Programming Methodology-Lecture19

Instructor (Mehran Sahami): Howdy. So welcome back to yet another fun filled, exciting day in CS106A. I don't know if there's any days actually we started where I didn't say that. I don't know. Someday, I should go back and watch the video. But they're all fun filled and exciting, aren't they? So it's not like false advertising. There's no handouts today. A little breather after the four handouts you got last time. And another quick announcement, if you didn't pick up your midterm already, you can pick it up. They're along the back wall over there in alphabetical order, so hopefully you can pick it up if you haven't already gotten yours. If you're an SITN student and you're worrying about where you can get your midterm, it will be sent back to you through the SITN courier, unless you come into class and you picked it up, in which case it won't be sent back to you because then you already have it. All righty. So any questions about anything we've done before we delve into our next great topic? All right. So time for our next topic. And our next topic is really a little bit of a revisiting of an old topic, kind of an old friend of ours, and then we're gonna push a little bit further. So remember our old interface. I always love it when math books say like "recall" and they have some concept, and I look at that, and I'm like, "Oh, recall the interface. Oh, what good times the interface and I had, like we were holding hands and running through a garden, the interface and I, and I recall our happy times together." So now's the time to recall the interface. What was an interface? Right? Last time when we talked about interface, we talked about something really generic, which was basically – it was a set of methods. And this was a set of methods that we sort of designate that some sort of classes actually shares. So it's a common set of functionality – common functionality among a certain set of classes.

And we talked a little bit about how yeah, there's – if you had some notions of [inaudible] classic standing in other class, you get sort of that same idea, but the real generality of interface is with – that you could have certain things that weren't related to each other in an object hierarchy or a class hierarchy that you still wanted to have some common methodology. And you sort of saw this all before, so in the days of yore when we talked about G objects. Remember? Those little fun graphical objects like the GLabel and the GRect, and all that happy stuff. And we said there were a bunch of interfaces there. Like for example, there was an interface called GFillable, and the GFillable interface had certain methods associated with it, like you could set something to be filled, or you could check to see if it was filled. And we said some of the objects actually implemented this interface, so for example, GRect, and GOval, and a couple others actually implement this GFillable interface. And there were other things like GLabel where it didn't make sense to have a filled or unfilled label, so GLabel didn't implement this interface. Okay? And it was just kind of a set of functionality. We're gonna kinda return to this idea of interface to talk about some of the things that we've actually done so far, like array lists, and some new concepts you're gonna learn, and how it relates to our notion of interfaces.

But the more general thing to take away from interfaces is in terms of thinking about how they're actually implemented for their syntax. Sometimes what you'll see in the actual code when you see that a particular class implements the methods that are considered part of some interface, the way we write is that is we say public class, and then we have the class name just like we usually do. So I'll write class name here, and I'll underline it to indicate that this is just a placeholder for the actual name of the class as opposed to writing class name. And then what we would write is implement – this would all generally be on the same line, much in the same way when you've seen before like your class, my program extends console program, or extends graphics program. Here we have a notion of implements, which is an actual word in Java, and then the name of the interface that it implements. So this over here would be the interface name. So you can imagine somewhere in the definition of the GRect class, there is a GRect class that implements GFillable. And GFillable is defined to be some interface somewhere which just specifies a set of methods, and any class that implements that interface needs to provide its own version of the methods. That's all it means, so for a class to implement some interface just means that that interface specifies some set of methods, and this class provides all of those methods. By providing all of those methods, it what's called implements the interface. Okay? And it's perfectly fine for a class to not only implement an interface, but also to extend some other class. That's perfectly fine, so the syntax is gonna get longer up here, but I just want you to see the implements syntax.

And then inside here, just like you would be used to – actually, I'll just draw the opening brace here and inside here – this would be all of your code for the implementation of the methods would go in there. So it's the same kind of syntax for writing a regular class, but now you're just specifically telling the machine, "Hey, this class is gonna support all the stuff from some particular or other interface," like GFillable or whatever the case may be. Okay? So that's kinda the concept. Now why do we sort of revisit this idea of interface is that now it's time to talk about some new things and the interfaces that they actually implement. So any questions about the basic notion of an interface or implementing an interface? Good call on the microphone.

Student: How's this different from – does this work?

Instructor (Mehran Sahami): Sure. Just press the button. We'll pretend it's working.

Student: How is this different from extending a class or just calling it an instance of another class?

Instructor (Mehran Sahami): Right. So that's a good question. The difference between this and extending a class is for example the notion of a hierarchy that we talked about. So if we have some class that extends some other class, we basically say, right – like when we talked about – remember in the very first day – primates and all humans are primates? We would basically say any human is a primate. The difference is there might actually be some things that are not primates, and humans may actually implement a class like the GIntelligence class, which means you have a brain that's larger than a pea or something like that. It turns out there's this other thing over here called the dolphin which also implements the Intelligence class. Right? We generally like to think of dolphins as – a point of debate. I don't how they actually measure this. There's like the dolphin SAT or something, the DSAT, and they're like [inaudible] – I don't know how they do it, but evidently someone has figured out how to measure intelligence in dolphins.

