Programming Methodology-Lecture18

Instructor (Mehran Sahami): So welcome back to yet another fun filled exciting day of CS106A. I'm feeling a little bit more relaxed. Are you feeling a little bit more relaxed? A few people are nodding. A few people are like, "Yeah, Ron, we took the midterm in your class, but we have like six other classes to deal with. You can feel relaxed, but we don't." But don't worry, things in this class are just – it's all downhill after this. It's just a good time.

So a few quick announcements before we jump into things. One of which is that there's four handouts, and if you didn't get the handouts – if they weren't there when you came in, they're all in the back now. You can pick them up after class. So you don't need to pick them up right now. You can pick them up after class.

The midterms will also be back after class. I'll show the performance of the whole class. I was very pleased. It was just – it was a heartwarming kind of thing to just see so many people doing so well. But you'll get them back after class. So you'll get them today.

You – one of the handouts also solutions for the midterm, and so you can – if you got anything wrong you can compare your answers with the solutions. One thing to also keep in mind is that the actual solutions we were looking for are shorter than the solutions that I give you. The solutions that I give you have comments. For one of the problems I actually gave you two different ways of doing it just so you see different approaches. But you weren't expected to actually write that much code. All we wanted was sort of the code without the comments, which is actually pretty slim if you consider how much code there is there without comments. But you'll get those back after class.

And then as you hopefully know, assignment number four is due today, and assignment number five, you're next handout – one of your handouts is going out. For assignment number five you will do yet another game because it's just so much fun to do games. And this time we're gonna be testing your abilities with – we're not testing your abilities. That's an awful way of saying it. We're going to be watching your abilities in the use of arrays flower in terms of designing another game. Okay.

So with that said, it's time to see how much pain we actually caused on you assignment number four. Hopefully not too much pain, although, I did hear there was quite a few people on [inaudible] on the wee hours of the night last night.

So anyone in the zero to two hour category? Was that actually someone back there? No. Just joking – yeah, kind of funny. Two to four hours? All right. A few folks. Good to see. Four to six hours? Oh, nice. Six to eight? Pretty large contingent there in the old six to eight category. Eight to ten? Wow, also pretty large contingent in the eight to ten category. Ten to 12? All right. We begin to see the drop off at ten to 12. Twelve to 14? Few folks at 12 to 14. Fourteen to 16? Sixteen to 18? Eighteen plus? Taking a late day? That's – yeah. Alrighty.

Once again – it happens every time. I don't make this up, right? You saw people raising their hand. The world is just – it's amazingly normal. It really is. There are two things the world is – it's linear, and it's normal, not necessarily at the same time. So again, ten hours, and less life is good. Even if you're in the ten to 12, you're still in pretty good shape. I didn't see actually too many people up in this range, which is good to see.

If you're in the 12 to 14, it's still not very bad, right? It's just some – there might have a bug or some particular thing that happened that caused a little extra time. If there was some conceptual thing that did not get cleared up as a result of actually doing the assignment, that's the whole point, right? Some times you might run into a roadblock, but the point is once you get through that roadblock, that concept is cleared up for you. That's the real point of doing assignments as opposed to just reading the book.

Reading the book's easy, and a matter of fact, it's surprisingly easy to read code, and think that you could write it, right? That's what most people do. They look at code, and they're like, "Oh, I could've written that." Right? It's like Jackson Pollock painting. You look at that, and you're like, "I could've painted that." Right? But there's a difference, right? One, he painted it first, and that's why it's worth like $20 million, and two, trying to actually do the same thing later on requires a different creativity on your part than just looking at what someone else did. So it's important to not just read code, and understand it, but to be able to write code, and understand it, which is the whole point of the assignments. Alrighty.

So with that said, time for some midterms statistics. So here is the stats on the midterm, and that's just a beautiful thing to see. Right? This little hump over here – I should've just tacked on to this, right? Then we would've actually had like a little inverse Poisson distribution that you would've said, "But, Marron, it's not normal." And in fact, you can approximate a Poisson to the normal, so it is.

But the important thing is that in fact the class did very well. There were six perfect scores on the exam, which is why I broke out the 90's as a separate category, which it just warms the cockles of my heart. What the cockles are, I'm not sure, but it warms them. And a lot of people who did very well, right? A lot of people in the 70 or higher category. Nice to see a very large contingent in the class in the 80 plus category.

And so the stats – this is out of 90, right. So this in not percentages, but this is out of 90 points. The mean was a 73.2. Median was a 77, which percentage wise translates into almost 85 percent. And then the standard deviation was 13.6 points. So if you're kind of over in this range over here, good times, happy days, life is good.

If you're kind of in this range over here, you want to make sure conceptually things get clarified. You want to look at your exam, and figure out whatever problems there were, look at the solutions, get them clarified, get the issues clarified in your mind. If they're not clear, talk to your section leader this week, talk to me, talk to the TA to get any concepts cleared up.

If you're kind of in this range or lower, it's – to be honest, it's time to worry a little bit, and you need to really –think of the midterm not as just, "Oh, my God, my grade is ruined," because the point of having midterms, at least as far as I'm concerned, is not just grades. It's great if everyone does well, and we just give more A's, and life is good. But really it's a diagnostic for you. It's a chance for you to get a realistic calibration of how you're doing relative to the rest of the class, relative to what we expect you to know.

And so if you find yourself from that calibration falling into a range that worries you a little bit, that you shouldn't think of as, "Oh, it's just time to pull your hair out, and panic." All that really is is an indication that you need to come clarify some concepts. And maybe they're clear for you, and something bad just happened during the exam, like an earthquake. Or maybe it's some conceptual issue that needs to be cleared up, and this is a chance while we're still well before the final exam, and well before several other assignments in the class to get that clarified for you. So think of it as a diagnostic. Okay.

