

Programming Methodology-Lecture14

Instructor (Mehran Sahami): Are we on? All right. Looks like we're on. Let's go ahead and get started with a few announcements.

So hopefully you're still turning in, or you will be turning in, your assignment three soon. There is three hand-outs today. They're not all in the back there right now. There was a huge problem with the copier today, and Ben is actually gone to make — to finish up making all the copies of all the handouts by the end of the class. So at the end of class you can pick up all three handouts. Some of you may have already picked up one. There was some copies back there. But the three handouts are basically your next assignment, assignment number four, which is a hangman game. So you'll be doing another game. This one's a little less graphically involved, unless you actually want to add a lot of graphics, which is another fun extension you can add to it if you want. But it sort of involves much more algorithmics with strings and also gives you practice using multiple classes. So this whole time we've talked about, oh, you can have all these different classes. Well, this is your chance to actually use multiple classes starting now in hangman.

The other two handouts are the practice midterm and practice midterm solutions. So those will also be here at the end of class. That will give you a whole bunch of details about the midterm and what the midterm actually covers, but it will also give you examples of real exam problems that have been given in the past. So you can work on them. Do it in the time, you know, time sort of setting so you can see what you're slow on and what you're fast on. But it's kind of in flavor, very similar, to what the real exam's gonna be in terms of what it covers, the kind of complexity of the problems, et cetera. So you can get that after class.

Last time to announce it. If there is midterm conflicts, email me by 5:00 p.m. today. So if you have an unmovable academic conflict with the midterm next week on Tuesday, 7:00 to 8:30 p.m., send me an email by 5:00 p.m. today. After 5:00 p.m. today I'm gonna take all the emails that I've got and try to figure out a time that everyone can make based on those emails and schedule the alternate time. So after 5:00 p.m. today I can't accommodate any other requests, unless of course you happen to be free at the alternate time you schedule, in which case you sort of got lucky. But if you have a conflict with the midterm, 5:00 p.m. today, send me email. That's the deadline.

Just in case you're wondering where you would actually take the midterm, it's in Kresge Auditorium, one of the few buildings on campus that's actually large enough to accommodate this class with alternate seating. So Kresge Aud. Anyone know what the significance of the, "K," in "Kresge" is? It's the same as the, "K," in "Kmart." So anyone ever gone to Kmart? Ever seen a Kmart? That's like Kresge-mart. That's where the, "K," comes from. So you wonder where all that money came from to donate that huge building. So midterm's in Kresge.

Along somewhat different lines then, we're gonna make a slight change to the syllabus. This won't affect anything that's on the midterm because the midterm will only cover stuff through today. Anything after today is not fair game for the midterm, will be fair game for the final exam. But a small change to the syllabus is — Friday's class will remain the same. We'll talk about files. But next week on Monday we were gonna start talking about debugging strategies. And I'm gonna defer that lecture until after the midterm. So actually starting on Monday next week we're gonna start talking about arrays. So if you want to keep up with the reading in the syllabus, that's Chapter 11. We're gonna start that on Monday, as opposed to on Wednesday. So I've just moved basically things up by one day for arrays. It's not a big deal.

And we'll actually talk about debugging after the midterm. Part of the reason for that is for your assignment number three — or, sorry, your assignment number four for hangman, it's actually a three-part assignment. It's all one big program that you write, but we sort of break it up for you into three parts. That third part that you'll do at the very end, which is perhaps actually the simplest part, will make use of arrays which is why we're gonna start talking about arrays. So you will have seen all that by the time you actually need to do that third part.

Last but not least, as you know, assignment three is due today so quickly let's take the pain poll. Just wondering, how many people actually added extensions to break out? Wow, lovely. How many people were really gunning for the plus plus? Cool. Good times. I look forward to seeing those, so it'll be a good time. So the — in terms of the pain poll, I want you to just let me know how much time it took you to do the basic assignment. So if you did extensions, don't count the time for the extensions. What I really care about is the base assignment.

Anyone under two hours? Just wondering. It would have been frightening if you were. I wouldn't expect anyone to be. Two to four? A couple people in two to four. Four to six? That's a pretty reasonable contingent. Six to eight? Also pretty healthy size. You can tell that the bar drawing is a very scientific process. Eight to ten? Okay. Ten to twelve? Beginning to quickly fall off. Twelve to fourteen? Fourteen to sixteen? I put in 16 to 18 this time just for good measure. Sixteen to eighteen? A couple folks. Eighteen plus? Anyone taking a late day? A few folks for 18 plus, all right. And anyone taking a late day. All right. We'll see where you end you. I always have this tendency to think, like, oh, if you're taking a late day it's like I should just count you in here, but that's really not the case. Sometimes you're just like, hey, I decided to go play in the sun or whatever. And the weather has been lovely, hasn't it? All right. With that said, world is normal, life is good, average is still, on average, less than ten. Everything we sort of hope for. We sing — we hold hands, we sing Kumbaya, and the birds are out chirping. Good time.

Any qu — well, I shouldn't ask you if there was any questions about break out because some people are taking late days. Any questions about strings? Because after today we're leaving — well, actually after last time we're sort of leaving strings behind. After right now, we're leaving strings behind. You will use them just to no end on the hangman assignment because it's all about text and strings and characters and it's just a happy

camper that way. But if you have any questions about anything you've seen with strings, ask them now or I won't say forever hold your peace. Ask them now or ask them at office hours, right? You can ask later.

Alrighty. So today's the day when we lift the covers on the machine. It's actually when we sort of like rip open your computer, although I'm not gonna rip open a computer. I considered doing that. But we're gonna rip open your computer and look at what's inside. And it's really, really small so you have to bear with me, okay? But we're gonna talk about is memory. And after today you will be experts in memory. You'll just be like, "Hey, man, I might not be able to remember anything, but I know all about the memory in my computer," which is the important memory to actually think about.