So you could say this dolphin class actually implements the methods of being intelligent, which might be some method like it responds to some stimulus in the world, but a dolphin's not a primate, so – at least as far as I know. Dolphin's not a primate, so there's no extends relationship here, and that's the difference. Right? So an interface in some sense provides more generality. When you extend the class, you're saying there's a direct hierarchical relationship among those objects. Interfaces, you're just saying there's this thing like the intelligence interface, and sometimes these get drawn with dash lines when you see them in diagrams. Both humans and dolphins provide all the methods of the intelligence interface, but only a human is a primate. That's the difference. Okay? So any other questions? Uh huh?

Student: Is the code in this instance the code for that particular class or the interface?

Instructor (Mehran Sahami): Pardon?

Student: You wrote code there as in – is that the code for that class or for the interface?

Instructor (Mehran Sahami): Yeah. So the code here is the code for class name, and somewhere else there would actually be the code for the interface. And that's covered in the book. We're not actually gonna be writing an interface together, which is why I'm not showing that to you here in detail, but the basic idea is just so that you would actually see that when a class implements an interface what that syntax would look like. Uh huh?

Student: Can you implement more than one interface?

Instructor (Mehran Sahami): Yes. You can implement multiple interface. For example, GRect implements both the GFillable interface and the GResizable interface, and that's perfectly fine. It's a good time. All right, so let me move on a little bit and talk about something that we're actually gonna deal with which is an interface, which is why we kinda revisit the subject. And the particular thing we're gonna talk about is something that's known as a map. Okay? So a map – and you might look at this, and you're like, "Map? Is that like oh yeah, the travel guide, like I got one of these, and I kinda look in here, and I'm like yeah, where was I going? Oh, here's a map of all the interstates on the United States, and I'm sure this is copyright, mapquest.com. There's this map, right? Is this what you're talking about?" No. This is not at all to do with what I'm referring to here as maps.

So when you think of the word map, you need to let go of what your previous notion of what a map might have been is, and think of the computer science notion of a map. So basically, all a map is – it's an interface in Java, which means it's some collection of methods that will implement some functionality. What are the methods? What is a actual map? What does it do? The way to think about a map is there's two key terms you wanna think about for map. One is called the key, and one is called the value. And basically, all a map is is it’s a way of associating a particular key with a particular value. You might say, “Whoa, man. That’s kinda weird.” Yeah, that’s a very abstract idea. So let me give you an example of a map that you’ve probably used throughout the majority of your life, but no one told you up until now that it was map, something called the dictionary. Anyone ever use the dictionary? The number’s getting smaller and smaller. No, I just let my word processor spell correct for me. A dictionary is a map. Its keys are the words that exist in the dictionary, and the values are the definitions of those words. So what a dictionary gives you is the ability to associate a particular key with a value. It associates words with their definitions. And the important idea to think about a map is in a map what you do is you add key value pairs – so like in a dictionary like the Oxford English Dictionary, right every year there’s a little convention, and they actually figure out some new words that are considered part of English, and they add them to the map that is the OED.

Anyone have a copy of the OED? Yeah, a couple folks. It’s a good time. Just get it. You get the little magnifying glass version where it’s got like four pages on one page, hurts your eyes, but it’s a good time. That’s what it is. We’re just adding key value pairs. Now what you do in a dictionary is when you look things up in a map, you don’t look them up by the value. Right? It would make no sense if I told you look up the word for me that has this as its definition. You’d kind of be like, “Yeah, Mehran. That’s just cruel and unusual.” The way a map works is you look things up by their key. You have a particular word in mind in a dictionary, you go to the dictionary, you look up that word, and then you get its corresponding definition. Okay? So what we like to think of a map in this abstract way it’s an association. It’s an association of a key to a value where when we enter things we enter both the key and the value together, and when we look things up, we look things up by saying get me the value associated with this key. Okay?

So dictionary’s a simple example of that. There’s other examples. Your phone book is same thing. It’s a map. The keys in the case of your phone book happen to be names that you wanna look up, and the values in the case of your phone book happen to be phone numbers, but it’s also a map. There’s maps kind of all around you. Everywhere in life, there’s a whole bunch of maps. No one told you they were maps before. And if I told you all before this class started, “Yeah, a dictionary and a phone book are really the same thing,” you might kinda look at me sorta weird and be like, “No, Mehran. One is like storing names and numbers, and the other one’s storing words and definitions.” But when you tell this to a computer scientist, they’re like, “Yeah. They’re both maps,” because now you know what a map is. All right. So any questions about the general idea of a map? Uh huh?

Student: [Inaudible].

Instructor (Mehran Sahami): As far as we’re dealing with right now, you can think of a key having just one value, but that value is really some abstract notion, right? So you could actually think we have some word – like let’s say we have a dictionary that has multiple definitions. Its key could be a word. Its value could be an array of definitions. So really, in the philosophical sense the value is one thing. That one thing happens to be an array that stores multiple values. So philosophically it’s one thing. In terms of the practicality of what you’re storing, you might actually be storing multiple things. And oh, maybe for Assignment No. 6, you might actually do something like that, but that’s not important right now because you’re still working on Assignment No. 5. All right? So [inaudible] something, you’re like, “Quick. Write that down.” Yeah, we’ll talk about it again when we get to Assignment No. 6. All right? So map is a particular interface, so that means we need to define some class that actually implements this interface, that provides some notions of these key value abstractions for us, and so in Java we have something called a hashmap. Okay? A hashmap is an actual class. It is a class that implements the map interface, which means everything that is a hashmap implements map. Okay?

