loop repeats.

This time in Line 5, what does she find in the drawer? The same darn sock.

And so the program—though you might think this sort of corner case is a little silly—it is incredibly representative of what happens all of the time in programming: failing to consider what might be a rare but feasible scenario.

And, just as we've described it, an infinite loop suggests something that goes on ad infinitum, forever. Well, have you ever been sitting in front of a program, your computer, and it just seems to, again, hang—not freeze, but it slows to a crawl. And it looks like, based on the light in the front of your computer, it's doing something, but it's never giving you control back.

Well, that might be an example, potentially, of one of these infinite loops. The program, because of a mistake the programmer made, somehow got stuck in a cycle because he or she did not consider whatever the user might have just typed in, or where the user might have just clicked, or what data might be available; what socks might be available to that program.

The takeaway here, then, is, one, there's these fundamental constructs in programming. You see them in silly things, like algorithms written in pseudocode for putting on socks.

But among those constructs are variables, loops, and these things called branches. And almost every language in which one would tend to write software has, among its constructs, certainly those three. And those are constructs that, despite their simplicity, prove to be very powerful.

Well, let's now make things more real. That was pseudocode—sort of an amalgam of some programming-language-like syntax, but also some actual English.

What is this here on the display?

Note, on screen: slide #7

What's this?

STUDENT: (inaudible response)

DAVID MALAN: XHTML, right? This is Extensible HyperText Markup Language. Well, there we go. It seems that we've already seen a language before. But just to be clear, we haven't seen a programming language before.

XHTML, and some might try to debate this, is a markup language, which means it's good for the kinds of things you've begun using it for: making Web pages; marking up text; presenting movies, and sounds, and images, and so forth.

It's not a programming language. Because in XHMTL itself there's really no notion of branches, or conditions. Rather it's all statements: Do this; do this; do this. And by that I mean, make this bold; make this green; make this big; make this centered. XHTML is about marking up a document.

But a programming language, fundamentally, let's you do anything. It allows you to manipulate in the computer in a way that allows you to solve problems, whether that's finding a phone number, searching some database, or simply allowing someone to check out with a shopping cart.

Well, what's a real programming language? This is an example of a language called what? Anyone know?

Note, on screen: slide #8

STUDENT: (inaudible response)

DAVID MALAN: Not Java. What's another you've heard of?

STUDENT: (inaudible response)

DAVID MALAN: Not Basic. What's another?

STUDENT: (inaudible response)

DAVID MALAN: Not C++. What's one more?

STUDENT: (inaudible response)

DAVID MALAN: Doesn't exist. Well, not really. Well, think about it. Not JavaScript, but another popular one.

So there's not really a C+. There's C, and then—har, har, har—C++ was the next version of C, and you know what "plus, plus" means. It means "increment." So this is an example of a language called C. It was one of the earliest high-level languages, which simply means this was among the earliest languages that people could program computers, without dealing with punch cards, without writing out zeros and ones themselves, without writing out very esoteric instructions that only a computer's CPU would understand.

C was among the first languages that allowed you to sort of write almost at that pseudocode level, where you can sort of write as you might speak but, granted, you use a bit more arcane syntax than we did for our socks example.

This is an example of source code. Whenever you write a program in a language like C, it tends to look like this. Parentheses are common. Squiggly braces are common. Words like "print" and "exit" and "int," for integer, and "main" are common keywords that you might see.

But a computer does not understand this. A programmer understands this. A computer understands only what alphabet?

STUDENT: (inaudible response)

DAVID MALAN: Binary, or zeros and ones. And so before you can actually run a program, you typically have to do what's called "compile" it, which literally means to take your source code and convert it, effectively, into patterns of zeros and ones that a computer does understand.

So when you are a programmer, and you're writing Windows software, you'll probably use a program like Visual Studio, or Visual C++. These are just programs that Microsoft wrote that allow other people to themselves write software. And among Visual Studio's features is the ability to make this red arrow happen: to convert what a programmer has written into the zeros and ones that an x86 CPU, an Intel CPU, understands.

Similarly might you write code very much like this on a Mac, or on a Linux computer. But if you want that program, that source code to be translated into zeros and ones that a Power PC's CPU understands, or some Motorola CPU, well, you need to use a compiler that's specific to that CPU.

So this is why, when you buy shrink-wrapped software off the shelves—even though that's becoming decreasingly common with how much software you can buy and download over the Internet. When you buy something, and the box says—maybe it's a game—"PCs only," that means, because even though the program, though it would be thousands, tens of thousands of lines long, not just six or so lines long, the programmers wrote stuff reminiscent of this, most likely.

(00:50:00)

But they used a compiler that was specific to x86-based PCs, the result of which is that the zeros and ones that they then burned to a CD or a DVD for you to buy from the store are patterns of zeros and ones that only, say, Intel or AMD CPUs understand.

A Mac would not understand those same patterns of zeros and ones.

If a company instead wanted to produce that software for a Mac, they'd probably write something similar to this, though there's enough differences between Macs, and PCs, and so forth, that you can't just take the same source code usually, and run it through a different compiler and get a Mac-based CD. There's a lot more work, typically, that goes into it. Such that, even for a while, I think Microsoft had separate divisions, one of which made Microsoft Word for the Mac; one of which made Microsoft Word for the PC. They're that different operating systems and CPUs.

But the point is, that before you run software and can double-click that icon, it's been converted from something like that source code to zeros and ones that we don't understand, but the computers do.

This is an example of the language that was cited, C++.

Note, on screen: slide #9

Looks very similar. And this is not meant to give you a sense of how to write C++, but just a snapshot of what it looks like. This, too, would be compiled down to patterns of zeros and ones, like this. And, believe it or not, I did take the time to actually compile that code into these zeros and ones, which are literally the zeros and ones that a compiler would output on, I think, an x86-based Linux computer. So these aren't me typing randomly at the keyboard: "010001". These are the real zeros and ones that are physically in that ".exe" file, or the file on, say, a Linux box that's double-clickable as well.

Finally, one other example. This is an example of? Take a guess.

Note, on screen: slide #10

STUDENT: (inaudible response)

DAVID MALAN: That's Java, yes. So this is an example of Java, which is another quite popular language. It is not compiled in quite the same way, but the spirit is the same. Java, for what it's worth, is compiled into something called "bytecodes," not into this stuff that previously I didn't slap a label on, but was called "object code"—a minor distinction that's not an important takeaway for now.

But the neat thing about Java is that, because it's compiled into something that's slightly different fundamentally from what we just looked at for C and C++, compiled into bytecodes and not object code, the implication is that Java programs can be run on any CPU.

So essentially what Sun Microsystems, the inventors of Java, did, was they designed a language that, rather than compiling source code directly into zeros and ones, they rather imposed an intermediate step. So that Java code gets compiled into something called bytecode. And on Macs, on PCs, on Linux computers, and even on other operating systems, even some of your cell phones these days, there is what's called a "Java Virtual Machine," a JVM, for short.