So the first thing in memory that we want to think about, right. When we talk about memory some people have heard the term like RAM for Random Access Memory. And so, like, you buy some new computer and you go, "Oh, I got, like, two gigs of RAM in my computer," right? This is what we're gonna be talking about, is what that RAM in your computer actually means, okay? And how it relates to your job of programs, and why it's all important.

So the very basic notion, the very simplest concept of memory inside the computer is something that's called a bit. And all a bit is — it's actually shorthand for a binary digit. So if you take binary digit and you squeeze it together real hard, all of sudden the extraneous letters pop out, and you get bit, okay? And this is just a zero or a one. It is a single binary digit. Binary being base two numbers which means there is only zeros and ones as digits in that number system, okay? And that's the simplest kind of thing. So somewhere inside your computer there's actually, like, a little piece of silicon or whatever and a little transistor that can keep track of a zero or one. So when you get — we won't go below the transistor level. If you're really interested in that take, like, you know, an E class. But the bit is a zero or one.

And then we take a column — a whole bunch of bits, and we stick them together into something called the byte. You're like, oh, that whole term like gigabytes. Yeah, this is what it's all about. What a byte is is eight bits. Okay. So basically eight binary digits strung together is one byte. Okay. That's all it is. Now, one byte of memory would not be very interesting, right? You — it's like, oh, I could have one really small number in my computer, right? Because eight bits can represent, like, an integer between zero and 255. It's not very exciting if that's all the memory you had in your computer.

So when we think about bits and bytes, we also like to think about larger pieces. There's a piece sometimes people refer to as a word, which actually has nothing to do with English words. But a word is generally the size of what an integer is used to store in some particular language. And it turns out an integer in Java is four bytes of memory. So it's 32 bits. Thirty-two ones and zeros comprise the space that your computer actually sets aside for a single integer in memory, which is why an integer has some bounded sides, right? You can't just have a number very close to infinity in an integer because it just has some finite size. It turns out to be four bytes. And that's something that sometimes people refer

to as a word. We won't use the word, "word," very much but just so you know. What we will talk a lot about are bytes. And so bytes — there's a lot of terms related to bytes that you've probably heard. So how many people have heard of a K or a kilobyte? Right. Probably a lot of people. And you would like to think that the metric system, right, kilo means what in the metric system? Thousand, right. So you would like to think a kilobyte is a thousand bytes. But, no. Computer scientists come along and they're like, "No, no, no, no, no. That's much too round and we're not decimal people. We're binary people." And the closest number to a thousand in binary is two to the tenth which is equal to 1,024.

So a kilobyte — one K when you talk to a person who's actually a computer scientist or someone who's putting the RAM together for machine, you actually, sort of, get a little extra, right. It's kinda like there you're, "Oh, I got some bonus RAM." And, like, this whole time you didn't even know. They were just giving it to you for free, right? And so there's this notion of a meg, right? Like a megabyte. And a megabyte is basically just a thousand kilobytes. As a matter of fact, it's not just 1,000 kilobytes, it's 1,024 kilobytes. Because we do everything in powers of two, so it's actually two to the 20th.

And then there is probably you — something you've heard of called a gig or a gigabyte, right? These are the common ones. This is 1,024 meg. So when you start adding these all together, right, you're actually sort of getting a lot of bonus memory for your gig. Because it turns out you're getting, you know, 124 megs, and each one of your megs is getting 124 K, and each one of your K is getting 1,024. So you're getting a little bit of a bonus. So now it's time for the advanced course. You're like, "Yeah, man, I got so much RAM in my computer. I got gigs and gigs." And so then I would say, "Oh, yeah, really? You got 1,024 gigs?" Because what would that be if that's what you had?

Student: Terabyte.

Instructor (Mehran Sahami): Terabyte. All right. So the next one is a terabyte. What comes after terabyte?

Student: Petabyte.

Instructor (Mehran Sahami): Petabyte. Okay. And now we're getting into dangerous territory. So each one of these is a factor of 1,024 greater than the next one, right? What comes after petabyte? Exabyte. There was a few folks. We'll see how long we keep you. How — what comes after exabyte? Anyone?

Student: Zettabyte.

Instructor (Mehran Sahami): Zettabyte with two, Ts. Not to be confused with Catherine Zeta-Jones. That's an entirely different zettabyte. So this is a zettabyte, all right? Just horrible, wasn't it? And then after zetta, just to drive the point home, is a yattabyte. I always like to think of it as like a yoda byte, right? It's just, like, so much

information you can't even conceive of it. But it's a yattabyte. Oh, yeah, that's a yatta good time. We just not even close to this, okay?

A couple things if you just want to think of orders of magnitude just so you have some vague idea. If you took the printed collection in the U.S. Library of Congress, that's roughly equivalent to about ten terabytes. Is all the printed material if you thought of each character being represented by one byte. Okay. And you're like, "Oh, that's interesting." If you took all printed material ever, that's about 200 petabytes. So at this point we're getting pretty close. Like, you can go down to Fry's, right, or some other place. I shouldn't name a particular manufacturer or a particular vendor. And you can probably buy about a terabyte of storage relative — well, I shouldn't say relatively cheaply but for a couple hundred bucks, okay? And you kinda crank that up and you can imagine someone for a couple hundred thousand dollars could have a petabyte, right? And then you sort of have a couple million dollars and you could have enough storage to store all the printed material ever, right? Which is not that far away.

And then the — I don't know how someone came up with this number, but I actually found this number and I was like, how do you measure that? Which is if you took the amount of storage that would be required to store all words ever spoken by human beings, okay? Like, where did they get that number, and it's increasing right now, right? They estimate that that would be about five exabytes. Right. So we're still sort of in this range, right? Like, yeah, Catherine, she's safe. And the little Zedi guy, yeah, we're not even close, all right? So we still got a lot more information to produce someday. And you're like, "Oh, I can do that now with my program. I can just have an infinite loop, and I can generate this." It'll still take a while. All right. But it's interesting just the amount of information that we're actually producing.