So any questions about anything related to the midterm before we sort of push off into our next great topic in computer science? But I was just very pleased. It just made me happy. It made me think like you had the easy button with you, and you just saw the exam, and you were like, [Easy Button] That was easy. Yeah.

It's just like – I got this thing – well, I got – I'll tell you the story. I bought this thing like a year ago, and I was like, "Oh, this will be great for class." And then it sat on a counter for a year. So now I'm happy that I can actually use it. And it is not particularly branded because I cut that off just in case someone wants to sue. All right.

So with that said it's time to actually get into our next great topic in computer science. So I'll just leave this up here if you want to write down these numbers for whatever reason. But to wrap up a little bit about one of our last great topics, which was arrays, and specifically multi-dimensional arrays. And I'll just call them multi-dim arrays. And you've seen a couple examples of these right now, but I want to – already, right, in the last couple classes you saw some examples of multidimensional arrays. But I want to show you yet another example to show in a very nuts, and bolts kind of way how funky things get when you have arrays of arrays, and you pass them around as parameters in a program – just so you see yet another example to drive the point home.

So let's say I was gonna write a program to, oh, keep track of midterm, and final scores, for example. And let's say the class was smaller than this. So I had at most 100 students because I was teaching some class like basket weaving or something, and it's just small. It's cuddly. We all hold hands. So I have some two-dimensional array that I want to have because I want to have midterm scores, and final scores, but I want to keep them separate. So I'm gonna have one array of midterm scores, and another array of final exam scores.

But in some sense what I really have is a two-dimensional array that's two by the number of scores that I have as a maximum. So let's say 100 is the maximum number of scores I'm gonna have. So I'm gonna a two by 100 array, and I'll call this scores, and I'm just

gonna create this by basically saying new int. And you've seen the syntax before. Two – and this is 100. And these would probably be constants that I would have in a program, but just to save space, and save time, I'm not – you actually declare public static final int., and saw max size 100. So I'm just gonna put 100 in there.

And as we talked about a little bit before, some particular entry of this grid, right – because this is just a two-dimensional grid – like score zero comma zero is just – this is just a single int. basically. But then we talked about this funky thing where we said, "Hey, you know what? A multi-dimensional array is actually an array of arrays." So a two-dimensional array's an array of arrays. A three-dimensional array is an array of arrays of arrays, right?

So we could actually think about, "Well, what happens if I only specify one of the dimensions of the array?" So what do I get if I say, for example, scores sub zero, right? That's not one entry, so it's not an integer. That's in fact a whole array, right? The way to thing about this is I have two arrays that have size 100. All right. So we have a little break in there. And so score sub zero is this array, and score sub one is this array, and scores is kind of this array of arrays, right? It's really an array that has two elements. So this is element No. 1 over here in the heavy chalk line of the array or element No. 0 – it's the first element in here is element number one. Okay.

So this thing – the type of this thing is really an integer array. And that's the funky thing to kind of think about. Okay. So score sub zero behaves in your program just like if you had declared a one-dimensional array, which means you can pass score sub zero to a function that expects a one-dimensional array, and that's perfectly fine. What it's doing is passing this element of the scores array of arrays, which happens to be a whole array of integers in its self. Okay. So any questions conceptually about that? All right.

So why don't we look at a little code that actually drives that point home. All right. So we'll put this away. Bye-bye midterm scores. Don't save. And we'll come to our friend the Eclipse, and we'll say, "Hey, Eclipse –" oh, we'll get to Roulette in just a moment. Don't worry. It's all about gambling in this class. All right. So it's not really gambling. All right.

So I'll tell you we have a program that wants to keep track of test scores. So here what I'm gonna do is I'm gonna have a program test scores, again, we're just gonna set the font to be large. It's not a big deal. And I'm gonna ask the user for the number of scores. So I'm gonna ask the user for number of scores, and whatever the user gives me is num. scores. I'm going to create my scores array to be a new array of integers that's two by however many scores you have.

And you should see this right now, and you should go, "But, Marron, you're missing the declaration of scores, right? This creates a new array of arrays, but where's the declaration of scores?" Ah, instance variable. So I want to show you an example of where we have an instance variable that's an array of arrays. So here I declare the instance variable. This just tells me that score's is going to be a two-dimensional array.

It is not yet created the two-dimensional array. It is just set – I'm gonna have this thing called scores that gonna be a two-dimensional array, I still need to create it. So when my program runs up here, I ask the user for the number of scores, and I actually create the two-dimensional array. I don't need to declare scores because scores was declared as an instance variable or an ivar, so it's already declared. I need to create all the space for the all the elements of that array. Any questions about that? Hopefully that's clear. All right.

So first thing I'm gonna do is I'm gonna initialize my scores, and here – and its scores. All this is gonna do is have two for loops that are gonna go through, and set all the scores in the grid to be zero to begin with. Now, I told you before, when you declare an array – when you create an array, the values of the array get initialized to be the default value for that type. And the default value for integers is zero. So you might say, "But, Marron, they're already zeros anyway, so why are you doing this?" I'm just doing this to kind of exemplify the point. I could put in a one here, and just say, "Hey, everyone starts with a one on the exam." Right? But we'll just say it's zero.

How do I actually think about this two-dimensional array? Well, first of all, I want to think about the rows of the array, and then for every row it's got a certain number of entries or essentially the number of columns in the grid. So how many rows are there in the array? If I look at the two-dimensional grid scores, and I say scores dot length, and scores is an array of arrays, it tells me what is the size of the first dimension, right? So if scores is a two by 100 array, what's the size of the first dimension? It's two.

So scores dot length – the way you want to think about it is this is scores. How big is scores? Scores actually is of size two. Each of the elements of scores of size 100, but if each of the individual entries of scores – score sub zero, and score sub one – is an array. So there's only two arrays inside scores. So scores dot length, back over here in my program, will in this case have the value too. Okay. So that's gonna be my outer loop. My outer loop's gonna say, "I want you to loop over however many rows you have in your grid, " is the way to think about it.