And so in the world of Java what happens is a compiler for Java translates the source code that programmers write into something called bytecodes, which looks similar, but it's not quite zeros and ones; it's closer to the original source code. Those bytecodes are portable, so to speak. They can be run on any platform, provided there exists what's called a Virtual Machine for that platform.

Long story short, all a Virtual Machine, or JVM, is, is a program, literally, that understands bytecodes and executes them on behalf of the CPU. The implication is that underneath, in this sort of layered idea that we're making, you can have an x86 CPU; or you might have, you know, in yesteryear, a Power PC's CPU; or you might have something that's from other domains: a Spark CPU, which is another CPU, more for corporate use all together.

The point, though, is that the code itself can be run anywhere. Because effectively, people have written software that effectively converts bytecodes to whatever the zeros and ones are that each of these CPUs requires software to be written in.

So originally, Java was touted as this wonderful language because, finally, you could improve development time, and efficiency, and earnings because you could just write software once and run it everywhere. And Java was touted as a wonderful language for the Web because you could download these so-called "Java applets" in your Web browser. And then, no matter what kind of computer—Mac or PC—you were running, you could run this software in your Web browser.

Well, Java applets, if you've ever pulled one up, are typically very slow and not terribly fun to interact with. And so you really never saw this happening.

But Java is in fact being used quite a bit these days for server-side development. A lot of big, very high capacity Websites tend to be written in something called Java; or Java Enterprise Edition is just a fancier way of saying Java with a whole bunch of special libraries, extra add-ons to it.

Whereas most software that you might install on your PC these days tends to be written still in C++, or a language called C#. But for the most part, those are perhaps the most popular languages. C#, C++, Java, and then there are other languages used, say, on Websites, besides Java. PHP is another one.

C, C++, PHP, Java we've mentioned, Pearl is another popular one, C# is another one. Just be familiar with these kinds of terms, and just know that these are alternative languages sometimes that make software development easier; sometimes that make it harder; sometimes that enable the programmer to do things that he or she couldn't do in other languages.

But for the most part, one's choice of languages—Python is another popular one—boils down partly to what infrastructure you already have in place; that is, what is your company invested in, and therefore, what language do you need to use to continue using that same infrastructure; two, what do you know in the first place? It's a lot faster to develop in something you know, rather than go learn something new, and then develop in it; three, where you want to run the software, and you can come up with a half a dozen other arguments, I'm sure. The short answer is it just depends on where you're coming from and what you want to do with the software, what language you use.

So, in years past, we have used a language called JavaScript, which should be on this list as well, because you see it similarly often as well. JavaScript, which is not the same as Java—they just kind of stole the first part of the name from it—JavaScript is a language that Web browsers execute. And it is a language that… you can do things in Web pages like error check fields, so that you ensure—before the user even clicks Submit—that they've provided their phone number, or their name, and last name, and so forth. It can check the format of their email address. And it can do a whole bunch of other things.

Increasingly, is JavaScript being used to make really interactive Websites. The problem is, it's an ugly-looking language. It is difficult to introduce to a class of ostensibly nonprogrammers in just an hour's time, or so. And you can't also do things that are terribly glamorous with it. And certainly not after just being exposed to it once.

What we decided to do, then, this year, is introduce you to something that is a whole lot of fun. This is a programming language called Scratch. And as I mentioned in an email last night, Scratch is a language that's being developed currently by MIT's Media Lab, down the road.

Scratch officially has been developed for after-school computer clubhouses for students, particularly in disadvantaged communities, whereby these computer clubhouses offer kids an opportunity to congregate together; to hang out in a fun, friendly, safe environment, and so forth, and actually do something intellectually stimulating: to solve problems, to implement games, to experiment with art, and so forth.

And so what Scratch is touted as is a programming language that makes it really easy and really fast to implement animations, games, art, and other such fun projects. And the neat thing about it is, is that it's a graphical programming environment.

A lot of the semicolons, and the parentheses, and the squiggly braces we've seen already tonight in these other programming languages, you don't have it in Scratch. Rather you do exactly the sorts of ideas that we've been talking about tonight. But you do it by way of puzzle pieces that only snap together if it logically makes sense for them to do so.

We used this for the first time this past summer in a Harvard Summer School course, with a similar group of diverse students, from high-school age, to college-age, to retired age. And we got some fantastic projects out of this, literally after the students having just sat down with Scratch, after a bit of introduction from me, and then taking the ball and running with it.

For instance, just to put this into perspective, among the types of submissions we got this past summer... I'm just going to whisk us away briefly to the Website for that course, which was Computer Science S-1, "Great Ideas in Computer Science."

We had a little gallery on the course's Website. You don't have to quite remember how to get back here. But just to give you a sense, this page is full of screen shots of the types of animations, or games, or programs that students this past summer wrote that give you a sense of sort of the playfulness of the language.

But mind you, these were students, many of whom had never once programmed before. But literally after one night of class, and a few hours of playing around on their own, they were able to implement things like this soccer game, where you try to score goals; this baseball game here; this sort of fairytale, which tells an animated story about some, I think, knight strolling up through a castle area; this sort of fun musical thing, as I recall, with a dog, and a dinosaur, and some graffiti on the back wall; this, which I'll demonstrate probably later, which is another fairytale of a fairy princess and a knight, and so forth. Again, literally, students who had never programmed before.

And the reason we introduce Scratch is because it allows you, the students, the neophytes, arguably, to hit the ground running immediately. And that's not something you can do otherwise without, say, a full semester of programming.

(01:00:30)

So with that said, let's take a five-minute break and return.

Hour 1

DAVID MALAN: We're back.

REI DIAZ: So this week, Section's going to be on CSS, or Cascading Style Sheets. We'll use them to prettify the XHTML stuff that we worked on last week. This Saturday will be the exam review, and we'll cover everything from the last exam up until today's lecture, in preparation for the second midterm.

Also, this Saturday will be Section at its normal time, and Workshop will be on Macromedia Flash—the animation of the horses, and stuff like that. Perhaps not that advanced, but we'll show you how to create some basic Flash animations and other neat stuff.

DAVID MALAN: Awesome, thanks.

And one of the goals for Eugenia's Flash Workshop will be, beyond just empowering you to make these animations, is perhaps to do it with an eye toward your Final Project. If you would like potentially to incorporate whatever Flash you produce in the Workshop, you're by all means welcome to incorporate it into your Final Project. And after Exam 2 next week, like last time, we'll do a little special something.

With that said, I need a volunteer who's good at kicking goals. Or bad. Bad is even funnier. So come on down if you're bad, too. You get to interact with one of the games submitted by one of our students this past summer. It's a penalty-goal kicking contest of sorts. Anyone at all? More fun?

I'm very good at it. It's more fun if one of you, who've never tried it before, plays.

Dawne will come up. Dawne is on the payroll now. Meet your new teaching fellow, Dawne.