Now, when we think about actually storing all this stuff inside a computer — so now that you've got some notion of, kind of, orders or magnitude for these things and what these little bits and bytes really are — the other thing that we want to think about is what kind of representations do we have when we actually store this stuff? So it turns out because we like to think in binary, right, we have bits which are binary digits. Sometimes it's easier for us to think not just in terms of all the ones and zeros themselves but some other base system that's easier to keep track of. And so one you might think of is Octal, which is base eight numbers, right? So remember when we did the little asky chart and I showed that to you in Octal, and I was like, "Oh, these are base eight numbers." Like, the digits go between zero and seven. Yeah, that was Octal. That's something that some computer scientists do, but most computer scientists, at some point in their life, that — as a matter of fact, I would say all of them in some point in their life. And now is that point in your life.

We'll deal with something called hexadecimal. And hexadecimal, hex being six — decimal — yeah, it's hexicomical. Hexadecimal, six, and then, "deci," being ten is base 16. We put the six and the ten here. You're like, "Shouldn't that be 60?" No, it's 16. We add. So this is base 16 numbers. And you might say, "But, Maron, yeah, like, I remember these three, four, five, six, seven, eight and nine and those are all fun." And, like, when

you told me about binary I was like, “Yeah, it was just these two,” and when you told me about Octal you were like, “Yeah, cut here.” Now you’re telling me about base 16. I got no more digits. Where are my other digits coming from? And you’re like, “Well, I could have, like, a ten.” And I’m like, “No, ten is two digits.” And you’re like, “So where does it come from?”

Well, what else do I have beside digits? I got letters, right? So in fact, A is ten, B is 11, C is 12, D is 13, E is 14, and F is 15. And so when I want to write some number in hexadecimal — I’ll just write a number in hexadecimal because it’s fun to write numbers in hexadecimal for about two numbers. And then it stops being fun. But just so you actually see it. 2B — you’re like, “Yeah, over on the other side of the quad, this is Shakespeare.” On this side of the quad, this is hexadecimal. What is 2B? It’s base 16. What does that mean? That means this is the ones column. This, in decimal, would normally be the tens column. This becomes the 16s column. The next column, if we had one, would be the 256s column, et cetera, okay? So if we want to think of this number we have two in the 16 column so that’s 2 times 16 is 32. And we have a B in the ones column. Well, what’s a B? That’s 11. Okay. So we add 11 to that so we have 1 times B which is basically equal to 11 and what we get is 43. So 2B is actually the number 43 in decimal that we like to think of, okay?

So you might say, “All right, man. That’s great. Yeah, sure, whatever. Why are you telling me about this?” And the reason why I’m telling you about this is when we think about the computer’s memory, we think in terms of numbers that are hexadecimal. So a lot of times when you see memory drawn out — and I’ll draw memory because memory is so cheap these days I can just draw it on the board. So we have a bunch of cells in the computer’s memory. Each one of these cells is one byte, which you mean — which you know underneath the hood what that really means is that cell contains eight bits. Okay. And so each one of these cells has associated with it some location in memoria where it lives. You can kind of think of this whole thing kind of like the system of houses in the United States, right? How do you know where a house is, right? This little box is a house. How do I know how to get there? It has some address. It has some place that if I know its address, I know how to get to that box and see what’s inside the box.

And in the computer’s memory we refer to the address as using numbers, and they’re just numbered from zero up until however much memory the computer actually has. So if you have two gigs it’s somewhere on the order of two billion, whatever that number is. Except, we’d write those numbers in hexadecimal. So somewhere in the computer you have memory location A000. And that’s just some box. And after it comes A001 and A002 all the way down. And somewhere down in the computer’s memory there’s FFFF, and if your computer is large enough it’s got some more Fs in there too. And it’s in there. Trust me. You’re like, “Oh, I think that’s a little, you know, a little scary now to use the computer.” And up here there was — the very first address is 0000. But when we actually write these addresses, even though they would translate into some decimal value, we refer to them with hexadecimal. And the reason why we do that is we want to distinguish them as being addresses or locations of where things live in memory as opposed to the actual contents of memory, okay?

So somewhere — yeah, at this location there may be stored the value ten in some binary representation. But we want to distinguish that this ten is not the address ten. It's a number ten. It's an integer ten, whatever it may be. Whereas there is some address that we care about that's actually different. Okay. That's why we distinguish between these things. So when you're running your programs where is all this memory coming from, right? And it turns out there's two different sections. There's actually three different sections but two that are mostly relevant to you of where your computer actually says, "Hey, this person just declared an integer," or, "This person just called a method," or "This person just created an object." Where am I gonna get the memory to store all of the information that's associated with their integer or their object or their method called, okay? And so the places where this memory comes from — and I'll show you all three. There is some place where all of your static variables or your constants, you could kinda think of them if they're final variables. There are some special place that's set aside — so this is kinda special. All right. Or if we want to get multicultural it's especial, right? So it's special and basically we have static variables and constants that are just stored in this special location. Where is the special location? We don't really care. Okay. It's just our little lovely special occasion.

Now, beside their static variables and constants the thing — because these — at the beginning of our program it gets started and our — these — all these things come into being, all of our constants. And they never change, which means they don't need to be in some portion of memory that we access regularly. They just need to be somewhere in memory that we know how to refer to them and that's all, okay? So we can have dynamic variables, and what are dynamic variables? Dynamic variables are just any variable that you use new on, all right? So when you say, "new," some object — like you say new G oval, and you pass with some parameters, that's a dynamic variable. And the memory that comes from a dynamic variable when you do this thing called new is something called the heap.

And the way you can think about the heap is it's just like a big pile of clothes. When you need clothes you come along and you say, "Hey, I need some new clothes." And it gives you some clothes. And when you're done you sort of just say, "Hey, I'm not wearing those clothes anymore." And somewhere magically down the line when there's a piece of clothing that you stop wearing, the computer knows that you've stopped wearing it and it says, "Oh, I'm gonna go take that memory back." That's called garbage collection. It's c — wouldn't it be nice if there was, like, little gnomes or elves and, like, when you stopped wearing a piece of clothes and they knew you were done with it because you just had no way of ever getting back to that particular T-shirt again, they just went and threw it in the trash for you or they took it to recycling, which would be the more useful thing to actually do. Then, that would be a form of garbage collection. It's memory from the heap, just gets reclaimed when it's no longer being used.