Map is not actually a class in itself. Hashmap is a particular class. Map is just a set of methods that I think are actually useful to have. Now where did it get its name? Why do we call it a hashmap? What’s this hashing thing all about? And in the book in excruciating detail, it talks about hashing, what hashing is, and how to implement hashing, and hash codes, and all this other stuff. You don’t need to know any of that for the class. All you need to know is how to use a hashmap. You don’t need to build a hashmap. If you wanna build a hashmap, in CS106B a couple weeks from now, you will build a hashmap. You will have the joy of seeing the underlying implementation of hashmaps in C++ of all things. But for right now, you just don’t need to worry about it. The reason why it’s called a hashmap is it uses a particular strategy to store these key value pairs called hashing. That’s all you need to know, and because it’s called hashing, we call it a hashmap. That’s life in the city. Okay? But so this particular hashmap is a template. Right? Remember we talked about templates, and we talked about how the array list was a template, and you could have an array list of strings, or you could have an array list of integers, or whatever the case may be. A hashmap is also a template, but the key here is there are two types involved. It’s like two scoops of raisins. You have two types involved in the template for a hashmap. You’re gonna have a type for your keys, and you’re gonna have a type for your values. So what does that actually look like in terms of syntax? Let me create a hashmap for you.

Now that you know all about interfaces, we can erase this part. So let’s create a hashmap. The class is called hashmap. Because it’s a template, it has this funky angled bracket notation, but it’s gonna have two parameters for two types for this templatized class. The first type is what are you gonna have for your keys. So let’s say we wanna implement a dictionary. A dictionary for its key type is gonna have strings because it’s gonna be words. What type – the next type – so you have a comma in here, and then you specify the next type, the type for your value. For a dictionary again, if we assume a simple definition rather than multiple definitions which could be an array, we’ll also just gonna have a string for the value type. So hashmap string comma string, or sometimes we refer to this as a map from strings to strings because it maps from keys to values is how we would actually specify this. We need to give it some name, so we might call this “dict” for dictionary, and then this is going to – we are gonna create one by saying a new hashmap – and this is where the syntax gets a little bit bulky, but you just have to stick with it – string, string, and we’re calling a constructor, so again we have the open paren, close paren. And that will create for you a hashmap that maps from strings to strings.

Now we could also create a hashmap for a phonebook, so let’s create a hashmap for a phonebook. In the case of a phonebook, you might say, “Hey, names are still gonna be strings, but phone numbers” – well, phone numbers if they’re generally seven digit phone numbers – let’s just say it’s seven digit phone numbers. We won’t worry about the ten digit phone numbers. Seven digit phone numbers, I can store that in an integer. And so you might say, “Hey, I’m gonna have a hashmap of string to int.” And at this point you should stop because your compiler will give you an error. What’s the problem with this?

Student: [Inaudible].

Instructor (Mehran Sahami): Int is a primitive. Right? All these templative types require objects for their types. They require classes, which means we can’t give it a primitive. We have to give it the wrapper class integer. So it’s gonna be a map from strings to integer, and we’ll call this thing phone book. And this is just going to be – we’ll give you the syntax in full excruciating detail – hashmap from strings to integer [inaudible] the constructor, and that’s gonna create a hashmap from strings to integers for you. The first thing doesn’t always have to be a string. It can be whatever you want, right? You can have a hashmap from integers to some other integer, where you wanna map from integers to integers if you wanna do that. Some people do. I don’t know why. Sometimes you do. Now the other interesting thing is because a hashmap actually implements this map interface, sometimes the way you actually see this written is you’ll see it written without the hash in front. Okay? Just for the declaration. Here you still have to have the hash, and here you still have to have the hash, but just in terms of the type, sometimes it will be written map string to strings is a hashmap of strings to strings.

And you should look at that, and at first you get a little bit tweaked out because you’re like, “But Mehran, you told me a map wasn’t a class. You told me it was an interface.” Yeah. And a hashmap implements that interface. So what I’m saying is I’m saying I wanna create a map, that map that I’m gonna call a dictionary, and what’s the actual thing that implements the dictionary? It’s a hashmap. Okay? Because the hashmap implements the map interface, it will have all the methods the map expects, so any time I use this guy, and I say, “Hey, it’s a map,” and I call any of the methods for a map, it’s perfectly fine because a hashmap is guaranteed to have implemented those because it implements the interface. So just so you see this in code, sometimes you’ll actually see the hash out here, sometimes you won’t. Sometimes you feel like a nut, sometimes you don’t. You should see it both ways. It makes sense both ways, but you can’t drop the hash here because a map is an abstract idea. You can’t say give me a new abstract idea. Okay? That’s not gonna work. You can say give me a new hashmap. That’s the concrete thing. And what am I gonna assign that concrete thing to? Well, this concrete thing implements the abstract idea, so it’s perfectly fine to say, “Yeah, here’s the concrete thing. Use it in the place of the abstract idea.”