Come on down. What I'm going to do is, currently, just to give you a sense—and you'll get a tour of sorts of the Scratch program. But long story short, I've fired up the software. I opened the student's program, and I'll show you how to do that in a moment. And then I simply clicked this little easel icon down here, which makes the program full screen. And Scratch is very nice, in that to run a program, you click the green flag, at top left. And if we want to stop it, we click the red light instead.

So, with that said, I'm going to hit the green flag. And as I recall, Dawne will simply use the Up/Down keys to move the soccer player up and down on the field. And I think you can hit Left or Right when you want to kick the ball. And meanwhile, the goal's going to be moving.

Soccer game vendor: Hot dogs, hot dogs, beer, hot dogs!

(soccer game audience cheers)

DAVID MALAN: So what happens if... ah, let me see. It's not the keys? Oh, there you go. Click the guy. So just click this in here.

DAWNE: Oh.

DAVID MALAN: Yeah. So you can't... Turns out you can't move up and down, but you can... So the cursor's on the guy. So just click the button when you want to shoot. And the idea is... The goal moves. The guy stays in place. Oh, and the keys don't do anything. I completely set that up wrong.

Right, yeah, there you go. Okay.

Soccer game vendor: Hot dogs, hot dogs, beer, hot dogs!

(soccer game audience cheers)

DAVID MALAN: Yeah, okay. So just click when you're ready to kick. No, no keys, no keys. Just this button.

DAWNE: Oh.

DAVID MALAN: Just that button. Sorry. Game's even simpler than I thought. You just hit one button.

Soccer game vendor: Hot dogs, hot dogs, beer, hot dogs!

(soccer game audience cheers)

Soccer game announcer: Go-o-o-a-a-l!

Soccer game vendor: Hot dogs, hot dogs, beer, hot dogs!

(soccer game audience cheers)

DAVID MALAN: Give us one more.

Soccer game announcer: Go-o-o-a-a-l!

DAVID MALAN: All right! Very well done! Nicely done!

(applause)

And actually, let me keep Dawne here for just moment. Because I have feeling I won't get a volunteer. What I'm going to do is I'm going to go to the Open menu, up top. I'm then going to say, no, don't save the current project, because we haven't made any changes.

I'm going to go into our Lecture 11 folder, which you will be walked through in Problem Set 8, as to how to install that Lecture 11 folder, if you want to play with these demos. And I'm going to load up one other program that was written by a former student, a former student, when he was in a class at MIT.

So this is the first program I wrote in Scratch. What you're about to see and hear is a song that I heard for six hours straight, while writing this program. And you'll know what I mean in a second.

Do we have any volunteer who would like to knock Dawne off her throne down here, or otherwise Dawne will get to play "Oscartime." Yes? All right, come on down. Thank you to Dawne.

We have a replacement. You have to be comfortable appearing on camera, needless to say.

"Oscartime" is also a program written in Scratch. It is all about watching as the trash is about to fall from the sky. And you, simply using the touchpad and mouse, need to pick up the trash by clicking it, and drop it into Oscar's trashcan.

With that said. It's about a two-minute game. The limit will be clear, based on the song. And here we go.

STUDENT: Just click right here?

("Oscartime" music starts playing)

DAVID MALAN: Yup, you're going to move the mouse around.

STUDENT: Okay.

Oscar (singing): Oh, I love trash...

DAVID MALAN: And just click the trash and drag it to the trashcan.

Oscar: Anything dirty or dingy or dusty...

DAVID MALAN: Now it's on the ground, so now you can... It's easier now. It gets easier. Yup. Click on the tra... Yup, there you go.

Oscar: Oh, I love trash.

DAVID MALAN: Oh, you have to use the left button.

STUDENT: Oh, okay.

DAVID MALAN: There you go. All right! Whoo!

Oscar: …and the laces are torn
A gift from my mother the day I was born...

DAVID MALAN: There you go, there you go.

Oscar: I love it because it's trash.

Oh, I love trash
Anything dirty or dingy or dusty
Anything ragged or rotten or rusty
Yes, I love trash.

Here's some more rotten stuff.

I have here some newspaper thirteen months old
I wrapped fish inside it; it's smelly and cold
But I wouldn't trade it for a big pot of gold
I love it because it's trash.

Yes, I love trash…

DAVID MALAN: It's almost over.

Oscar: Anything dirty or dingy or dusty…

DAVID MALAN: It's a good score so far.

Oscar: Anything ragged or rotten or rusty
Yes, I love trash.

Now look at the rest of this junk.

DAVID MALAN: Big finale.

Oscar: I've a clock that won't work
And an old telephone
A broken umbrella, a rusty trombone
And I am delighted to call them my own
I love them because they're trash.

Oh, I love trash
Anything dirty or dingy or dusty
Anything ragged or rotten or rusty
Yes, I love, I love, I… love trash!

(song ends)

DAVID MALAN: All right! A big round of applause!

Never seen a volunteer leave so quickly. Okay. That's very good: 41! Now, granted... In retrospect, I realize maybe I should have made the song a little a little shorter. But it is synchronized with the words he's singing. And that's a very good score.

What you'll see in Problem Set 8 is that you're invited to play "Oscartime" yourself. And for a point, try to come up with your own high score. And you'll see, if you look around at the program—we won't do this tonight—that this program is the result of implementing… looks like nine sprites, as they're called.

So let's go ahead and dive into Scratch. And realize that, though we're introducing you to this graphical language—that allows you, again, to make animations, artwork, games, and so forth—it's the constructs that this language, Scratch, supports; and it's the fact that you are in fact programming with it, that are the real takeaways.

We just have the fortune this year, because of MIT's Media Lab—specifically the Lifelong Kindergarten Group, over there. They've given us access to the software. You'll be able to download it from the Web yourself, install it on your computers, and begin to explore programming in what we think is a very fun programming environment.

So, with that said, this program here is the Scratch version of the program in the previous three slides.

Note, on screen: slide #11

So we saw a C program, a C++ program, and a Java program, each of which implemented a terribly simple program—the first program ever written, if you will, called "helloWorld," which simply prints to the screen the words "Hello world!"

But recall the kinds of syntax that you saw in these programs. You needed to write, not the zeros and ones, but these parentheses, and the fairly cryptic-looking syntax:
public static void main (String [] argv)

Note, on screen: slide #10

I mean, just to get yourself into the simplest of programs—one that prints to the screen, "Hello world!"—there's a lot of overhead. There's a lot of distraction. Similarly was the C++ version similarly difficult to acclimate to, perhaps on first pass. And the C version wasn't much better at all.

(01:10:00)

Fast-forward now to Scratch, that same exact program becomes a two-piece puzzle, so to speak.

Note, on screen: slide #11

So the first file—and these are files you can play with, if you download them for the project—is one called Hello1.scratch. So in each of these examples is the name of the file that implements the program you see on the screen.