Well, as long as you're using it — so if you have some G oval object over here, which we'll just call G. As long as G is still being used, this memory is still set aside for you in a place called the heap. And I'll show you how the heap is kinda set up in just a second. Okay. So last but not least, beside static variable and these dynamic variables, there is

another kind of variable that we care about which are local variables. Okay. So local variables come from a place called the stack, to be differentiated from the heap, okay? And the difference between the stack and the heap is that for the stack whenever you have local variables in a method or you also have parameters to a method — like, those parameters, remember, you get copies of the values of the parameters which means they have to live somewhere in memory. Some memory gets set aside for you automatically in this place called the stack. When you have your local variables and your parameters and your method is running those variables are alive. And when those variable go out of scope, like the method actually ends or you get some closing curly brace in which it can cause the variables to actually go away. The variables go out of scope in one sense or another. Then, the stack automatically says, “Hey, I’m gonna take that memory back from you.” Okay? So, again, this memory, at least in the Java world, is automatically managed for you. If you come from sort of the C or C++ world then you’re like, “Oh, how do I free memory?” You don’t worry about it in Java. It’s taken care of for you. It’s a good time, okay?

So what does this actually look like in the machine — in the computer’s memory? Let’s draw that over here. So where is the special memory? Where is the heap, and where is the stack? It’s kinda funky how it’s set up but you got to see it just so you can see what it looks like. Here is the area over here where all of our static or special variables are kept, okay. It’s just some section of memory; usually this is some low area in memory. What that means is the numbers, the addresses here, tend to be smaller. Like, let’s say around 1,000, okay? Which is actually 1,000 in hexadecimal, not in decimal. After the special loc — the special set of stuff we have the heap. And the heap grows. It grows downwards. What does that mean? That means if it starts at 2,000 and you say, “Hey, heap, I need some space for my new G oval.” It says, “Oh, okay. Well, I need to get you some more memory down here. Maybe this is location 2,004. I’ll get some more memory over there. And then you say, “Hey, I need a new G wreck.” It says, “Oh, okay. I’ll get to location 2,008 for your G wreck, and I’ll give you some memory over there.”

So the numbers that was associated with the heap increase, or what we refer to as the heap grows downward. Because you can think of memory — the low values are over here. The low addresses and the high addresses are over here. There is something else called the stack, right, that we just referred to, so let me stop having the heap kinda take over the whole board. The stack starts off in the very high addresses. So here is the stack. It starts off at a place like FFFF, right? Somewhere really — with a really large value for its address. And as it allocates more memory, it grows upward which means the addresses get smaller. And you might look at this diagram and say, “Hey, Maron, do they ever meet?” And they do sometimes. And if they do, it’s real bad news, all right. So if your heap and your stack ever overlap, they start writing over each other and usually what happens is your machine just crashes. But that’s why you got so much memory these days. You’re like, “Oh, I got a gig of memory.” Yeah, because there is a billion data addresses between the heap and the stack, usually not actually that much in a program. They put them a little bit closer together. But you got so much space in here it’s just frightening.

And you could probably use it in your program, right, by having some loop that just allocates a whole bunch of memory. But, really, the thing you want to keep in mind is that these things, at least for the programs that — for most programs that you will ever write, you never have to worry about the heap and the stack actually coming together, okay? So the one other thing that's associated with local variables, parameters and dynamic variables — mostly, actually, with dynamic variables, is how much space gets set aside for something, right? And that's one thing you want to think about is different kinds of variables actually have different amounts of space associated with them. So something like an integer, at least in Java, is four bytes. And this is not something you actually need to worry about because it turns out on different machines they may actually be implemented in slightly different ways.

A character — it uses a special kind of encoding where most characters are actually in two bytes. Sometimes if you get into some very funky languages that have lots and lots of characters or the way the special encoding for characters — the Unicode in coding is actually set up. Sometimes it can actually be as large as four bytes. So as far as you're concerned, you don't really care about the underlying numbers. It's an abstraction to you, and that's the key information, right? Even these little things like an integer or a car is just hiding information. It's hiding the fact that it's a bunch of ones and zeros underneath the hood, and you don't need to worry about it or how big it is. All you need to know is how to use it, okay?

So along with that, with dynamic variables there is also a little bit of overhead memory. And it's just an extra amount of memory besides how much space is taken up by, like, your integers or your cars or whatever variables are inside your object to keep track of other information about the object. And how much information is actually kept track of in that overhead is not important. But it's just important that you actually know that there's some overhead associated with it. So putting all this stuff together, it turns out that we can look at the mechanics of what actually happens underneath the hood when you run your program and you create new objects, right? So if we go to the slides for a second let's actually see, sort of, diagrammatically what's going on when we actually create new objects and declare variables, okay?

So what we're gonna have is some class called the points. And this is a super simple class. All it does is you can create a point with this constructor by saying, "Here's an X and Y location." And all it's gonna do is store an X and Y location for you and let you move them around, all right? It's real simple. It's something you could imagine might be, you know, convenient to have when you're doing graphics. So you pass an X and a Y, it sets its own internal instance variables PX and PY, which I just called PX and PY to remind us that they're our private X and our private Y. They're inside this particular class. And then there's another method called move that you pass some offset in the XY location, and it just changes its private X and Y by whatever offset you give it, okay? Fairly straightforward. Should be, you know, pretty easy to understand given all the stuff you've done with graphics so far.

And then we can say, “Okay. You have this class. You haven’t done anything with the class yet, right?” We’ve just said, “Hey, I have this class.” No memory has been allocated. Nothing’s actually been done. What you’re gonna have is some program somewhere, like my program, that’s gonna come along and create new points and call methods on those points. And we want to understand what’s actually going on in the computer’s memory when this happens, okay? Is everyone happy with the point class? I’m gonna make it disappear in just a second. Can you remember what it does? There is two methods. There is the constructor and the move, and then there is two private variables, PX and PY, okay?