For every row, how many entries do you have in that row? What I'm gonna do is I'm gonna say, "Hey, all the rows have the same size, so I'm just gonna arbitrarily pick the first row because I know there's always gonna be one row. If I knew there was always going to be ten rows, I could say, "Score sub nine." But for programmer, they're just like, "Why are you doing score sub nine?" So the general idiom – the way we using do it, is just score sub zero because you always know that there is a first – always a first entry in an array – or in an array of arrays.

So score sub zero, what is that? Score sub zero I just told you is a whole array of integers, right? It's this thing right here. So its length is 100 because there's 100 integers in it. So in this for loop we loop through 100 elements, and so all we're gonna do inside here is we're gonna set scores I sub J. I's gonna very from zero to one because it's actually gonna count from one to two, but not include two. And J's gonna vary from zero to 99. And so it's – we're gonna set all the elements of the grid to be zero. Any questions about that?

The syntax just kind of looks funky because here we have scores without a sub script on it. Here we have scores with a subscript on it. In both cases we're taking the length. And the important thing to think of conceptually is it's an array of arrays. So when we just take the first length, we're saying, "Hey, how many sub arrays do you contain?" It says, "Hey, I contain two arrays. I'm an array of arrays." And when we take the length of the first sub array, we get the actual number of columns for a grid. Okay. And you can generalize this to three, and four dimensional arrays, but for this class really all you're gonna need to worry about is two-dimensional arrays at most. Okay.

So that's gonna initialize all the grid to be zero. So that's what an int. scores is gonna do. And then we're gonna write out score sub zero before we do any incrementing – action increment all the scores inside because it's just good to give everyone points.

But before the increment, we're gonna print out the scores array at score sub zero. So if we look at print list – right – all print list does is – this is a really simple thing. It just gets passed an array of integers. It goes through all the elements of that array, right? So we just are calling the array list in this case, so it's just counting up from zero up to the length of the list, and just writing out all the individual elements, one on each line. So it's a trivial thing to write. You've seen this a whole bunch of times with arrays so far.

But what are we passing to this guy, right? This guy's expecting a one-dimensional array. What we're passing is score sub zero here, which means what we're passing to it is just this single entry from the scores grid. We're passing a sub array of the scores grid. We're passing the very first array that's at score sub zero, which means score sub zero here is a whole array of integers, and that's the type this thing's expecting, so everything's perfectly fine. It may seem weird that we're sort of taking this grid, and dissecting it into its individual rows, but the important concept – and I'll say it once again – each one of those individual rows is just an array in itself. Okay.

So last thing I want to do besides just printing out the scores, I want to go through, and increment the score list. Now, here's something that's funky. An increment score list – I'm gonna pass in one of the sub arrays. I'm gonna pass in score sub zero.

What is increment, and score list do? It's expecting an array of integers. It just has a loop through that list, and it's gonna go through, and just add one to every entry in that list, right? Very standard thing you would think of doing maybe with some array of numbers. You just want to go, and add one to it. And that's all this method is doing. The only thing that's funky about it is when we call that method, we're passing it a portion of the scores grid, which is just the first array in that scores grid. Okay.

So any questions about any of this? If this is all making sense nod your head. Excellent. If it's not making sense shake your head. Feel no qualms to shake your head. Good, good times.

So let's run this. We'll be happy. We'll just make sure it's working. I'll give it a small number of scores because it'll be like a tiny wee class. How many – I should just ask.

How many people are in the smallest class that you happen to be in not counting the instructor? Anyone lower than five? One. How many people are in that class?

Student: Four.

Instructor (Mehran Sahami): Four. That's less than five. And as a matter of fact, that's all you need for candy because one is – one here – and that's amazing, right? It's just a class of four people. That's a good time. Sometimes – all right.

So we'll say the number of scores is four just to emulate your class, and everyone started with a zero, and then they all got a one, which in a class of four people probably means 100 percent. All right. So that's all it's doing, right? And all of this is in terms of score sub zero. Score sub one has the value zero. It still maintains the value zero. We never touched it when we were doing the incrementing. So any questions about that? No. No. No. All right.

So besides just doing life in this array way, you can also do things in the array list way. Okay. So we talked about arrays, and array lists. So here I'm just gonna have a single array of – array list of scores. And what I'm gonna do here, just so you can see the syntax, is again, I'm gonna have a private instance variable that's my array list scores that's an array list of integers. I only have one, and I'm not asking the user how big it needs to be, right, because array lists are dynamically sized. It doesn't matter. As the user keeps giving me scores, it's just gonna get bigger, and bigger. So I don't need to have the size a priori.

So in my instance variable, I actually not only declare scores, but I actually create the array list. So I create a new array list. It's empty to begin with. I can just add stuff to it. Not a whole lot of excitement going on there.

What do I do? I'm still gonna ask the user for the number of scores, but here when I'm initializing my scores, I don't need to worry about creating an array of a particular size. Here I just say, "Hey, you gave me some number of scores. You want me to initialize this array with how ever many scores you gave me." So I'm just gonna go, and do a for loop, and add that many elements to the array list because every time I add, the array list just automatically grows in size to be able to store all the scores. And I'm gonna store zeros.

Well, as you know a priori, right, zero is just an int. It is not an integer. So in Java 5.0, and later because life is happy for us now, we get this automatic boxing, which means the computer sees the zero, and goes, "Oh, that zero's an int." Your array list stores integers, right? That's the parameterized type of your array list. So I'm automatically gonna box up your little zero like you're at the integer restaurant, right, and it's like, "Oh, I got this O." And it's like, "Yeah. Yeah. I can't eat the O, let me box it up for you as this thing called an integer, and then you can have it." And then it says, "Oh, I can add an integer to scores." And it just takes care of that for you. Okay.