To download Scratch, as your own Problem Set 8 shows, you can access it via this URL, which isn't public, per se, but the Media Lab has been kind enough to offer you, E-1, and you, the E-1 podcast subscribers, access to the software via this link.

And by all means. I would encourage you, those of you with kids, in particular, if you want to have your kids, for once, help you with your homework, if you haven't already, this might be the assignment to do so, because they, too, might get a kick out of it.

So how do we use Scratch? Well, I load the software by double-clicking the Scratch.exe, or whatever icon on my desktop. And the problem set tells you how to get to that point. Just going to be a folder, say, on your desktop. Mac or PC is supported, even though I'm using a Mac. No, I'm using a PC.

What you do to open a project like that is to simply click the "Open" option at top. If you haven't made any changes to the program, you just say, no, save no changes.

And then you'll get the default folder that Scratch comes with. Scratch comes with a whole bunch of projects that MIT teaching fellows, and faculty, and also kids in these computer clubhouses have contributed to the project. So what you get out of the box for free is a whole bunch of demo games, and artworks, and animations, which we'll leave to you to explore.

We'll focus on a few that convey some of the basic constructs we want to introduce tonight. But realize, that if you just flip through these folders—for instance, games—there's a whole bunch of options in here of varying difficulty; many them written by young children, some of them written by older MIT students. But the point is that it wasn't hard, for the most part, for anyone to dive into this language and start producing some really neat stuff on day one.

We'll spend most of our time in this Lecture 11 folder tonight. I'm going to open the file called Hello1. And as you'll see, we have the following layout. On the left of the screen is essentially all of the puzzle pieces that we have at our disposal. They're categorized into categories like Motion-related puzzle pieces; and Looks-related puzzle pieces; Control-related puzzle pieces. We'll highlight some of the blocks in each of those categories. But for the most part, things are fairly intuitively categorized.

And this is the sort of language that you don't need a user's manual for, really, even though the problem set does point you at some online references for ramping up. It's really the sort of thing that you learn, honestly, by playing around. And if intuitively you think there probably exists a condition, a branch via which I can do this or do that, well, there probably is, under the Control category, for controlling your program, so to speak.

In the middle of the screen, we have the Scripts area. This is where we will write our programs, aka, scripts, by simply dragging puzzle pieces and letting them lock together.

Down here we have our Sprites area. A sprite is simply a character that you can manipulate. The first character we're going to manipulate is that cat on the screen, which is the default sprite. But the neat thing about Scratch is that you can put costumes on sprites.

So what you saw to be Oscar the Grouch was sort of like a cat dressed up as Oscar the Grouch. But really a sprite isn't necessarily a cat. That's just the default costume.

You can make your own icons, be it in Photoshop or in a built-in graphical editor—a simple one, like MS Paint. But you can import graphics, too, if you find something on the Web that you're allowed to and would like to import. So you can do some neat things.

And the soccer player, for instance. That student, as you probably gathered, I think hand-drew the soccer field. And you can kind of tell. But, in the end, it worked out pretty well. And I think that too was his voice, late at night. Scratch also lets you record sound. So he, late at night, was "Go-o-o-a-a-l!" right, much to the chagrin, perhaps, of his roommates.

So you can do some neat things. And we're only scratching the surface of some of the features. But it really is easy to get into.

Finally, you have the so-called Stage, at top right. This is where your sprites actually do something, and can interact with each other, can move around. When I clicked the easel before, all that did was make the Stage full-screen, just because it's a little more fun to see your program fill the screen. But the dimensions of these rectangles are exactly the same.

Finally, you have down here, the Stage. If you recall, in my "Oscartime" game, there was always that "Sesame Street" lamppost on the screen? That wasn't a sprite. That was just the Stage, like a wallpaper, behind the scenes. So that's what you can use the Stage for, to just lay out a graphic, for instance.

So how do we get started? Well, I'm actually going to show you this first. Hello1.scratch is just a program. What is a program? It's one or more scripts. Well, because we've opened Hello1.scratch, we see in our Scripts area the script called Hello1.scratch. Notice that this sprite is highlighted, down here. So that means that that script or those scripts, if there are many, belong to this sprite.

And this is how you'll see you can make different sprites do different things. In a moment, we'll have an example of a cat chasing a bird: two sprites with their own scripts, so that they operate independently. And you'll see what occurs.

This program here begins with the "when green flag clicked" puzzle piece. And, as I said before, to start a program, you click the green button, or in this case the green flag, and that starts your program. So the means via which you get a program to start running is you start your script with that "when green flag clicked" icon.

Anything below that gets executed subsequently. This block here, "say hello, world!" is what's called a statement. When we had Dawne down here, doing her socks example, and we said, "put on sock", "open sock drawer", those were statements. They're not conditions, they're not loops, they're not variables; they're statements: Do something.

So most of these puzzle pieces, such as "say something", is a statement, and it's telling the sprite in this case to do something, to state something. Well, let's go ahead and run this program. You're about to be incredibly underwhelmed by our first program.

I'm programming already, right? So not too bad. Well, let's see if we can't build on this and do some more interesting things.

Well, here's just a snapshot of some of the statements Scratch supports.

Note, on screen: slide #12

You should be able, very quickly, to infer, from what is written on a puzzle piece, if it's a statement, if it's a condition, if it's a loop. Because if it's a loop, you'll see something like the word "while", or "forever", something that intuitively is a loop.

And that's the beauty of Scratch. It's not arcane commands. It's puzzle pieces shaped logically to fit where they should, and also label, as you might expect, intuitively. So "say Hello!" is one statement. "wait 1 second" is another statement. "play sound meow" might be another statement.

How can we incorporate such things as these? Well, here are two new examples: hello2.scratch and hello3.scratch, written in my sort of shorthand notation, up there.

Note, on screen: slide #13

This is referring to two different files, left and right: hello2.scratch and hello3.scratch.

Take a wild guess: What is the script on the left going to do, once we execute it?

STUDENT: (inaudible response)

DAVID MALAN: Say hello for one second; or, really, say hello three times, pausing for one second in between.

Well, let's actually build this. I could just open hello2.scratch, which you can do at home, if you'd like. But let's just reconstruct this.

Well, I'm going to move over here. I'm going to say New. And it's just going to give me an empty everything.

All right, so now I need to go to, let's see, Control, which is where all the major Control structures are. And to make a program, let's click and drag, all right? "when flag clicked".

We could drop it anywhere, but sort of in terms of neatness, let's start at the top left and work our way down.

The next thing I wanted to do is what? I wanted to say something? Well, we don't have to make precisely that program. So let's go to Looks, which somewhat counterintuitively is where "say" is, because "saying" in this domain means showing something, change the look.

So let's do this: "say hello for..." something. In fact, let's make this a little more interesting. What do you want him to say?