So let’s see what happens when your program is actually running, okay? Now, just a little mnemonic. We have the heap and the stack. Heap is gonna grow down, stack is gonna grow up. Just so you can see that as we go along. And the first thing we’re gonna do is our method run gets called, right? When your program first starts up the method run gets called. What happens when a method gets called? Well, when a method gets called, we create all the local variables and parameters. Run has no parameters, so we don’t need to set aside any space for parameters. It does have two local variables, P1 and P2. So space — notice this is starting at the very bottom of memory like FFFF. There is some overhead for the fact that we create a method, and I just sort of blocked that off and called it overhead just to match, sort of, the same style that the book does it in. And then there’s some space, some number of bytes. In this case, I just said four bytes that gets set aside for P1 and some number of bytes that gets set aside for P2.

Right now they don’t contain anything because my program is just started. I haven’t gotten to this new point stuff yet. All I’ve done is said, “Run.” So it says, “Oh, okay. You have some local variables. I’m gonna set aside space for your local variables. Now what are you gonna do?” I’m gonna say, “Well, I want to create that first point. I’m gonna call new.” If I’m calling new, that’s gonna be a dynamic variable which means it’s gonna come from the heap. So here is what happens. I call this line, and I say P1 equals new point two comma three. So it says, “Hey, you’re calling new. You need some new memory. That’s dynamic memory. I need to get it for you from the heap.” Where does the heap start? The heap starts up at the top and goes down. What data is associated with a point? What values do you need to store to actually keep track of a point? Well, a point inside it has its private X and its private Y. So I have to set aside space for everything that a point encapsulates, which includes all of its private variables and its public variables as well. But all of its instance variables get space allocated for them.

And there is some amount of overhead associated with the fact that we need to keep track of some extra information about this point object, that you don’t need to worry about, the computer needs to worry about it. So there is some overhead. Let’s say there is four bytes for that. Four bytes for the integer PX and four bytes for the integer PY which is why you can see memories going from 1,000 to 1,004 to 1,008. So each one of the boxes up here is not one byte. I’ve just kinda stylized it to make it easier. Each box is now representing four bytes because integers are four bytes and the amount of overhead we’re keeping we’re just gonna say is four bytes, although you don’t need to worry about that, okay?

And what did we do when we called the constructor? We set the private X to be equal to the value two that got passed in and the private Y could be equal to the value three that got passed in so that all the constructor did was it wrote into the little places for PX and PY, which means it wrote into the memory locations that the computer set aside for it the values two and three. Okay. Any questions about that? Um hm?

Student: Why is there [inaudible]?

Instructor (Mehran Sahami): The 1,000 — sorry, is on the — oh, excellent point. Why is there a 1,000 over here? Because the one last thing I didn't actually mention is after we set aside the memory for the new points, what are we gonna do with that point? We're gonna assign it to P1. So here is the critical concept. How does the computer know where P1 and all the information associated with P1 is? Every object lives in an address. The way the computer actually is referring to objects underneath the hood is by the address at which they live, okay?

So you can think of — there's some little house over here that's P1's house and there's, like, the little kids in the house, PX and PY. They're like, "Oh, we're the little children. We store integers." And then I say, "Well, how do I know how to access you, P1?" I need your address, which is 1,000. That's where you live. So when I do this new and I create the memory in off the heap, it says, "Hey, I'm gonna assign that thing over to P1." What's the thing that you're actually assigning? You're assigning the memory address for where P1 lives or its starting address. And from the starting address we can find everything that lives in P1. So it's getting stored here. This value, 1,000, is not an integer. You don't want to think of it like the value 1,000, it's an integer. You want to think of this as the memory address 1,000. Okay? Um hm?

Student: [Inaudible].

Instructor (Mehran Sahami): Overhead. So it — that little overhead gives the additional information that you keep track of so — So how do we do the next line? Okay. Very similar kind of concept. We say, "Hey, I'm gonna create a new point. Oh, heap, I need some more memory." And so the heap says, "Okay, well everything up until 1,008 was allocated, so I'm gonna allocate — give you some more space and memory that's enough space to store another point which means I'm gonna give you space for your instance variables, PX and PY, and your overhead."

Notice this PX and PY are distinct from that PX and PY because this is the PX and PY that are the instance variables of P1. The second PX and PY over here are the instance variables of P2. They're distinct things, so they have separate memory to store them, okay? And same thing when we return a value that we're gonna assign to P2, what we're gonna assign is the address that's the beginning of the space that we set aside, the memory that we set aside, for P2. And that's at, you know, 1,000C so 1,000C gets stored there, okay? Any questions about that? It's basically the same thing except you can see the heap is growing down. All this memory has already been allocated, so any new memory we need requires the heap to keep growing downward.

Now, next funky thing, right? We're gonna do a method call. So we're gonna say, P1 dot move. We're gonna do a method call. When we do a method call what ends up happening is we're gonna have some parameters and some local variables which means we're gonna have to allocate some memory on the stack for that method call. So when we start the method call what ends up happening is we say, "Hey, P1, I want you to move yourself by 10 comma 11." So what actually happens in memory? What happens in memory starting at this point upward is we create a new stack frame. Remember when we talked about stack frames? We said when you call a method it's sort of like the previous method or previous place you were running just gets suspended, and we sort of create this new frame on top of it that contains all the information. We drew little boxes for the variables and all that when we did stuff like tracing palindromes and all that happy news. Remember that? If you remember that, nod your head, right? Yeah, what was going on underneath the hood? It wasn't just nice — I mean, at the time, it was nice pictures. This is what was going on underneath the hood.