Any questions about that? Kinda funky, but that's the way it is. So when we have these key value pairs, how do we actually add stuff to our hashmap and pull stuff out of our hashmap? So there's two methods. There's a method called put and a method called get. These are the two most commonly used methods of the hashmap. And the way these things work is put – let me write them more toward the middle of the board. Put strangely enough – I know this is gonna be difficult to see, but put actually puts something in your hashmap. So the way put works is you give it a key and a value. The types of those things have to be the types corresponding to whatever object you've created, so if I wanna have – for example, I wanna put something in my dictionary, I'd send the put message to the dictionary hashmap, and I'd say, "Hey, add this key value pair," and key and value type strings to match this thing. Or if I have some phone book, I could say put key value where value better be an integer and key better be a string.

Besides putting things – it's kinda fun if you can put a lot of things in a hashmap, but it's not so much fun if you can't get them out. It's kinda like you had your phone book was kinda like your cell phone that you put all these numbers in, and then you tried to get a number out, and it just said, "No, I'm not gonna give you any numbers." You would probably quickly stop putting numbers in. Some people wouldn't. I don't know why, but I've seen that happen.

You're like, "You've seen people not be able to get numbers out of their cell phone?" Yeah, like their cell phone's busted and they just can't deal with that. But that's not important right now. What's important right now is you actually wanna be able to get values out. So out of our dictionary, you might say, "Hey, Mehran. I wanna get some particular word." I wanna get the definition for a word, really. This word is my key, so one way I could think of it is it's a word I'm looking up, but in the abstract notion, it's really a key. And so what this gives me back when I call get on the key is it goes over here and says, "Hey, dictionary. Do you have some value associated with the particular key?" So if this key exists in the hashmap, what it gives you back – so it will return to you the corresponding value. If that key doesn't exist in the hashmap, it gives you back null if key not found. Okay? So that's the way you can determine is this thing actually exist in my hashmap or not. If I try to do a get on it and it's not there, I'm gonna get back a null. Okay? So any questions about that? Uh huh?

Student: If the value's an integer, will it still give you back null?

Instructor (Mehran Sahami): Yeah, because integer is a class, so you're gonna get back object of type integer, and if you happen to look up something that's not there, null is a – it's the empty object. That's why you can't do int because there is no null int. Good question.

So let's create a little hashmap, add a couple values to it. So if let's say I had my phone book, I could say phone book dot put – always a dangerous thing to give out your phone number, but it's on the first handout. It's 7236059. Call. It's a good time. So that's just gonna put – this thing is an int, right? So in order for it to become integer, what's gonna happen is this is automatically gonna get boxed up for you as an integer, and then that's what’s gonna get stored. So this autoboxing is what’s happening to you, which makes it a little bit more seamless to use it with an int, but if you try to do this in an older version than Java 5.0, it’ll actually give you an error because it didn’t do the boxing for you. Notice there’s no dash in there by the way because then my phone number would be some negative number because it’d be 723 minus 6059, and that’s a bad time. So then we could add someone else to the phone book. Put Jenny – anyone know Jenny’s phone number?

Student: 8675309.

Instructor (Mehran Sahami): 8675309. It’s amazing how much longevity that song – like I remember listening to that song when I was in high school. It’s such a catchy tune. And if you don’t – you have no idea what I’m talking about, look it up. Just query 8675309. You’ll find it on Wikipedia. It’s not important. It’s really totally irrelevant, but it’s just fun. All right. So once I have these entries in the phone book, I could say something integer mnum, because that’s what I wanna be Mehran’s number, equals phone book dot get and give it Mehran. Okay? Important thing to remember is these are – the keys are all case sensitive, so if I put in a lower case M on Mehran here, it would return null because it wouldn’t be able to it. So keys are always case sensitive in maps. Just an important thing to think about, okay? So with that said, we can actually get a little bit more concrete with the hashmap. I need a volunteer. Come on down. It’s easy. You’re up close. You’re gonna be a hashmap. Here’s your hashmap. I’ve just created you. You exist. Rock on. All right. So I hope you don’t plan on taking notes any time soon because it might be difficult as the hashmap. So what we’re gonna do is we’re actually gonna enter – we’re gonna call these commands on the hashmap so we can actually store these things in there, and then see what other commands – other things we can do. So normally the thing that gives bulk underneath the hood to a hashmap is our friend the Ding Dong.

So what we’re gonna have is certain things that we’re gonna enter in our hashmap, which is basically a Ding Dong sandwich, and on one side we’re gonna have the key, and on the other side, we’re gonna have the value. So what I’m gonna do is I’m gonna write the keys in black and the values in red so we can keep track of them. That way we won’t get them confused in our minds. So the first key I’m actually adding is I’m gonna add myself. Mehran – I guess I could’ve written this before, but I didn’t, so I have my key. And that key has with it a value, and the value sticks with it, so 7236059 – I shouldn’t have put the dash in there, but I accidentally put the dash in there, and I say put. And this goes into the hashmap. Okay? At this point, I have no notion of what’s in my hashmap, or ordering, or whatever. I just created something. I tossed it in the hashmap. So then I say, “Well, Jenny, she was always a good time.” You’ve gotta listen to the song, all right? If you have no context for the song, you’re like, “That’s not funny.” 8675309 – we put Jenny in the hashmap. Now that we have these things in the hashmap, there’s an interesting thing that comes up, which is when I had an array list, I knew how many things were in my array list. I could refer to the first element. I could refer to the second element. What’s the first element that’s in my hashmap? You’re like, “Well, you put in Mehran first,” but it’s not guaranteed to necessarily be the first element. A map has no intrinsic ordering. It’s all just kinda mushed in there. So there are some things we might additionally wanna do on the hash map to give us some notion of what’s in the hashmap and how big the hashmap is.