STUDENT: (inaudible response)

DAVID MALAN: "I survived E-1". And "for", let's say, "1 second". All right, next? And notice, too: Though I dragged it to the area, I dragged it close enough that you got this white line, which says, "Hey, this will fit here." And as soon as you let go, they snap together, like true puzzle pieces.

All right, let's go over to Control. And you'll get better at knowing where these things are, with practice. Let's "wait 1 second".

And then let's go back to Looks. And let's say something different now. Let's say, "say Okay, I didn't for 2 seconds", and then we'll leave it at that.

All right, let's go ahead and run this.

Not too bad, right? Simple but incredibly easy to build.

Let's look back at this example.

Note, on screen: slide #13

The example at right is going to do what? Infer from the puzzle pieces alone.

In one sentence, what is this program at right going to do: hello3.scratch?

STUDENT: (inaudible response)

DAVID MALAN: Perfect! It's going to meow three times and it's going to pause two seconds in between each.

Let's go ahead and just open this one. And again, would not be hard to reconstruct it manually. This is hello3.scratch.

And, just for kicks, I'm going to expand it to full screen. But the effect is the same. Click the green flag now.

(cat sprite meows three times, pausing in between)

My first program with sounds. And how do I do sound? Well, as you might have glimpsed, there's a Sound panel here. It comes with a bunch of default sounds, but you can also import sounds. And I'll defer to your own exploration as to how to do that. But that's how, first instance, I imported that Oscar the Grouch song. And you can import sounds like the meow. But the meow happens to come with Scratch.

(01:20:00)

Well, what else can we do? Well, in programming, in most languages, many languages, there's this other notion, which we didn't quite discuss earlier, that of a Boolean expression.

Note, on screen: slide #14

A Boolean expression—which is sort of inexorably linked with the notion of a condition—is an expression that is either true or false; named after, literally, a Mr. Boole, right? And Boolean expressions have this close relationship, as an aside, with the notion of zero and one; true and false. And true and false is how we'll spin them tonight.

These are some of the examples of Boolean expressions that Scratch supports. Again, constructs like these are found in other languages. Here is, in the domain of Scratch, what exists: "touching mouse-pointer?"

Think about what that means. Either a sprite is touching the mouse-pointer, the cursor, or it's not. It's either true or false; yes or no; on or off, right? The same ideas recurring again and again.

"mouse down?": Well, either the mouse is down: true; or it's not: false.

"<" (less than): Is the thing on the left less than the thing on the right? So we clearly can manipulate numbers with Scratch.

"and": another Boolean expression. Is this Boolean expression *and* this one true? If so, the whole thing is true. If one or both of them is *not* true, then the whole thing is false. So it's sort of an intuitive notion of what it means to be something *and* something.

Well, let's deploy these. To deploy them, we need to build on the notion of a condition, right? Because Boolean expressions, in and of themselves, not interesting.

But if we use them to govern what we're going to do, then they become a feature.

Well, here are some examples of other Control structures: "if" conditions.

Note, on screen: slide #15

And notice that, similar to what we had in our own sock-changing example, we had this notion of "if," or "if else," right? We could either do this, if something is true; else, do that.

If you wanted to have three conditions, with Scratch you can nest things. Because these blocks—in addition to snapping together, if they're logically consistent—they'll also grow to fit each other, so that you can fit as many statements inside of them as you want.

So even though this starts to indent a little misleadingly, this is Scratch's way of allowing you to do: "if something's true, do this; else, if something else is true, do that; else, if something else altogether is true, do this other thing."

And you can nest these things as many times as you want. Well, let's use these.

In hello4.scratch and hello5.scratch, we have these two examples.

Note, on screen: slide #16

On the left, tell me in a sentence, what will the program at the left do?

It's almost silly, right? Because we've hard-coded in something here.

STUDENT: (inaudible response)

DAVID MALAN: Perfect. It's silly because 1 is always less than 2, so it's sort of a weird use of an expression. But that's fine. The behavior of this program is very deterministic. It always meows.

Well, let's quickly whip this one up, rather than opening it. I'm going to make a new project. Get that Control structure: "when the green flag is clicked".

I need to then grab an "if" construct. So I'm going to drag this from down here. Notice you can scroll down to some others. "if"...and notice it will be snap right into place.

Now I need that Number-related block. So these are all under Numbers, and also they're color-coded, which helps to give you a little mental cue.

I want the less-than ("< ") one. Notice that, even though this is quite larger than the hole, everything is shaped to be similar in structure. So notice, if I hover over that, it gets highlighted. If I let go, it jumps right in there, and it grows quickly to fill it. It's the shape that's important.

Finally, I can just type in these white boxes "1" and "2". And then what I wanted to do is play a sound. I want to play the sound "meow". So I'm going say, "play sound meow".

Go ahead and click the green flag.

(cat sprite meows)

Right? Pretty deterministic. Let's change it.

"if 2 < 1" what's going to happen?

STUDENT: (inaudible response)

DAVID MALAN: Nothing. Completely uninteresting. But confirms what our intuition might say.

What about at right? What does this program do?

STUDENT: (inaudible response)

DAVID MALAN: Good. So this is a neat feature of Scratch. It allows for what are pseudorandom numbers, or random numbers. This is a powerful thing if you want your game to be interesting, right?

"Oscartime": not so interesting if the trash always fell in the same location, right? Even a child would probably get pretty bored with that. So even "Oscartime" uses some random numbers to say, "start the trash at location 20." Next time "start the trash at location 40." Next time "start the trash at location 55." And it just gives a bit of variability to the program, which tend to make games and other types of programs more interesting.

So at left, we've nested a few blocks, and it looks like, if it is true that Scratch picks a random number from one to ten, and that random number is less than six—that is it's one to five—then it's going to play a sound: the meow.

In other words, because there are five values that are less then six, pretty with much 50 percent probability do we meow; 50 percent of the time we won't meow. And we can sort of vet this by just trying it.

Let's quickly open hello... I think this is, uh, what did we say, five? Yup, hello5.scratch is in Lecture 11, as with everything else.

And let's go ahead and play. What's going to happen when I click Play?

STUDENT: (inaudible response)

DAVID MALAN: Excellent. It might; it might not.

It looks like it picked a number that was not less than six.

(cat sprite meows)

This time it did.

Didn't.

(cat sprite meows)

Did.

Didn't. Didn't. Didn't.

(cat sprite meows)

Did.

Now, you do this 100 times, you'll get roughly 50 yeses, 50 nos. Do it a thousand times, you'll get roughly 500 yeses, 500 nos, and so forth, assuming that the random number generation is accurate, which hopefully it is. And it tends to be in programs.

Randomization, incidentally, is used a lot in cryptography. And it would be a very bad thing in cryptography if your random-number generator always generated the same number, right? It's not that hard to guess, then, what a secret key might be.

All right, how about a loop? Let's now introduce the same construct we explored in a different domain earlier.

Note, on screen: slide #17

Scratch has such constructs as these: "forever". Turns out, sometimes an infinite loop is a useful thing, if you really want the program to do something forever. For instance, I always wanted trash to fall, again and again.