So memory was getting created on the stack, and that's why we refer to this as a stack frame. It's keeping track of that frame of information for the method call and it does it on the stack, which is why it's a stack frame. What information goes on the stack frame? Well, you get some overhead for the fact that you're calling a method, so there's this — overhead is this first four bytes here. Then, you get all of the parameters that you're passing to the method. And you might say, "But, Maron, the parameters I'm passing to the method are DY and DX. What's this "this" thing all about?" Okay. There is a hidden parameter that you didn't know about until now and now you know about it. When I'm calling a method on a particular object, how do I know which object — once I'm actually getting into the details of things — I actually called the method on? And so when you make a method call on an object, the very first thing after the overhead that gets put onto the stack is, essentially, a pointer to the object, right? It's what we refer to as the this pointer. Remember when we talked about the this pointer, and we said this is basically a way for an object to refer to itself when it's been called? Well, how does it know how to refer to itself? It needs to know where itself lives.

You might say, "That's kind of weird, Maron. Why does an object need to know where it lives itself?" Because when I write some method, these methods are not yet associated with particular objects that I created. Once I create a particular object and call that method on them, this method over here needs to know, "Hey, when I'm doing these modifications on PX and PY, which PX and PY are you referring to?" That's where this comes in. This said, "Hey, I'm gonna tell you the place where the particular object lives whose method is being called." So whatever modifications you make over here, you know to make them to the object that lives at this address, which is why it's called this. Okay? So when you make a method call the first thing you get is the address of the particular object that you're making the method call on, right? We're making a method call on P1 and P1 lives at 1,000. All right. So P1 is just 1,000. So this is the value 1,000, and then I get the parameters in reverse order because the stack is growing upward, right? So I get DX first, then I get DY. And those just get the value 10 and 11 that are passed into them. I get copies of those values, okay? Any questions about the this pointer? Uh huh?

Student: I just had a question about graphics variables. And when you add like a G oval to the screen or something —

Instructor (Mehran Sahami): Um hm.

Student: — is it a dynamic variable or is it a local variable? It's — if you add it to the screen in a method and then you — it's still on the screen in a different method.

Instructor (Mehran Sahami): Yeah, a G oval — G oval is gonna be a dynamic variable because you called new on it, so anything you say new on is a dynamic variable.

Student: Then why is point A a local variable?

Instructor (Mehran Sahami): Because what's going on here is that P1 is just the address of where the actual stuff lives. All the data associated with that particular object, P1, is on the heap, okay? So P1 is a dynamic variable, right? The place P1 — where I store the address or the place I keep track of its address to know where to look it up is something that is local. So I keep track of P1 by itself as being local. The actual object itself that's created when I do the new is on the heap. Does that make sense? So the address is a local variable. So when I say, "Point P1," all I'm getting is an address. When I do the new, that's when I'm actually saying, "Hey, set aside space for a dynamically created object and let me know where it lives." So, yeah, there's a subtle distinction there but the important thing is that the variable itself, just the address can be a local variable, but the actual object itself is the dynamic variable. Uh huh?

Student: Well, so do all the different parameters that you — or all the different messages that you could possibly pass to like say a G oval get stored on the heap or, like, would the set fill be just false by default? Would that actually go on the heap or is there nothing that's put there until you actually send something to it?

Instructor (Mehran Sahami): Yeah, so you would need to know all the internals to G oval and G oval does actually keep track of an internal variable for whether or not it's filled, right? So when you create an object, whether or not you set its variables, those variables all need to have some initial value that the object is gonna set itself if you don't set them. So they're all on the heap. Uh huh?

Student: So if there is some sort of dec — like an integer declaration like you're saying N — X equals, like, ten or something, do you get, like, something on the stack for, like, where X is and then you get, like, ten in the heap?

Instructor (Mehran Sahami): No. So N — that's the funky thing, and we'll get to that in just a second. Ince, doubles, characters and bullions are different than objects, right? So when you use ince doubles and characters, you never say new. So if you don't say new, it's gonna show up only on the stack. Things that you say new on are the things that show up on the heap. So all — either your basic types of variables like cars and bullions and ince and doubles, those — when you declare those you never said, "Hey, I need a

new character.” You just said, “Hey, car CH,” and you got a character CH. That was all on the stack. The only time you actually created something on the heap was when you said new. Like you said, “Hey, I want some new G recs.” Then you actually got one on the heap.

So let’s keep following this through and see what happens when we make this move call, right? We got all this information on the stack. We haven’t actually done any of the instructions and move yet. So move comes along and says, “Hey, I’m gonna do the first instruction. I’m gonna add whatever is in DX, which happens to be the value ten, over to PX.” Which PX do I know it to refer to? It’s the PX that I get to by following the this pointer or going to the address of this. So it goes up to 1,000 and looks at the overhead and so it can figure out where PX is starting at location 1,000. And it says, “Hey, it’s over here. Its value was two. I add 10 to it, so now it gets the value 12.” Okay? And it does the same thing on the next line when it’s modifying PY. It says, “I want to modify PY. Which PY am I referring to?” The one that’s at location 1,000, so it goes to 1,000 and says, “Hey, where can I find PY from starting at 1,000?” It’s over here. And it adds 11 to the value 3 that was there before, and I get 14. Okay? Now this method is done, right? The move method, we executed everything in the move method. It’s done.

Here is the funky thing. Remember I told you when a variable that’s on the stack goes out of scope, when it’s no longer being used, the computer automatically comes along and says, “Hey, that variable or that memory is not getting used anymore. I’m gonna take it back.” So all the memory over here on the stack that’s associated with the stack frame for the move method, right, which is the parameters to move. The this param. or the this variable, which is actually sort of a hidden parameter to move and the overhead associated with move, that was all the memory that got allocated when I did this move method. This move method is now done, which means all of the memory that’s allocated just to execute the move movement automatically gets deallocated by the machine. It’s what we refer to as popped off the stack. Because if you think about the stack growing and we get to a certain point and we’re like, “Hey, we’re done with this top part of the stack.” We sort of say, “Boink,” and we pop it off the top of the stack and it goes away. So when we pop it off the stack, moment of silence, it goes away, right? And then the program just keeps executing from wherever it left off, right? If there’s another method called, guess what? The stack is gonna now grow from this point, okay?