So here’s a couple more methods, and you could just bear with me. I swear it will be over soon, the pain and the agony. No, that’s a good time. All right. So a couple other methods – I might have to ask you to just move over slightly – is we can remove a key. So if I ask to remove a key, what I pass it is the key, so I might say something like phone book dot remove and I give it some key. Now if that key exists in the phone book, it removes it from the hash map, and it is now gone. So if I call remove on Mehran – oh, hash map, I need Mehran. So you look somewhere inside the hashmap and you say, “Oh, here’s Mehran. Remove. Phone number goes with it.” That’s just the way it is. The key and the value always stay together. Now besides removing something – and if Mehran didn’t exist – so I could actually say remove Bob. You look in there. There’s no Bob. You don’t need to do anything with the hashmap. It’s sort of this simple operation. There’s no exception thrown, nothing like that. I asked to remove something. It’s not in there. We just keep sticking with it. But what I can actually ask is, “Oh phone book, do you contain a key?” And so there’s a method, contains key, that I give it a key and it returns to me a Boolean. So I can ask you, “Oh phone book, do you contains key Jenny?”

Student: True.

Instructor (Mehran Sahami): True. Good times. Phone book, do you contains key Mehran?

Student: False.

Instructor (Mehran Sahami): False because the key’s been thrown out, right? We already removed that key. And last but not least, we can ask the phone book for its size. So this will return an int. Phone book, what’s your size?

Student: One.

Instructor (Mehran Sahami): Excellent. Thank you very much. And you’re done, phone book. Thank you very much. Nice work. And for that, we’ll just – you and everyone around you. It’s an array list of Ding Dongs.

Student: [Inaudible].

Instructor (Mehran Sahami): All right. So that’s the way you wanna think about it. It’s this bag. You put stuff in the bag. You take stuff out of the bag. Whenever you put stuff in the bag, you put them in in pairs. When you take stuff out of the bag, it’s always pairs. And all the work that you’re doing is always in terms of your key, but the value is associated with it. So when you wanna remove stuff or check to see if it contains, you never look based on the value. You always look based on the key. That’s just the way things are. Now this whole notion of maps is part of a bigger framework, so if we can get the computer for a second, I’ll show you the bigger framework just real briefly. There’s this notion called the collections hierarchy, which is – you're like, "Oh my god. It's big. It's huge." No.

Most of the stuff you don't need to worry about. There's some of this stuff you've already seen. An array list you've seen. An array list is something called an abstract list, which is just the abstract concept of a list, and that's part – that actually implements an interface called list, which implements an interface called a collection. So array list, this thing you've been using this whole time, we didn't tell you until now is actually something that we could refer to as a collection in the same way that we could refer to a hash map as a map. Now there's a few other things in here that we're just not gonna deal with in this class, like a tree set. Or you might notice there's something over here called a hash set. Hash set and hashmap, not the same thing. So you should just know that. You don't need to know what a hash set is. You just need to know that it's not a hashmap because the words aren't the same. If they were the same, then it would be called a hashmap. But there's this notion of an abstract set that they are subclasses of that implement some abstract notion of a set, so there's this big kinda complicated hierarchy, but the important thing to remember is all of this stuff at the end of the day is a collection. So there might be some operations we might learn how to do on collections that applies to all of it because all of these things at the end of the day implement the collections interface.

There's a same kind of notion for hashmaps. The hierarchy's just a little bit smaller so it's easier to see. A hashmap is this notion of an abstract map, which we talked about. It's kind of an abstract concept that has key value pairs, and what really defines that is an interface that's called a map. Now it turns out that besides hashmaps there's another kind of map called a tree map, and a tree map also allows you to put things into it and get things out of it. The underlying implementation is just a different way underneath the hood of actually implementing a map. And that's the whole critical concept here. The critical concept is the abstraction that the idea of a map interface gives you. When you think about something that implements the map interface, all you need to worry about is what are the methods that are involved in a map. Underneath the hood, hashmap may have some different methods than tree map for some specialized kind of things, but we don't care. As long as we just treat everything as a map, someone someday comes along and there's the hashmap, the tree map, and then they create the super cool funky map. And you're like, "Wow, the super cool funky map is just way more efficient than all the other maps that existed," because someone had this great idea about how to implement maps like in 2027.