But it turns out, there are blocks—statements like "stop script"—that you could put inside of a forever loop, so that the loop does actually stop, even though it says at top, "do this forever".

Or you can say, "do something some number of times"; "repeat 10" times in this case.

How might we integrate these? Well, here are three new examples.

Note, on screen: slide #18

At far left is hello6.scratch. What's this thing going to do, in a sentence?

STUDENT: (inaudible response)

DAVID MALAN: Yup. Unlike the last example, which meowed, I think, three times, and waited a couple of seconds in between, this thing is going to meow forever, and only pause two seconds in between. But it's not going to stop until we hit the red stop sign in Scratch. I won't even bother playing that because it's pretty easy to just infer from the code.

What about the middle example, hello7.scratch? What is that going to do?

STUDENT: (inaudible response)

DAVID MALAN: Good. If the cursor's on top of the sprite, then it will meow; and otherwise, it won't. So already—even with just introducing a couple of new constructs, and even though this program is only six or so blocks long—notice if I open hello7.scratch, which is here, notice that, when I click the green flag, nothing happens.

But now...

(cat sprite meows)

...we sort of have...

(cat sprite meow)

...granted, simple, but a program that allows you to pet the cat.

And every time...

(cat sprite meows)

...you pet it, it meows at you.

So already, baby steps, granted. But we're already building more interactive programs, again, with terribly few blocks, with terribly straightforward syntax.

What about hello8.scratch, at far right?

STUDENT: (inaudible response)

DAVID MALAN: It's always going to play a sound every two seconds, but...

STUDENT: (inaudible response)

DAVID MALAN: Good. And I'll summarize by saying, if you pet the sprite this time, he does something a little different. So I'm going to hit Play.

(cat sprite meows)

Just going to meow a lot, every two seconds. If you try to pet this cat, though.

(cat sprite roars)

All right? Now get it goes back to meowing. Pet it again.

(cat sprite roars)

Okay. And so again, terribly few additional blocks. But, again, the program's beginning to get a little more interesting, right?

Slap a wallpaper on there, maybe another sprite, and we begin to have real interactivity, real animation, a real game.

What about Count.scratch, at far left?

Note, on screen: slide #19

What does this program seem to do?

STUDENT: (inaudible response)

DAVID MALAN: Keeps counting, and it keeps saying the current count. Well, let's just slap some jargon on here: "set counter to 1". What is "counter" probably? What kind of construct is that?

STUDENT: (inaudible response)

DAVID MALAN: A timer, but let's use the programming speak from tonight. This is an example of... ? I didn't call it x, y, or z. I called it counter, but what is it?

STUDENT: (inaudible response)

DAVID MALAN: It's a variable. And I'm initializing the variable; not to zero, like we did with socks_on_feet, but rather to one. And then "forever" I'm saying the current value of the counter, and that's Scratch's way of allowing you to do that, by tucking the variable inside of the "say" block.

I'm going to say it for one second; I'm going to wait one second; I'm going to change the counter by one, aka, ++; and then I'm going to repeat forever and ever.

If I open up count.scratch, what we have here is now a program that, if you look back now at the code, it should make more sense: simply has our cat counting, from one up to infinity.

(01:30:10)

Now, not terribly interesting in this context, but I clearly was using a counter in "Oscartime" for what feature?

STUDENT: (inaudible response)

DAVID MALAN: Counting the score. Every time a piece of trash was successfully dropped in, I "++'d" it; I changed counter by one.

What about the example at right, Hello9.scratch? What does this do now?

STUDENT: (inaudible response)

DAVID MALAN: Excellent. And so that's the value add here. Essentially, this is the same example as before, where I picked a random number and meowed only with 50 percent probability.

This program, by using the variable, and storing the random number in that variable, called "number," in this case—could have called it "*x*"; I called it "number" instead—it says the number just so that we can get some reinforcement as to why the cat meowed or didn't. It's just showing us the number.

So in Hello9.scratch, notice that we have this right here. Hello9.scratch. I'll hit Play. Six is bigger than five.

The next one is... oh, and this doesn't even have a loop. So it looks like I have to play this one again and again. Let's hit it again.

Nine is not less than six.

(cat sprite meows)

There we go. One through five was chosen.

(cat sprite meows)

One through five. Two.

(cat sprite meows)

Nine. Same program, but now a bit of feedback.

What about this guy?

Note, on screen: slide #19

Well, now, finally, we have a bit of animation, right? And with only introduction of a couple more blocks.

First let me open Move1.scratch. This is going to empower our sprite to start moving around, quite literally. And in moment we'll introduce a second sprite.

Here. I'll make this full-screen, and I'll click Play. So now, simple animation, but again, only takes a few blocks.

(cat sprite roars)

We have a cat that's moving around, and every time it hits the edge, not only do we play a sound, it also reflects, like a pool ball would, off the side of a pool table, and keeps going.

All right, how did we do this? Well, this is just... this program here: "forever" do the following. "if"…

(banging continues)

Okay, that's going to annoying.

(banging stops)

"if"… "forever" do the following. If you're touching the edge, well, what, intuitively do we want the cat to do? Play the sound called "bang", which I also took out of Scratch's arsenal of sounds.

"if on edge, bounce", which—this is sort of a cheat on Scratch's part. It's sort of a condition and a statement combined. But that's okay. It's sort of a little faster to implement with blocks like that.

Then it's going to "turn 15 degrees" and "move 2 steps", which is effectively like two pixels, both at the same time. "else" if the cat is not touching the edge, it's going to move just one step.

With that said, if I wanted to speed up this cat, which—this is the same script—what number could I change to speed up the cat's movement?

STUDENT: (inaudible response)

DAVID MALAN: Yeah, so let's change this "1" to "5", which means he's taking five steps at once. Hit Play again, and now he's really moving.

(banging continues, as cat hits wall)

Again, simple change. All right, well, let's introduce a second sprite.

Note, on screen: slide #19

All right now it starts to get fun, if you don't just have a cat moving around.

But let's have, for instance, like a cat chasing the bird. Well, let's look at the bird first. The bird is at your left. And this is going to be one script belonging to one sprite; the other script will belong to the other sprite. So they operate sort of independently.

And at this point, we can introduce the notion of a "thread." A thread is a common construct in many operating systems, and really, programs. And a thread refers to the ability of a program to do two things at once. If a program is multithreaded, it can do two things at once.

So really, we can think of these scripts as threads. And if a program has two or more scripts, both of which start running when you click the green flag, it's as though your program has two or more threads, because, by definition, they execute simultaneously.

In the real world—or in the real virtual world—when you go to the File menu in Microsoft Word, and say "Print", usually you immediately can get right back to typing, even though it might take five minutes to print out your twenty-page essay. Well, that's because Microsoft Word is multithreaded. It has at least two threads, one of which allows you to keep typing your essay; the other thread allows it to go off and print.