Is there any questions about that? If we were to call the move method again, it would recreate all of the stuff it needed on the stack for another invocation of move because all that stuff is temporary, right? After it does move once it doesn’t need it anymore. If you involve move again, it will create it again. That’s just the way life is. All right? Now, any questions about that? Now, one thing to keep in mind that’s important is that there’s a duality here, right? I keep referring to this thing called the pointer. So I will just whip out the pointer, right? Think pointer like pointer. I always feel like a little evil when I actually hit the board with a pointer. But the way to think about the pointer is, right, just think of the value 1,000. Really, this is referring to address 1,000. This is just kinda pointing up to 1,000, right? We don’t really care what the value was in here. All we know — it’s some memory address up to 1,000.

So what are we going to do? Let's just draw it that way, okay? So rather than having this thing write out 1,000 in here and remember, oh, yeah, that 1,000 is supposed to be a memory address, et cetera, we can just draw it as a pointer and say, "Yeah, this thing is really some memory address, and the memory address is just pointing over here." Yeah, this 1,000 over here lets us know that it's 1,000, but graphically it's a fun way to keep track — and you're like, for some definition of fun — it's a fun way to keep track of it because then you know that you're not gonna muck with this variable in here as though it were an integer, right? This thing is just some — referring to some piece of memory where you're actually storing some object. And similarly, so is P2, okay? So that's what we like to think of as the pointer view points of the world, right? It's just the point that things that are memory addresses are really pointing somewhere. They're telling the machine basically, like, "Look, I know where you live."

Let's do another one real quickly, okay? So another memory allocation example. Here's, again, our points, right? Same point that you just saw, except I've left off the move method. And now I'm gonna create something called the line. And the line has two points associated with it. So when I create a line, I pass at two points to its constructor. Like, two things of this object. And it has a — a line has a beginning and an end which are private points that it keeps track of, okay? So I say, "Hey, beginning is gonna be the first point you give me. End is gonna be the second point you give me because you could have told me a long time ago that a line is defined by two points." So if I know those two points, then I know what the line is, okay? So what's going on if we actually think about this in the computer's memory? If we want to try to create two points and then create a new line. Okay? So everyone's sort of clear on points and lines? They're just two basic classes just to kinda illustrate the point. Ha, ha, yeah. Not funny. So when we actually create the two new points we get exactly the same sort of diagram we had before, right? We have memory allocated on the heap at location 1,000 for point one and 1,000C for point two.

The other thing we have is in this version of the run program, we have a local variable called line, which I've just abbreviated LN. And so we've gotten some memory allocated for it over here. Notice it doesn't have value yet. It doesn't have a value yet because we haven't asked the heap yet for the new memory that actually stores all the information for that line. All we've done is say, "Hey, you know what? I'm gonna have some local variable called line in here." When you get to the new line over here, then you're actually gonna ask the heap for memory, and things are gonna get set up. So let's see what happens when that line gets called, okay? We're now gonna go into new and excruciating detail. When we say new before we do any kind of assignment over here — so this still remains blank. When we say new what it does is it says, "Hey, I'm gonna go over to the heap, and I'm gonna allocate space for a new line." What are the variables that are part of a line? There's two variables, a beginning and an end, okay? And they're both points. I haven't initialized them yet. I'm gonna actually get into the details of the line constructor in just a second, but it sets aside space for you for its variables which are beginning and end.

Now, we call the constructor method — so, actually, I didn't show you this before when we did it for points, but what was actually happening when we called the constructor, is the constructor is just a method just like any other method. So what happens with this method? Except the constructor doesn't have a this pointer associated with it because it hasn't created the object yet. It's still in the process of creating the object. What it does have, though, is it has the parameters that get passed into it. Well, what are the parameters that get passed in there? P1 and P2. What are P1 and P2? They're pointers. They're just values that are — happen to be memory addresses. So what I do is I pass a copy of the address, 1,000, and a copy of the address 1,000 and C, and I don't create new versions of P1 and P2. All I do is I pass copies of their addresses, okay? Are we clear on that? That's the critical concept here.

So what happens after that? This guy comes along and says, "Hey, sa — beginning to be P1." Well, what's P1 that got passed into me? It's 1,000, so it just sets beginning to be 1,000. And then the next line comes across and says, "Hey, set end to be P2." Well, what's the P2 that got passed into me? It's 1,000 and C, so it just sends end — sets end to be P2. And then it says, "Hey, I'm done. I did my work. I'm the constructor. I did the two lines you told me. So it's done." And all the memory that's allocated with this particular method call, which are the two parameters in the overhead that's associated with calling this method, all get popped off the stack. So they get popped off the stack and they're gone. And now we say, "Hey, thanks, line constructor. You just created a, you know, first of all, thanks, new. You allocated the memory for us. Thank you, line constructor. You went ahead and initialized all the fields that I needed for a line. Thanks a lot. What do I do with that now?" And it says, "Yeah, what I need you to do is assign that object where that object lives to the variable line."

And so the variable line, which is over here on the stack, is a local variable, finally gets assigned the address of the line object on the heap, which is at 1,018. So when the whole line executes, basically what it's done is it's called the con — set aside memory with new, called the constructor, had the constructor fill in the initial values. And then what actually gets assigned to the line over here is the beginning memory address for this object that was allocated on the heap. Okay? Any questions about that? Now here is the funky thing. Let's take a look at this in the pointer viewpoint of the world, right? So the first thing we could do is say, "Hey, let's look at the pointers over here, right? These are all pointers. This points over to 1,000. This points over to 1,000 and C. This points over to 1,018. So we could just draw that looking like this.

But we're not done with all the pointers, right? There's still more pointers left. Guess what? Beginning and end, they're just pointers back into the heap, right? Beginning, it just says, "Hey, the point that you gave to me is just a point that starts at memory location 1,000 and end is just location memory of memory 1,000 and C." So really these guys are also just pointers back to the respective locations to those particular objects, okay? Any questions about that? I know that looks kinda funky and the reason why I did it on slides as opposed to on the board is because I want to post the slides so you can refer to them back after class if there's any questions. But that's the funky thing. You can actually have things on the heap that refer to other things on the heap. Okay? Uh huh?