Well, if you go back through your code and everything's written in terms of a map interface, and super funky cool map – was that super funky efficient map? Gotta get the name right – actually implements the map interface, none of your code changes, except in one line when you create your map. Rather than saying map string string hashmap, you say map string string super funky cool map. And everywhere else in your code when you refer to dictionary, you're just using the methods of the map interface and no other code changes. So that's the power of thinking of the notion of object oriented programming and object oriented design in these abstraction hierarchies is the fact that as new implementations come along – and they really do, right? It's not like computer science is a static thing. Ten years from now, there will be other implementations of these things which will be more efficient and better than what exists now. But if we relied on always thinking about hashmap, then our code would be much more rigid than if we just think about this abstract concept of map. Any questions about that? All right. So think abstractly. That's kinda the bottom line from that whole little diatribe. So the one thing that we wanna think about is when we have these things like maps and array lists, and I told you an array list is part of the collection, what do I actually get with these things? What does a collection buy me other than the fact that I get to draw this complicated picture and be like, "Oh, look. This is a collection framework?"

What it gives you that's interesting is a notion of what we call an iterator. What is an iterator? An iterator is just an idea basically. It's a way to list through a set of values. What does that mean? What that means is if I have some array list, and that array list contains like ten different items in it, one way I could think about that array list is having some for loop that counts from zero up to nine, and I get each one of the individual values. Another way I can think about that array list because an array list is really a collection is if all collections allow me to have an iterator, this iterator will allow me to very easily – and I'll show you the syntax – have a sequential process of going through all the values in my array list. And for an array list, it's not as exciting as some other things because you're like, "But I already have a way of going sequentially through my array list." So let's just compare and contrast them. The idea when you think about having a list of values and having an iterator is – first of all, let's create an array list, and I'll show you how to get an iterator from it. So let's say we have an array list of strings. And we'll call this names because it's gonna just store a bunch of names for us, and so we say new array list. It's an array list of string. We're calling a constructor so we have the prints. Then what I say is, "Hey, array list. I want some way to be able to go through all your elements in a very easy way in an abstract way because I don't wanna have this grungy syntax of asking you for your size and having a for loop." That's something that's very specific to an array list. What I really wanna do is kind of have this generalization to say, "Hey, you contain some set of values, and I just wanna go through each one of your values one by one." So the way I do that is I create an iterator. An iterator has a templatized type because if I have an array list of strings and I wanna go through all the values one at a time, the values are all strings, which means I need to have an iterator that iterates over strings. So the types are generally the same. If I have an array list that I'm gonna iterate over, its iterator is always gonna be the same type as the type of the array list.

And I'll just call this it. And it, which is – oftentimes you'll see iterators often called I or it just for shorthand – is named dot, and now it's time for you to see a new method of the array list, iterator. So what this method does is it says, "Hey, array list. Give me an iterator over your values." And what you get is something whose type is iterator of strings. Now how do you actually use that? Here's how you use it. So what you're gonna do is you're gonna have a while loop. And the way an iterator works is you can basically ask the iterator two questions. You can say, "Hey, iterator. Are there any values left in you? And if there are, give me the next values." So the way you ask to see if there's any values left in the iterator is you say, "Hey, it. Do you have a next value?" so has next. That returns to you a Boolean if it has a next value or not. And then the way you get the next value interestingly enough is you just say, "Hey, iterator. Give me the next value," which returns to you whatever your type your iterator's templatized type is. So if you have an array list of strings, and you have an iterator over strings, when you call it dot next, you will get a single string. So if our array list over here – let's just say names – contain the names Bob, Alice, and a C name, Cal. Not very popular, maybe it'll come into vogue someday. Name your child – I was thinking, yeah, I should name my child Cal Sahami. That would be so wrong on so many levels, but that's not important right now, besides the fact that my wife would probably beat me to death.

But what this really gives you – names is this array list is when I create the iterator – you can think of the iterator sort of sitting at the first value. And I say, "Hey, do you have a next value?" and it says, "Yeah, I do." It doesn't give it to you. I just says, "Yeah, I have a next value." "Oh, give me your next value." So it gives you back Bob as a string. Bob still is part of your array list. It doesn't go away, but now your iterator sort of automatically moves itself to the next element. So when you say, "Do you have a next element?" "Yeah." "Give it to me." You get Alice. And it moves itself down, and then yes for the third value, and you get Cal. And after you get Cal, you say, "Hey, iterator. Do you have a next value?" So after it's given you Cal, it's sort of gone off the end, and it says, "I don't have a next value." And that's why you always wanna check if it has a next value before you ask for it because if you ask for a next value when it does a next value, that's bad times. And if you really wanna know what that does, just try it in your code, and you can find out. It's always good to try to try some of the error conditions yourself, but this is the simple idea.

Notice there's no notion of asking the array list for its size, or anything like that, or needing to know that the array list is actually ordered. And the important thing there is because there are sometimes – there are some lists like array lists which are ordered. We know that Bob is element zero, Alice is element one, Cal is element two. But we just talked about our friend the hashmap, and now you're probably thinking, "Yeah, Mehran. Why were you talking about all this array list stuff when you're talking about the hashmap?" Because the hashmap didn't have any ordering over its keys. And so one of the things we can do is we can ask the hashmap, "Hey, what I want is an iterator over your keys." So let's see an example of that. So in terms of thinking about the hashmap – and we could've done the same thing with a for loop in the case of an array list. It wouldn't have been very exciting, but I would completely trust that you could probably do that, or maybe you already did for your hangman assignment. Okay?