And that's a bit of a white lie, because there's also what are called print spoolers built into printers, and so forth, these days. But, for the most part, most every program these days that does a lot things seemingly at once, is multithreaded.

So is Scratch multithreaded, and you use threads by just writing multiple scripts.

So this one at left—it's starting to get a little bigger. But, again, still pretty self-explanatory.

The Stage in Scratch is laid out, like in Photoshop, in terms of pixels. And you'll be able to experiment, either at home or in next week's Section, with exactly where (0, 0) is, and where (100, 0) is. It's pretty much a Cartesian plane, with *x* coordinates and *y* coordinates.

So what this left script for the bird is doing is it's moving the bird, at the moment the green flag is clicked, to location (-150, 150).

So we'll see in a moment what that means for the bird, where he appears. But it's essentially somewhat to the left and somewhat down, as we'll see.

Then it points it in the direction 45 degrees. So it's angled in a certain direction. And then it "forever" does this: if touching the edge, "bounce". Okay?

And I realize, this is a bit of a bug, since I wasn't thinking, when I did this. It's sort of redundant to say, "if touching edge" and then "if on edge, bounce". So we'll fix this in a moment. But that's simply not necessary. Scratch now supports this new blue block that we described earlier, "if on edge, bounce".

But "if not touching cat?" "move 3 steps". Well, let's see what that really means. Let's turn our attention to the cat's script for a moment.

The cat is going to start, notice, in a different location, but still kind of close: (-160, -160) this time. It's going to point in what direction? A "random" direction, from "91 to 179", which I just sort of chose arbitrarily so that it would start at an angle, rather than going straight up or straight across.

And then it "forever" does this: if it's touching the bird, play the roar sound, and then stop the script. You're all done.

Otherwise, point towards the bird and move 1 step.

Given this definition, which of these sprites—cat or bird—is moving slower?

STUDENT: (inaudible response)

DAVID MALAN: All right, so the cat seems to be moving slower, because he only takes one step each iteration, whereas the bird seems to take three steps.

So let's see what the implication is.

I'm going to open Move2.scratch. And that again is in the Lecture 11 directory, which is downloadable from the URL that's on all of the slides.

Move2.scratch is here. I'll leave the program alone. But just appreciate, if you would, the redundancy in the bird's script.

And now notice the feature I've just sort of revealed by clicking around. Right now the bird is highlighted, which means I see the bird's scripts, or in this case, script.

If I click the cat, I see the cat's script. And so here we have the notion of two different threads and two different sprites. But it's really no different from before. I just have multiple sprites.

And you can create new sprites simply by clicking the appropriate icons here. Like the cat icon: click it—you'll get a new cat, which is the default costume for a sprite. And then you can change the costume by playing around, for instance, up here.

Again, a lot of the program is designed to be very intuitive. And we're only going to give you a sense of some of the features, because part of the fun in it is sort of exploring exactly what can you do and how can do it.

But it's the programmatic constructs that tonight are the most compelling for us.

Let's go ahead and run this. I'll make it full-screen. Click the green flag.

Notice the bird started in the left of the screen. Now he's bouncing as before. Cat keeps...

(cat sprite roars)

...pointing toward him. And as soon the cat touches the bird, the cat does that roar.

Same thing again.

Notice, if I wanted to make this slightly more interesting, I could have the bird start in a random location, too.

(cat sprite roars)

And he caught him again.

Well, what more can we do? Hello10.scratch.

Note, on screen: slide #22

Well, this is kind of a neat way to show you how to use variables to maintain what's called "state" in a program.

Variables allow you to keep track of sort of the state of your world. This program quite simply is one that has a cat meowing, but it allows me to mute the cat, or re-enable sound just by clicking the spacebar.

So let's put this into context first. Let me open Hello10.scratch, in Lecture 11, the last of our Hello examples.

I'm going to click the green flag.

(cat sprite meows)

There we go, green flag.

(cat sprite meows)

Okay, but notice, on your own source code, if you want to look down, there's that "if" condition, that said, if space pressed, change the value of that variable.

I'm going to hit the spacebar. And it goes quiet because I've changed the variable, called muted, this time to one. And notice that the part of the loop in this program that says, play meow: yes or no, first says, what's the value of muted? Does it equal one? Then don't play anything. If it equals zero, do play something.

Let's hit it again.

(cat sprite meows)

Now it comes back on.

So the program was still running, but every time the loop iterated, notice that "if muted = 0" is zero, that is it's not muted, play the sound. But if it is equal to something other than zero, for instance one, than it doesn't play anything.

(01:40:02)

But how is this happening? It seems to have two different scripts. Well, threads are useful. So that even conceptually you can divide your program into different sort of modules, one of which handles a certain task; the other of which handles the other task.

So at left, these two scripts are both associated with the same cat sprite. So this sprite just has two scripts.

The one at the left just meows forever, every couple of seconds, but only if the variable called muted is zero.

Meanwhile, the thread at right simply spends its entire life listening... or watching for whether the user hits the spacebar. And the moment the user hits the spacebar, because that script at right is doing this forever, it's going to change at that moment in time the value of "muted" to "1," and that is going to have an effect on the script at left, because the script at left is using that same puzzle piece, that same variable.

And so what's neat about Scratch is that, not only can you have variables per sprite, it turns out—and if you decide to explore this—you can also have what are called "global variables," which are simply variables that not only can one sprite use, but all of your sprites can use.

And in this way can your sprite sort of interact and share information. In fact, one of the ways that "Oscartime" works is that, effectively, when you do drag a piece of trash to the trashcan, that sprite signals to Oscar that it is now in the trashcan, and therefore Oscar should increment his own variable for the score.

So you can have some neat interactions. And I will say that "Oscartime" is sort of a complexity that we don't expect for your Problem Set 8. It's meant to give you a sense of what you could do with Scratch, if you spent eight hours on a Friday night, listening to that damn song on repeat for at least six of those hours. Can't stand that song anymore, not that it was on my playlist earlier.

But here's another example, which I pretty much borrowed from one of the programs that an MIT teaching fellow made. It comes with Scratch.

Note, on screen: slide #23

I simply changed the name, changed the icon, and a couple of the lines of code. But if you open up this, at your leisure; perhaps if you find yourself particularly frustrated at some point soon, you can open David.scratch. I'll make this full-screen.

This actually—just take note—has three sprites: a left glove, a right glove, and me. And it's got a Stage, which is like our wallpaper, in this case.

And, again, some of the details here, like, how do I put a wallpaper on the stage, we'll leave to you to explore, to figure out, a la our Google Earth example. Or again, next week's Sections on Wednesday and Saturday will introduce Scratch in a hands-on sense, with you in front of computers.

Let's make this full-screen. Click the green flag. And this game, if you read through the source, happens to be played such that, with the arrow keys, I can move left and right. And then if I hit "forward," I can start punching myself. And the more I punch myself, I change colors, getting greener, sicker and sicker.