So when it scores it just goes through, and adds that many zeros to the list, and then we can write out the scores before we increment, and we'll print list our scores. And all print

list does is – you've actually saw this exact same code last time. It has an array list. We don't need to specify the parameterized type here. This will just work on any kind of array list. It goes through up to the size of the array, and it gets each of the individual elements, and prints them out. Okay.

So all we're gonna do now – the interesting part is to increment the score list. Here again, we're gonna go through scores up to the size of scores, but the syntax looks different. The syntax is actually a little bulkier than our array syntax, so we could just say, "Scores sub, you know, I plus/equals one." Right? To add one or plus plus if we just wanted to add one.

Here the funky thing if we want to set a score is inside the scores list we have to use the set method. There's two parameters we sent to the set method. The first is the index. So we're gonna say, "Set at location I." So whatever the old value was, we don't care. We're just gonna slam something new over it, but set something at val – in place I.

What are we gonna place at location I? We're going to get whatever was previously at location I, which is an integer, and we're gonna add one to it, which means if I have this thing that's an integer, and I want to add one to it, I take that integer, I un-box it to become an int. This happens for you automatically. So I get the int., which is zero, which is what I put in there before. I add one to it. So now I have the int. one, and now I want to store it back into something that's stores integers, so it automatically gets boxed up for you as this integer, and gets stored back.

So the syntax looks a little bulky, and that's really the point of showing this to you. It – sometimes using array lists can actually be a bulky thing to do. But if we go ahead, and run this – do te do te do – we're running. We're running. Life is good. It chugs along. I got to get a faster Mac. Alrighty.

Test scores with array lists – and it's gonna ask me for the number of scores. We'll say three this time because someone in your class dropped. And we had three zeros that got entered to begin with, and then we went through, and for each one of those zeros we sort of took the zero, added one to it, and stuffed it back in. So we got three ones. Not a whole lot of excitement going on there, but you see the difference.

So when you see these two things one natural question that kind of comes up in your head is, "Okay, Marron, I sort of see two ways to do this. Which one should I use? Should I use an array list or should I use an array? And as a matter of fact you just told me on this next assignment I'm gonna be working my array muscles. So what's a good idea to think about?" Okay.

And here are the pros, and cons. So if we think of array lists – and I'll just give you the pros, and cons relative to array lists versus arrays. The pros of an array list, right, are that at – the biggest pro is that it has this dynamic resizing. So if I need to add more elements – if I don't know the size a priori, this is a great way to go because it automatically resizes. Okay. And the other pro that it has is there is a bunch of high-level operations

that automatically supports things like contains, right? So you can just see if an array list contains a particular element or not. So it has other operations – I'll just say ops. that are provided for you. Right? Which is a nice thing to have – so you don't need to worry about going, and finding some element in array.

It does have some cons though, and the cons are that it's less efficient in terms of how the computer actually deals with an array list versus an array than an array. So it's less efficient than an array. The other con that it has is its syntax can get kind of bulky. Syntax is bulky because if a lot – one thing that I really want to do often is if I want to have some array, I want to make some modifications to values in that array. Well, in an array I just say, "Hey, you know, I have score sub three, and I stick in some value, and life is good." Or I say, "Score sub three plus/equals five, and I add five to it, and life is good."

Here the syntax is bulky because I need to say, "Get the old value, do the modification to it, and then set it as the new value." And so I'm really doing two method calls every time I want to get, and set a meth – a value, and that's just bulky. And bulk means it becomes more error prone.

The other thing is in the pre-Java 5.0 world. So we'll just say pre-5.0, which would you sad, and weep, and any time any – or if you run into any friends that are using a version of Java earlier than 5.0, you should just pat them on the shoulder, and take a moment of silence with them, and then like – I wouldn't say slap them around, but I would say, "Encourage them strongly to get a newer version of Java." Pre-5.0 – this was horrendously bulky to use because all of that boxing/un-boxing stuff didn't happen automatically, and you had to worry about, "Oh, I need to box, and I need to un-box." It was a bad time.

The main difference – and you might say, "Yeah, well, you know, I don't care about efficiency, right, Marron? I got a whole bunch of memory, and my computer's fast. And yeah, bulky syntax, I can deal with that because I cut, and paste anyway. And I'm using Java 5.0 or later, so I don't really care about any of this, man. I'm just – I'm going the easy route."

And the real big difference – the thing to keep in mind for one versus the other is if you know the size of your array beforehand – if you have a fixed size, almost always go with a regular array, not an array list. Keep that in your mind when you're thinking about the program that you're going to be writing. Fixed size – there's a lot of things for which you know they will have a fixed size or some maximum size. Hum. Let's take a moment to think about that. Fixed size – your next program – array. Good? All right. So just in case it wasn't clear. All right.

So now, with that said, it's time to actually move onto the main topic for today. So any questions about array versus array list? Hopefully that should be clarified. All right. And you should just be thinking right now, "Oh, Marron, [Easy Button] That was easy." Can you even hear that? [Easy Button] "That was easy." Yeah. That's what I'm talking about.

That was easy broadcast in stereo. All right – which means it's easy from two different sides.

So now, the thing to think about is our friend debugging. All right. Which is something you've done this whole time, and you're like, "But, Marron, I've been doing debugging this whole time. How can we have a whole lecture on debugging?" Well, because it's time to think a little bit about that zen, and the art of debugging, and actually a bunch of practical tools I'm gonna show you to do debugging. Okay.

The first thing to think about debugging is when you are a computer programmer or software engineer or in the larger sense computer scientist, there are different roles you play when you're building software. There's a portion of your time which is spent on designing software, and here you're really like an architect. You're kind of looking at big pieces, and how they fit together, and drawing schematics, and figuring out what your sub pieces are, and doing decomposition, and the whole deal.

Then you go into coding. And when you're a coding – when you're coding, you're an engineer. Woo-woo. You got you're little hat on. You're in your train. No. You're writing code, and you're thinking about good software engineering, but you're building something, and you're driving a process forward.