When we think about a hashmap, a hashmap by itself doesn't – is not a collection, so it doesn't provide you with an iterator directly. And the reason why it doesn't provide you with an iterator directly is because if you were to say, "Hey, hashmap. Give me an iterator," it would say, "But I store key value pairs. I can't give you an iterator over key value pairs because an iterator is only defined by one type. But what I can give you is I can give you a set of my keys, and that set of keys is a collection, so you can ask it for an iterator." So the way that works is if I have some hashmap – let's say I have my phone book hashmap, I can say, “Oh phone book dot” – and the method is called keyset. Lower case K – it’s sometimes hard to draw a lower case K, upper case S, so what that gives you back is a set of just the keys, so just the strings of the names in the phone book. And then you can ask that keyset dot – so this would all be on one line. “Hey, keyset. You’re a collection. Give me an iterator.” So you would say iterator. And what you’re gonna get back is you’re gonna fum phone book. “Phone book,” you’re gonna say, “What’s your keyset?” and you get a set of keys. What’s the type for the keys of phone book? Not a rhetorical question. Strings. So if I say, “You’re a string set. String set, give me an iterator,” what you get back is an iterator over string, so I can assign that over here by having iterator of strings, and I’ll just call this I equals phone book dot keyset. I need to have these parens in here because I’m actually making a method call. This looks a little bit funky, but the way to think about it is phone book dot keyset is returning to you a object which is a set of keys, and then to that object, you’re sending it the iterator message which is giving you back this iterator of strings. Okay? So once I get back this keyset from the hashmap and get its iterator, this exact same code over here works on that same iterator. So I could use this code on an iterator that was generated from an array list, or I could share exactly that same code from an iterator that comes from the keys of my phone book. So if I had my phone book that had Mehran and Jenny in it, what I would get is an iterator that’s going to iterate over Mehran and Jenny.

It has no notion of the actual values that are associated with those keys because the iterator is just over the set of keys, not the corresponding values. Any questions about that? The other thing that’s funky, just to point one thing out, is in an array list, an array list is ordered, so I’m guaranteed to always get my set of keys from the iterator in the same order as they’re stored in the array list. So I’ll always get as the first key whatever’s at index zero. The second key will be whatever’s at index one, and so forth. Hashmap has no ordering. Even though I entered Mehran first and then I entered Jenny second, there is no actually no reason why Jenny couldn’t show up as the first value. The only thing I’m guaranteed is that every value will show up once, but I get no guarantee over the ordering. So no order, and this one you always get order, just something to keep in mind. Was there a question? Or was that the question? Yeah, just for thinking of it, one step ahead of me. So one other piece of syntax that’s a little bit interesting to see that you should also see – so any questions about the notion of a keyset or an iterator? If you’re feeling okay with iterators, nod your head. If you’re not feeling okay with iterators, shake your head. All right, question back there?

Student: Does keyset return an array or an array list? What does keyset return?

Instructor (Mehran Sahami): It actually returns a set, and as far as you need to worry about it for this class, you don’t need to worry about a set, so the only time you’ll actually use the keyset syntax is with an iterator to get its iterator. But it actually is returning an object that is a set, or it actually implements the set interface. Okay? All righty. So one last thing to see, and then I’ll show you all of this kind of stuck together in some code. Java actually gives you – because iterators have become such a common thing to do, like people were writing this code and left right – they were writing this code until the cows come home. And they were like, “Ah, but it’s such a pain. I gotta create the iterator, and then I go through all the values, and that's such a common thing to do that I wanna go through the values, wouldn't it be nice if there were some simpler way of doing this?"

And enough people sort of yelled and screamed this over the years that the people who created Java 5.0 said, "Okay. We're actually going to give you a version of the for loop which is totally different than what you've seen so far." So now you're old enough to see the new improved for loop. And the way this new, improved for loop looks like is you say for, you specify the type of the thing that you're going to get as every element of the loop – so in this case you would say something like string name. Then you give a colon, and then you specify over here the collection over which you would like an iterator. What does that mean? That means the collection that you would like an iterator over is whatever thing you are calling the iterator method on. You do not actually call the iterator method. You just specify the thing that you would've called the iterator method on. So over here what we would actually have is phone book dot keyset. Phone book dot – this would all be on the same line – keyset. Okay? And then inside this loop, what you can do is this word, this name of this variable is a declaration of a variable that sequentially will go through all of the elements of this collection over here. So this collection could be the keyset from a phone book. It could actually be an array list because over here we were asking the array list directly for an iterator. So we could say for every string and name colon names, and that means give me the name one at a time out of names, and inside here we could do something like print lin name. So name is just a variable. The way you can think of name is name is getting the value one at a time of an iterator over whatever you specified here. But the funky thing about this is you never created the iterator. You never said, "Hey, I need an iterator." You just said, "Yeah. This thing? You can get an iterator from that. And I wanna get things one at a time from this iterator, and the name I'm gonna specify for that thing getting one at a time is this variable over here, and it's gonna have this type."