Student: [Inaudible].

Instructor (Mehran Sahami): Well, and d — yeah. It's — I mean, you need to convert it to decimal, right, if it's a hexadecimal number. You don't care what the actual value is, all you care about is that it's an address. It's not, you know, a number that you would manipulate like an integer.

Student: [Inaudible].

Instructor (Mehran Sahami): We can talk about that offline. I mean, it's easier, as far as the book's concerned, to be able to do differentiation, to differentiate between addresses and non-addresses, but there is also some history to it in computer science.

So I just want to show you one thing very quickly, and then I'll give you a quick video. So if we have point, P1 equals new point one comma one and then we have some point, P2, that equals P1. What's really going on in the computer's memory, right, is P1 is just some address somewhere. And this is on the stack. And P2 is also just some location that's also gonna be an address because it's just an object. Now, this is the stack. Here is the heap over here. When I create P1, I sort of create some space over here that's gonna store a one and a one and there's some overhead down here that I don't care about. And this guy's gonna point over to here. Let's say this is a memory location ABCDF. Okay? It's just some memory location. When P2 comes along and I set P2 equal to P1, it just gets the same value as P1. It gets ABCDF. Which means this guy is pointing to the same location, okay?

So if that's the case, then if I do something like say P2 dot move ten comma ten, what happens there? It says, "Hey, I'm gonna go and move based on P2." Well, where is P2 pointing? It's pointing over here. It's pointing the same place P1 is and says, "Oh, I'm gonna move it by 10 10 so I get 11 and 11. And you might look at that and you say, "But, Maron, you just changed P1 without ever referring to P1." Yeah. If you have two objects that are pointing to the same place because you set them equal to each other, what you've done is set their pointers not created copies of objects, okay? That's why — it's sort of like you're old enough to know what's going on now, right? Before we just said, "Hey, every time you just create a new one and that way they'll all be distinct." If you don't create a new one and you set some local variable to be equal directly without doing a new — notice I didn't do another new point here. All you're doing is setting up pointers to be equal to each other. Which is why in the days of yore when we talked about our friend the string — remember the string, and we said if you had some string S and this was equal to hello and you might want to say — and this was S1 and I had some other string over here, S2, and this was also equal to hello. And I might want to say if S1 is equal to S2 and I told you, "Yeah, this doesn't work because you need to call S1 dot equals S2." The reason why it didn't work is because when you check for equality here, it's not checking to see if these strings have the same characters. These strings are objects. They have memory associated with them. When you check to see if the actual objects are equal to each other, it looking to see, "Are they referring to the same actual thing underneath

the hood?" These are not. These have two versions of hello in different places in memory. One is over here and one's over here.

So S1's a pointer or S2's a pointer here. S1's a point here. These guys are not equal to each other. They would only be equal to each other if they were, in fact, the same object and I set S2 equals S1, okay? Is there any questions about that? Alrighty. So if you want to go you can go now. And if you don't, I'll show you a little video. It's just a little fun claymation.

[Video Plays]

Male Speaker:Hey, Binky. Wake up. It's time for pointer fun.

Binky:What's that? Learn about pointers? Oh, goody.

Male Speaker:Well, to get started, I guess we're gonna need a couple pointers.

Binky:Okay. This code allocates two pointers, which can point to integers.

Male Speaker:Okay. Well, I see the two pointers, but they don't seem to be pointing to anything.

Binky:That's right. Initially pointers don't point to anything. The things they point to are called pointees and setting them up is a separate step.

Male Speaker:Oh, right. I knew that. The pointees are separate. So how do you allocate a pointee?

Binky:Okay. Well, this code allocates a new integer pointee, and this part sets X to point to it.

Male Speaker:Hey, that looks better. So make it do something.

Binky:Okay. I'll dereference the pointer X to store the number 42 into its pointee. For this trick I'll need my magic wand of dereferencing.

Male Speaker:Your magic wand of dereferencing? That's great.

Binky:This is what the code looks like. I'll just set up the number and —

Male Speaker:Hey, look, there it goes. So doing a dereference on X follows the arrow to access its pointee, in this case to store 42 in there. Hey, try using it to store the number 13 through the other pointer, Y.

Binky:Okay. I'll just go over here to Y and get the number 13 set up and then take the wand of dereferencing and just —

Male Speaker: Oh, hey, that didn't work. Say, Binky, I don't think dereferencing Y is a good idea because, you know, setting up the pointee is a separate step, and I don't think we ever did it.

Binky: Good point.

Male Speaker: Yeah, we allocated the pointer Y, but we never set it to point to a pointee.

Binky: Very observant.

Male Speaker: Hey, you're looking good there, Binky. Can you fix it so that Y points to the same pointee as X?

Binky: Sure. I'll use my magic wand of pointer assignment.

Male Speaker: Is that gonna be a problem like before?

Binky: No. This doesn't touch the pointees. It just changes one pointer to point to the same thing as another.

Male Speaker: Oh, I see. Now Y points to the same place as X. So wait, now Y is fixed. It has a pointee. So you can try the wand of dereferencing again to send the 13 over.

Binky: Okay. Here it goes.

Male Speaker: Hey, look at that. Now dereferencing works on Y, and because the pointers are sharing that one pointee, they both see the 13.

Binky: Yeah, sharing, whatever. So are we gonna switch places now?

Male Speaker: Oh, look, we're out of time.

Binky: But —

Male Speaker: Just remember the three pointer rules. 1.) The basic structure is that you have a pointer, and it points over to a pointee. But the pointer and pointee are separate. And the common error is to set up a pointer but to forget to give it a pointee. 2.) Pointer dereferencing starts at the pointer and follows its arrow over to access its pointee. As we all know, this only works if there is a pointee which kinda gets back to rule number one. 3.) Pointer assignment takes one pointer and changes it to point to the same pointee as another pointer. So after the assignment the two pointers will point to the same pointee. Sometimes that's called sharing. And that's all there is to it really. Bye-bye now.

[End of Audio]

Duration: 55 minutes