Then there's this thing called testing, which you do to your code. And a lot of times that means it's 4:00 a.m., you think your code works, the project is due tomorrow, and you decide, "Hey, I'm gonna do some testing." And this is the role that does not get taken seriously because it's 4:00 a.m., and your project is due. Often times testing means, "Hey, I ran it once – maybe twice, and I felt good, and I stopped. And I tried the simplest possible case I could try."

That's not the role you want to play when you're testing. When you're testing, you're the vandal. And that – like you got the spray can in your hand, and you're like, "Yeah, we're going out." You are thinking about, "What do I need to do to break this program?" You're not thinking about, "What's the least I can do to please make the program run once, and get through, and hopefully there's no bugs, and if it makes it through once, I'll just believe it's correct, and that's a good time, and I'll turn it in?"

You're just banging on the thing until you break it. As a matter of fact, a lot of software companies – most good software companies have people who this is their whole job, right? They don't do this. They get code from other people, and just try to break it. And this is a skill. You actually – you see some amazing things that people do to break code, but they do it, right? And you got to appreciate that because when you're writing your paper on your word processor at 4:00 in the morning, and it crashes, this person wasn't doing their job because you found the crash, and they didn't. Okay. So that's what you want to think about. You really want these people to be vandalizing the software. Okay.

And then after the vandal comes along, and smashes everything up, then you have this job that's left to you, which is debugging. And when you're debugging, you could think,

"Well, I'm the police officer, and I go, and track down the vandal, and make them stop." In fact, no, you want the vandal to keep vandalizing. You're the vandal. You got to keep trying to break your software, but at the same time you're a detective to try to find out where the breaks are happening, and how to fix them. Okay.

Now, here's something that people find a little odd, and I like to call them the four – well, some people don't find them odd. The people who do software engineering don't find them odd. I like to call them the four D's of software. Okay. And the first D is design. You've already seen that. And then there's development, which is actually writing your code. And then there's debugging. And then – anyone know what comes after debugging? You're like, "D party." No. Deployment. Okay.

And in this class so far we've focused on design, and development. And now, we're gonna spend some time thinking about debugging. And deployment is kind of like when you give it to your section leader. We don't want to worry about like you released software version 1.0 of your hangman program, and then we're like, "Oh, so when is 2.0 coming out?" And you're like, "Hum? Let me get back to you on that." And then you issue a press release, and the dates slip, and shareholders get angry. All right.

Yeah, like – see we should have CS106a shareholders for like, "When is hangman coming out? Oh, my God, you slipped the deadline." All right. Then you'll know how serious those late days are.

Then there's deployment. Okay. And the one thing that I would pause at, and actually a lot of people agree with, is that any problem that cascades from one step to another causes a factor of ten times in the cost to fix it. Okay. Which means you come up with a bad design for something, you're gonna spend ten times as much time writing the code for that bad design. And the bugs that come up as a result of bad design are gonna be a 100 times as costly to fix. And once that software gets shipped out into the field, and the only way to fix it is to issue a new version, that's a 1,000 times more expensive.

And people look at this, and they're like, "But, Marron, you can't be serious, right? How could it be a 1,000 times more expensive?" Well, let me give you a little example. How about our friend the space shuttle? All right. We have a space shuttle mission going on. Space shuttle had an initial set of software that was written for it. Okay. Right.

And we're thinking like, "Oh, you know, like in tex." How much do you think one line of code costs when they were all said, and done? If you think of the total amount of money spent on the software development effort for the space shuttle versus the number of lines of code that were written, how much did it cost per line of code? Anyone want to venture a guess?

Student: [Inaudible].

Instructor (Mehran Sahami): Like dollars. $10.00. That's kind of fun. That's probably a job I'd take. Anyone want to venture another guess?

Student: [Inaudible] thousand.

Student: $1.00.

Instructor (Mehran Sahami): $1.00. Higher – higher. Does that scare you?

Student: No.

Instructor (Mehran Sahami): Does that scare you? I'd take that job. I'd be like, "$1,000.00 a line? Yeah, today in tex. Tomorrow, [inaudible]." Right? Like I'll take that job. That's the right number. Okay.

There is about 500,000 lines of code – and now there is more – but in the initial version of the space shuttle, and it took a total cost of $500 million to develop that software. Okay.

So this is serious, right? This isn't something where we're like, "Yeah, it's fine, and we're vandals, and we're debugging." Right? When you're on the space shuttle, you better hope someone was doing their job doing this, right? Because if something's wrong, you can't fix it up in space, right? In space no one can hear you scream – well, if you're in the movie "Alien." But if you're on the space shuttle, it's real difficult to actually fix software. Okay.

Mars Rover was actually lost because of a software issue. Not actually – well, it was a software design problem that came up. All right. Anyone know what the design problem was?

Student: [Inaudible].

Instructor (Mehran Sahami): Yeah, it was basically two different – it wasn't – it was all basically the same difference as feet versus meters, right? One person was coding to a spec that said, "Oh, the input you're gonna get is in feet." Another one thought it was in meters. Right? Those are off by a factor of three. That's enough to kill the project. And guess how much money was spent on that thing. A lot. All right. So it's important to think of this – well, so how do you actually think about doing good debugging?

So the way we want to think about debugging is in some sense it's a river. And you're like, "What? What is Marron talking about?" It's – this is a river. This is the river of your program flowing. All right. And it's just nice. It's wonderful. And you're like sitting there, and you're having a picnic down here, and the sun is shining, and the birds are chirping, and you're like, "If my program gets here, and life is good, everything's just fine."

But what happens when the river is flowing? You run your program, and you get all this sewage down here. And you're like, "Having a picnic next to sewage is no fun." Well, what happened? So you go all the way upstream, and you look at the beginning of your program. You say, "Well, at the beginning I knew I had no bugs because my program

didn't contain any lines of code." And somewhere along the way as my program's running there was – oh, say this factory that was belching sewage into your water stream which showed up at the end when you're running your program, and this thing is a bug.