So this syntax, Java automatically what it will do is construct for you basically underneath the hood – you never need to worry about the syntax stuff for that – it will create an iterator for you. It will automatically check to see if the iterator has a next value. If it does, it will assign the next value to name, and then execute the body of your loop. And after it executes the body of your loop, it comes back up here, checks to see if iterator has a next value. If it doesn't, then it immediately leave the loop. If it does, it gets the next value, sticks it into this variable – so it clobbers the old value, but it sticks it into this variable, and then you can do whatever you want. So this will actually write out the whole list of keys from the phone book. And this only exists in Java 5.0 or later. And sometimes people refer to this as the for each construct, but the name's not as important as just understanding the syntax. And if you try to do it with something from Java in an earlier version than 5.0, it just doesn't work. So any questions about that? So let me show you some code that kinda puts this all together, a little code. So here's a little hashmap example. What I'm gonna do, and I put in a little bit of print lins here, just so you could see – like remember we talked about our little lecture on debugging? Where we said, "Hey, you know what? Before you call a function, just add a print lin. You know you got to that function. That's all you need to do." Then you know before you're gonna call this function, I'm about to call that function. It's the simplest form of debugging if you didn't have the full power of Eclipse available to you. So what we're gonna do is we're gonna read in phone numbers by reading in a list of phone numbers. What are we gonna read them into because we're not passing a parameter here? We must have some instance variable. So down here I have an instance variable that is – oh, and I forgot private in front of it. My bad. So it's a private instance variable.

It's a map from strings to integers because this is our friend the phone book that we just talked about, and I'm gonna create for this map – notice I'm just saying map here. The actual implementation I'm gonna use is a hashmap from strings to integers. So it's the syntax you saw before, before I erased it. And then what we're gonna do is to read in the phone numbers is we're just gonna ask the user, "Give me a name." So we're gonna have a while true loop. Give me a name. We read in that name. If the name is equal to empty string, then I know the user's done entering names. And if they gave me a valid name, then I say, "Hey, give me a phone number as an integer." So I get the phone number's integer, and then I will add that name number pair to the phone book. Notice because this is an int, there's boxing going on here. This number's automatically getting boxed up for you to go from being an int to an integer so I can stick in a string and an integer into the phone book. Again, that only works in Java 5.0 or later, but that's what you should be using. So after I read in all these numbers from the user, I'm gonna allow you to look up numbers. And the way I'm gonna look up numbers is I'm gonna say – I'm gonna keep looking up numbers until you tell me you're done. So I'm gonna ask for a name to look up. If you're a done, you'll give me a name that equals the empty string to indicate that you're done looking for names. Otherwise, what I'll do is I'm gonna ask the phone book to get that name. What it's gonna give me is it's gonna go look up the name. It's gonna go look up Jenny in the phone book and say, "Hey, Jenny. Are you there?" And if Jenny's there, what I get back is the value associated with Jenny, 8675309, as this integer. I don't get back an int. I get back an integer. And this is critical.

I shouldn't declare this as an int here, and the reason I shouldn't declare this as an int is because there's a chance that number could come back null. So if I say, "Hey, phone book. Give me the number for Bob," and it's like there's no Bob in the phone book, so here's null. I can't treat that null like an integer. I can't box and unbox the null, so I need to specifically check for that, and then I'll say that name is not in the phone book. Otherwise, what I'll do is I'll print it out. When I print it out, then it does the unboxing for me and prints out the integer. So that's look up number. Last but not least, I wanna display all the phone numbers at the end that are in the book, so what I'm gonna do, I can actually show you two versions of that. One version uses our happy friend the iterator. It says, "Oh, phone book. Give me your keyset, and from the keyset give me an iterator, and I will assign that to it which is an iterator of type string." And then I'm gonna do the loop that you just saw before. While the iterator has a next value, I'm gonna get its next value which is a name. All the iterator is just iterating over all the keys which are names in the phone book. To get the associated number, I need to say, "Phone book, get the number associated with that name, and then I print it out." Note here I don't check for an error condition because I know all of the names that I have are valid names in the phone book because I got them from the phone book, so I know when I go look them up, they'll always be there. And the alternative way I could've done this is using this for each syntax, which is just so lovely, right? I don't create an iterator. I just say for every name that's in my phone book keyset, get me the number by looking up that name in the phone book. And then I just print it out.

So just so you believe that this actually works, and then we'll go in our final minute together, so we put in Mehran, 7236059, I won't give you my home number ever. Just kidding. I did one quarter. That was a mistake. Jenny, 8675309 – I won't ask you to give me a number because it will be [inaudible] like random people will call you and be like, "Is this your real number?" Because it turned out when that song came up about Jenny, a lot of people called that number, and there were people who got really angry because some people really have that number. So we're done entering numbers, so we're done entering names, and we wanna look up someone. I wanna look up Mehran. Mehran's not in the phone book? Case sensitive. I need capital Mehran. So Mehran is in the phone book. I'm done looking folks up. And then it displays the phone numbers. Notice it displays Jenny first even though I entered Jenny second because a hashmap has no guarantee of ordering over the iterator, important thing to remember. So any questions about anything you've seen? All righty. Then I will see you on Friday.

[End of Audio]

Duration: 50 minutes