Notice that I'm saying things. Turning even greener, a little blue now. I can move over. Hit him with the left. Hit him with the right. The left. He's moving fast. So there's clearly some randomization, right? There's some picking of random numbers. Almost...

So this is a little Internet speak. At Dan's request, I got a little hipper with this example. And it says… It's not a typo. "I got..." How do we pronounce? "Pwned?" "I got pwned!" which... *P* on your keyboard is very close to the letter *O*. So in Internet-speak, saying "pwned" is sort of the cool way of saying "I got owned!" which is like saying "I got beat up, or beaten, or knocked out," in this case.

So hopefully now I'm all that cooler for having changed the text to "I got pwned!" Anyhow, Google "pwned" or look it up on Wikipedia for the history of such things.

Well, this we won't spend much time on, just because it begins to get a little scary. But realize that that whole game, if it can be called that, was the result of just—what—one, two, three, four, five scripts, three of which are terribly short; one of which is pretty long.

But again, if you read through it, perhaps after tonight, you'll see that, again, it's just implementing the logic of the game: move David back and forth somewhat randomly. And if he ever gets hit with one of the other sprites; that is, one sprite is touching another, which is one of the blocks under the Sensing category, well, then change David's color—change the color, really—of David's costume.

My face is just a costume on what might otherwise look like a cat.

Well, there's this one other feature of Scratch, and really, of a lot of programming languages in general, which is this ability to signal from one thread to another.

And threads can talk to one another, effectively, either by using some global variable, such that one changes a variable and the other watches that variable, as in our muting example. Or you can send a message; you can trigger an event, throw an event that another sprite or another script listens for, or waits for.

So this is a very simple example, Marco.scratch, that demonstrates the use of what's called "event handling." And this is common in most programming languages that interact with users.

I'm going to open Marco.scratch. And notice that all this program does is the following.

I hit the spacebar, and then she responds "Polo". So notice, let's play it again. The moment I hit the spacebar, the boy says, "Marco". Hands off the keyboard. The girl later says, "Polo".

Now we could certainly just use a timer of sorts, right? Wait two seconds and then say "Polo".

But notice that these two characters are two different sprites, each of which has a different script.

I'm going to go back to the PowerPoint so we can see them simultaneously.

Note, on screen: slide #24

It's the same stuff, but notice that the boy simply waits forever, until I press the spacebar. That is, forever do this. If the spacebar is pressed, say "Marco" for two seconds. That's pretty obvious. But then "broadcast" what's called an "event." This is sort of like a message, a whisper, if you will, between sprites.

Meanwhile, the girl is designed not to start acting when you click the green flag, but when she receives "event"; that is, when she hears the event whispered to her. And when she receives that event, she says, "Polo" for two seconds.

And we offer this as a very simple example, but one that hopefully will offer you an ability to solve some problem, perhaps, that you might come across in your own implementation of your own project for Problem Set 8, as, you know, "How can I get two sprites to talk to one another?" Well, you might be able to do so by way of this event handling.

You've already seen "Oscartime." In Problem Set 8, what you'll be tasked with is—in addition to debugging a little program; and teaching us, the staff, how to make peanut butter and jelly sandwiches—you will quite simply be challenged to implement something in Scratch.

And you saw a glimpse of the gallery from our Summer students, what they implemented. I promised to show one of those as well.

We looked at the soccer game, which was very interesting in a quite impressive program that that student submitted.

I wanted to give you an example of an animation as well, that's not interactive, but sort of tells a story.

This, too, is going to be in this same directory that we've been playing with. I'm going to make it full-screen, and I'm going to hit Play.

There's not much sound.

That's a bug, right? We're not at the end yet. But it's okay.

(fairytale animation plays)

You can make sprites zoom in, as this student happened to do; that is, grow in size.

Dragon sprite appeared at left.

(dragon sprite roars)

Knight appears at right.

(princess sprite screams)

One of the neat things is that she used a... She used a pseudorandom number generator: pick a number between one and ten, and only 50 percent of the time does the princess actually come back to the knight. So that's kind of a cute little trick.

One other program that I'll demonstrate, submitted by another student. This I would say was sort of on the extreme end of "Wow, can't believe a student implemented this so quickly!"

This is one of those, um... oh, what's the name of the popular arcade game that looks like... "Street Fighter." There's another one that's very popular.

STUDENT: (inaudible response)

DAVID MALAN: Sorry?

STUDENT: (inaudible response)

DAVID MALAN: "Mortal Combat" this reminds me of.

So I'm going to hit Play up here. Okay.

Press C for control, so I have some instructions. Notice I can move around. I can jump by hitting Up. And then if I hit the spacebar, notice I can strike the demon with lightning. But I have a sort of light force. You'll notice at bottom left… I'm really getting pwned here. Oh, and I'm almost... Oh, I have failed. My life force is gone.

So this, too, was particularly impressive. This is a student who had a good number of scripts, a good number of sprites.

But what you have to appreciate, especially if you're somewhat daunted by the complexity of some of these programs—for instance, this one; even some of the ones in Scratch's sample directory—no one sits down and writes these things all at once.

The first thing, for instance, I did with "Oscartime," was, one, I pulled up Photoshop, and I made the icon. And I downloaded a picture of Oscar from the Web, and so forth. Did the easy stuff, right? It's not quite programming yet.

(01:50:00)

But then I simply said, "All right, what's the first step of this program? I minimally need to get trash to start falling." So I found a costume that looked like a ball of trash, and I simply implemented a sprite with a script that chose a random location at top; was originally invisible; then eventually made itself visible. And then just started moving, by moving one step, waiting half a second—a millisecond, for instance—then moving again.

And once I was comfortable that, "All right, I got the trash falling. Now let me try to introduce the soundtrack with another sprite, and have the song start playing. And now let me go back to that original sprite, and introduce a wait timer." And I looked at my watch, and I said, "All right, I want the trash to start falling after twelve seconds. So I added a wait statement inside of this block for the trash, and said, "All right, don't start falling immediately. Wait twelve seconds." And now I had the song synchronized with the first piece of trash, and so forth.

So when you approach your Scratch problem set, Problem Set 8, don't try to bite off the whole thing at once: baby steps, not because it's necessarily hard, but because you'll drive yourself nuts biting off too much initially.

And, plus, this is a very representative process, when programming. When people sit down to write real programs in Java, C++, C#, they don’t sit down and write the program and then go test it. Baby steps.

Write part of it that's very easy to debug, to find mistakes in, to refine. Then move on to something else. And we'll leave it to you to decide what parts of your program make sense to bite off first.

But part of the challenge, part of the fun in the process will be figuring out on your own, "How do I go about implementing my algorithms? How do I go about implementing my art, animation, or game?"

So good luck with that. You have almost a month to do it.

We will see you next week for Exam 2, and a little something special.

(end)

(01:51:48)