And often times the problem is there is not just one. If there was one you'd be like, "Oh, I can it. That's all good." Yeah, there's like factories all down, and up the river. There's some on the other side of the river. There's some in the middle of the river. All right. It's a bad time.

So how do you find those factories that are belching sewage? So here are some things to think about. First of which is the mindset of what causes – what tends to cause bugs. Okay. There's three things. I wouldn't say they cause all bugs, but they cause the vast majority of bugs. One is bad variable values. Okay. Some where in your program you have a value that is not what you expected, and as a result your program is doing something that it shouldn't do. What that tells you if you think about this mind frame is knowing the values of your variables at different points in your program is critical to understanding when the bad values entered your water supply.

Second of which is faulty logic, right? You meant for the computer to do X. You programmed the computer to do Y. Right? X, and Y were not the same. There's no such thing as a computer glitch. The computer does what you tell it to do. And so you might say, "Hey, the computer's not doing what I told it to do." Yeah, it's doing exactly what you told it to do, you just didn't tell it to do the right thing.

And last, but not least, and this is probably the biggest one is unwarranted assumptions. Right? "Hey, I thought the value were gonna be meters. You didn't tell me they were feet until after the mission launched, and we lost the Rover." Think about what you're assumptions are. And that's why when we talked a long time ago about pre and post conditions – right – when we talked about it in the little world of Carol – "Oh, we just didn't want them to walk into a wall." That was what assumptions were all about. What can your method assume when it's called, and what can you assume after your method's actually done, and do those match up?

So the things to keep in mind – and these are s – part of the zen of programming – debugging. And then I'll show the actual doing of debugging. Okay. People have a tendency to look for complex problems. Most of the problems are simple. So don't fool yourself into thinking a problem always has to be complex to be a bug. Some times it's just a very simple piece of faulty logic or some value that you mistakenly added one to or added three to some where that you didn't mean to under some conditions. Okay.

Be systematic. There's a tendency, and this tendency tends to increase as the amount of time left until the deadline for the assignment decreases, which is to just haphazardly jump around code, and be like, "It's got to be here some where. Is it over here?" No, that's the game I play with my one and a half year old son. "Is he over here? No. Is he over –" and he's just standing there with his hands over his eyes because he has the simple idea in

mind, and I'm looking for something complex, but I'm not being systematic. I'm just like, "Is it over here? Is it over here?"

And then finally at the end, right, had I just started at the beginning of my program, and began to trace it through, I would've found my problem. But I have a tendency not to do that. And part of the reason I have a tendency not to do that is that I make assumptions about where the problem is.

Now, hunches are great. It's great to think about, "Oh, I think the problem should be here," and have some intuition, and hunch about that. But when you have an intuition versus a fact, and the two collide – this is in your eleven rules of debugging on your next – on one of your handouts that you get. The fact always wins. Right? Your hunch never beats out a fact. So if you make assumptions about the problem, and you don't see the problem in the place of the code where you thought the problem should be, maybe the problem's not there, and you need to reassess your assumptions about those pieces of code that you thought were simple, and so they were error free, and in fact the error could be there. Okay.

Other thing is be critical of your code. Right? People have a tendency to look at some function or some method, and they think, "Oh, it's super simple. There can't be an error there." Or they just glance over it. I've actually seen at major software development companies, very large pieces of software written by brilliant, top of the line software engineers who the problem in their code was this instead of this. Right. It was harder for them to find that bug than to find issues with, like, memory, and garbage collection, and leakage, and all this other stuff because they were just used to that. Right?

They weren't critical of things like this. And when they finally saw this error you could just see like the eyes just nearly pop out of their head, and their head was gonna explode. But that's all it was. It was something simple they just – the code – they glanced through that code a thousand times and couldn't find it because they just didn't look at it critically. Okay.

And last but not least – and this is the important thing at 5:00 in the morning where you're programs not working. Remember this. Trust me. Don't panic. Panic is the worst thing you can do with bugs. And why? Because people sit there, and they think, "Oh, my God, there's this bug in my program, and it's over here. No, it's over here. No, it's over there."

No. Your code's not changing unless you change it. If you have a bug in your code, there's a bug. It's in one place. It remains in one place. Every time you run your program it's in that same place just begging you. It's like, "Come on. Come find me. Did he find now? No. Come on. Come find me." It's just there. It's that same little line of code, right? How many total lines of code are there? Like a couple hundred at most, right? It's in there. It's not moving. It's up to you to find.

Don't panic about it. Be systematic, question your assumptions, be critical of your code, and eventually you'll get there. But if you panic you won't because you won't be seeing it. Okay.

So what are some approaches we can take to debugging now that we have some of the basic sort of philosophy down, and the fact that the real critical thing is don't panic? The simplest approach to debugging is something that's called printf debugging, and printf is actually something that comes from the language C.

The way you can think about it is println debugging. And the basic idea is right – if bad values are the problem with your program, which in most cases they are, put some extra printlns in your code that tell you what the value of your variables are at different places in your code. Use printlns to let you know that you called a particular method. So at the top of a method just have a println that says, "Hey, method X just got called." And right before that method's supposed to be done, you add another println that says, "Hey, method X is just about to finish." And if you notice some problem that happens between those lines probably something's going on in method X.

Another thing you can do is just write out the values of your variables. And that's probably the thing I would recommend the most. At different places in your program just stick in printlns, and see is the values that you expect there matching what you thought was gonna actually be. Right? And if they're not you can sort of see at what point did the variables change the values that you didn't expect, and so something bad must have happened between the last time you saw the values that were okay, and the time in which they became bad. Okay.

The other thing – and this is a little bit more involved, but a not a big deal. It's something called unit testing. Right? People have a tendency to test their entire program at once. They write the program, and they run it, and something doesn't work. Well, why not actually test out individual units or individual methods in your program, right? Write a call to a particular method where you pass in the values for all the parameters with known values, and see if the thing that it gives you back is what you expected it to do.

That's called unit testing because you're just testing one unit at a time. And when you're done doing the testing you can take those calls out, and run your whole program. But when you write some method, especially if it's a complicated method, just write a couple lines at the top of run that the first thing they do is call that method to see if it's actually doing something reasonable with some artificial values that you give it. Okay.

So, one way that you can combine a lot of this stuff is if you happen to be working in a development environment that gives you facilities to do debugging more easily. And so I'll show you, if we come to the computer, Eclipse actually has a very nice debugging environment. Okay. And so rather these printlns, you can do sort of in any program, right, because even if you're doing a program that you don't have a nice debugging environment, you can still do println, and life will be good.

So let's actually debug a program using the debugger in Eclipse. So here's a little Roulette program, and the way Roulette works is we're gonna have some constants to begin with, how much money you start with, and how much you're basically wagering or betting on every spin of the Roulette wheel. So the way Roulette works is we have a big wheel that we spin. And here's all the rules of Roulette down here. I'll just show you the rules. Ah, too much text to read.

The basic idea is the wheel has a bunch of different numbers on it from zero to 36 all sort of consecutive numbers. And the way you can bet on numbers is you can either bet for an odd number to come up, an even number to come up, a high number to come up, which means a number in the top half from 17 to 36, or a low number to come up, which means from one to 16. And you might – or if – one to 18.

You might say, "But, Marron, what about zero?" Zero is always a loser. So, zero doesn't count as an even number. It doesn't count as a low. It's just always the losing value. And the reason why that exists is because that's the house's advantage. It gives them a slight advantage of about three percent over the player, and that's how in the long term they make money because zero is always a winner for the casino, never a winner for the lo – the player, other than that, everything's even money. If I bet $10.00 on low, and a five comes up, which is in the low half of the range, then I win $5.00. Okay.

So that's what the instructions basically say. Just a bunch of printlns they write out in instructions. Okay. We're gonna have a random number generator, rgen. That's an instance variable that's just generates random numbers, and it gets an instance of the random number generator. Hopefully after Breakout, and Hangman, this should be fairly comfortable to you.

And now we kind of come to the rest of the code. Okay. First thing I'm gonna do is set the font to be big. I'm gonna write out the instructions, and then I'm gonna play Roulette. And so I say, "Hey, in playing Roulette I have some starting amount of money that the user gets. As long as their money is greater zero, I tell the user you have some amount of money. I ask them which betting category they want to bet." That means either even or odd or high or low. So they type in a string. There's four betting categories.

I spin the Roulette wheel to get a value between zero and 36, and then if the winning – if there's a winning category based on the outcome that they have, I give them – I basically say, "You won this amount," and I increase their amount. If they didn't get a win in that winning category, then I say, "The number is not –" basically, "That number is not whatever you bet. So you're gonna lose." And I subtract the amount of wager from the total money. And if they happen to get out of this loop then their money is down to zero, and I say, "You ran out of money," and we stop. Okay.

So I might just run this program. Do te do te do – and see what's gonna happen. Notice I didn't show you all the other code yet? Because I want to show you the facilities of the debugger rather than just having you stare at code for a bunch of time, and try to debug it

because that's not the way you want to debug by just staring at code. You want to look at what the symptoms are.

So here we're gonna run Roulette. And it writes out all the instructions, and it says, "Enter betting category." And so we'll say, "Hey, I'm gonna bet even." "Ball lands on 33. That number's not even so you lose." I'm like, "Okay. I'm gonna go for even again." Good old even. Even always wins. "Number 35. You lose." "I'm going for even again." "27, you lose." "Even come on." "Nine – 31 – 15 –" and then you're like, "I had so much faith. I lost all this money, and then ball landed in two, and I still lost money."

And you just kind of sit there, and you're like, "Yeah, this casino's not really what I had in mind." And you try to leave before the bouncers get involved. But you realize you actually have a bug in your program, right? You bet on even. The ball landed in two. That should be an even number. And you still lost money. So then you think, "Okay. Something's wrong. What's wrong? Well, let me figure out right – this winning category thing, I should've said, 'If the outcome moded by two or remaindered by two was equal to one that should be odd. And if the remainder after I divided by two is zero that should be even.' So I would've thought this would've been the case. What's going on here?"

And so what I can do in Eclipse is I can set up what's called a breakpoint. The way you set a breakpoint is that a particular line over here on the gray bar you click twice. And if you click twice, notice I got that little circle there. Can you see the circle on the very edge of the screen? That's called a breakpoint. That means if I run the program now with the debugger my program will stop at that point, and allow me to look at, like, values of variables, and other kinds of things. Okay.

So I set that breakpoint by double clicking, and now I need to run again. So the way I run again is see this little bug up here. That means run with the debugger. See up here? It looks like a little tiny bug. So I click that to run with the debugger, and now it starts rerunning my program again. And it says, "Enter a bet." And so I say, "Even." And it says, "This kind of launch is configurative. Open the Stanford debugger perspective when it suspends. Should I open this perspective now?" All right. It looks all big, and nasty, and you just say, "Yes." Okay.

And it says, "Hey, your program got here. Look at what values you actually have." And so there's a couple things to look at. Up in this debug window up here, it shows me – remember when we talked about stack frames. This is what's called the call stack. The run method called the play Roulette method which called the is winning category method which is where I'm currently suspended. So it tells you all the functions that are currently – or the methods that are currently active to get you at your current point.

Over here in the variables window, it actually tells you the value of the variables. So my bet is even, outcome 13. I say, "Huh? It says the ball lands in ten. I thought that was supposed to be even, and I was gonna even. What's going on? Over here it tells me that my outcome, which is the thing that I'm gonna mod or divide by two to see if it's odd or even is 13. That's not what I got, which means at this point sewage has entered the

system. The value that I thought I should have is not the value that my program actually has." Okay.

So what am I gonna do? There's two things I'm gonna do. First of all I want to say, "Oh, I'm in this debugging perspective. How do I get back to the little editor that's my friend?" You come to the Stanford menu, and you pick switch to editor. Okay. That takes you back to what you're used to. Notice we're still stopped here, but you're back just in this editor view.

There's a couple things we're gonna do. First thing we're gonna do is I'm just gonna say, "Whoa. Stop the program. Something bad is going on." Next thing I'm gonna do is say, "Hey, my program involves random numbers. So I'm gonna set the seed for my random number generator, so I can count on always getting the values the same every time." So if you're debugging with something that involves random numbers, set the seed before you go into debugging, and that'll guarantee you always get the same numbers. So I set the seed.

The other thing I'm gonna do is say, "Hey, the number that got printed on the screen was different than the number that I actually got. So let me look at the code that prints a number on the screen." So here's spin Roulette wheel. The ball lands then it generates a random number, and then it returns a random number. What's the problem here? Anyone want to venture a guess?

Student: [Inaudible].

Instructor (Mehran Sahami): They're not the same random number, right? It generates one random number to say, "Hey, the ball lands in some number." That was kind of a social. And then it says, "Hey, you know what? Yeah, our casino – we just remove that ball, and we just pick another number randomly to say that's what it really was." Okay. That's – maybe a lot of fun for the casino, not so much fun for you.

So what we really need to do is inside here we'll have something like spin, and spin is gonna be whatever we get from the random number generator. Okay. So we'll just say, "Hey, spin is this random number." And then what we're gonna write is spin over here. And what we're gonna return is also spin. So we only generate one random number.

Let's save that, and we're like, "Oh, we only generated one random number. That must be the bug. I must be done. Oh, yeah. Yeah, good times. Good times." So I run my program again. Do te do – and I get this whole thing, and it says, "Enter betting category." "Even." "27, you lose." I'm like, "Oh, it must be working." "Even." "15." I'm like, "Oh, yeah, my program is so good. I'm such good –" yeah. And you think you've found your bug, and then you realize when you're the vandal, and you're trying to see if that error condition still exists, "Hey, we found one bug. That was great. That didn't solve this problem."

So you say, "Okay. Back to the drawing board. Let me quit out of here. I'm gonna go, and set another breakpoint over here. Okay. So let me clear this breakpoint. I'm gonna set

another breakpoint. I'm gonna run it with the debugger. Okay – which is what I didn't do before." So now I run with the debugger, and it says, "Enter a bet." And I say, "Even." And it says, "You've stopped here." And so, "Yes. I want to do this debugger perspective."

And I say, "Hey, the outcome is 27, and so my – if I look at – my bet is not even, right? It should be odd." So I say, "Oh, that's great. Well, it shouldn't be odd, so that shouldn't match." There are some tools up here I can use to say, "Step through my program." There's one in the middle up here if you can look at this line that has sort of this curve to it. That means step over. And all of these – all these different icons are actually explained in your debugging handout, but step over is the critical one. It says, "Just execute this line of code, and go down to the next line of code."

You have two other options. You have step into, which actually makes a function call or method call if you happen to be in a method, and step return, which says, "Step out of the current method I'm in." They get a little funky.

So step over is what we're gonna do. So we step over, and odd doesn't execute, and you're like, "Yeah, odd shouldn't have executed. Even – yeah, my bet is even, right? I know over here my bet has the value even, so I execute another line. Whoa. What's going on there? I should've gotten bet was equal to even, and return the fact that my outcome was actually true that I actually won." What's the problem here?

Student: [Inaudible].

Student: [Inaudible].

Instructor (Mehran Sahami): Pardon?

Student: [Inaudible].

Instructor (Mehran Sahami): Ah, strings. These are strings. I can't use equal equal on strings, right? It just doesn't work. It doesn't work when you're not looking either. So I say, "Hey, come back to the editor, and when you want to make some changes – yeah, instead of checking for equal equals, I actually need to say dot equals here."

As a matter of fact I'm gonna use equals ignore case because that'll make the life of the user even better. They can either put in odd or even in upper case or lower case. So I'm just gonna do a little copy, and paste, which is always a dangerous thing to do. Do te do – so close. Do do – and then for high – and at this point I could go ahead, and save. Oh, it's still in this perspective. I want to quit out of my program.

And now there's one final thing I'm gonna do here. Let me switch to the editor, and I'm gonna run one – let me save, and run one last time. And this becomes very subtle. So if I say, "Hey, I'm gonna bet on odd. Yeah, look I'm winning now because I know that odd

numbers come up. I should've remembered this from last time because I set the seed, right?" And I keep betting on odd.

And then one interesting thing that happens is zero came up, right. This is something that may take a while to come up. So I start betting on even because I lost with zero before. It's gonna take me a while before zero might ever come up again, but I need to be – I need to persevere. And it's probably not gonna come up in this run. So let me quit this, and run it again. Do te do – even. And I should get the same sequence of numbers. Even, even, even – oh, I lost on zero. Right?

There's one subtle bug that's still remains in my code which is for the case for even I not only need to check for even, I also need to make sure that the value is not equal to zero. So the other final thing I need to put in here is and and outcome outcome not equal to zero. Okay. And then my code is actually fixed.

So sometimes it may take me a while to find the actual error in my code through running, but now you can use the debugger to find it. All right. And if there's any questions I'll take them after class, and you can pick up your finals –

Student: Thank you.

Instructor (Mehran Sahami): – over from – or the –

Student: Thank you.

Instructor (Mehran Sahami): – midterm exams over from Ben.

[End of Audio]

Duration: 54 minutes