Programming Methodology-Lecture24

Instructor (Mehran Sahami): So welcome back! So yet another fun-filled, exciting day of 26A. After the break, it almost feels like – I came into the office this morning, not that I wasn't in the office during the entire break, but I came into the office this morning and it felt like a new quarter had started. And I was like, oh, it's been a whole week. And I'm sure for you, it feels like you just wish a new quarter was starting because we still have two weeks left.

So a couple quick announcements before we get into things. One is there is one handout, which is your section handout for this week. And kind of one of the themes of this week is bigness. In some sense writing bigger programs, bigger data structures, that's the whole deal. And we'll kind of talk about that as we go along.

Another quick announcement, just wondering how many people tried the Name Surfer demo online and had a problem with it? You folks, yeah, we updated it. So evidently there was some issue that only shows up on Windows XP with Java 1.6. And like, if you had a Mac you didn't see it, if you had Vista, presumably, you didn't see it, if you had Java 1.5 you didn't see it. But in that one case, it would come up, so the name suffer web applet demo was updated a few days ago, I think on Friday, maybe. So now it should work for everyone, hopefully. If you still have an issue, let me know. The only thing that you'll see now, though, if you'll try running this applet, is that the interactors, instead of it being on the south border of the screen are on the north border of the screen. That was just a little hackler we had to put in there to get things to work. The functionality is exactly the same.

As a matter of fact, if you want, you can put your interactor and border on the north instead of the south. It'll make no difference to the rest of your program other than where you say south for adding your interactors, you say north, that's the only place it makes a difference. But you actually see that in the web applet version, the interactors are just in the north border instead of the south. Otherwise, it doesn't make any difference. But in case you saw that and was like freaked out, there's nothing to worry about. Okay. So also, I hope you had a good break. Just wondering, how many of you actually enjoyed their week break? Good time. And how many were working most of that break? Yeah. Good times. Hopefully, it didn't cause you too much pain, but if it did, hopefully, you're like, all caught up or ahead of the game in all your classes, now, so life is good.

So I want to spend a little bit more time talking about today, well, actually a lot of time talking about today, is thinking about data structures, building large-scale data structures. And we begin to talk about it just a little bit before the break and it's been a while so we're going to review it a little bit and kind of build up even more. But one of the things we talked about, in the past, right, what a lot of our computers do is they manage data. They manage lots of data. And in fact, I would venture to guess that there's a whole bunch of applications out there that manage a whole bunch of data about you, but you may not have thought about all the data they actually manage. So some of the things that actually come up, for example, online stores, right. Anyone actually bought anything online, just wondering. Yeah.

There's a huge amount of data's that involved with that. Not only the particular transactions you make when you buy something, but keeping track of accurate transactions, figuring out things like people who buy product X also, tend to by produce Y. All of that is data management. And what makes those companies successful is they just do a very good job of managing their data. Okay. There are other things like, I'm almost frightened to ask, but social networks, like, Facebook, or MySpace, or ORCHID, or Friendster, or Linkdin, or you could just keep going on. Anyone on a social network? Just wondering. Yeah. That's good because your next assignment is going to be to implement one so you can see what it's actually like. But that will be coming in a couple of days. And they're not that hard, really. But what it is is a data management problem. Right? And it keeps track of things like who you are and information in your profile in the social network, and who your friends are, and all that happy news.

Or you know, even things like a friend web search, right? There's a huge amount of data you need to be able to keep track of to be able to web search, right? So all these things are all about managing data well and so part of this class, right, is you've got a whole bunch experience in terms of building up code, and different kinds of classes, and doing nice things with user interfaces, and the whole deal. And one of the things that we need to spend a little bit more time on is talking about how do you manage lots of data and then do something interesting with that. Okay. So here's some principles to think about, if we think about good software engineering, some of the principles of thinking about data kind of in the large. Okay.

When you think about keeping track of lots of data, one of the things you want to think about is the information you want to keep track of, what are the nouns you want to keep track. And you're like, I don't want any nouns, what do you mean by the nouns? Let's say I was writing an application that was an online store, to keep track of, oh, let's say, music. And so one of the things I would want to think about is, where are the nouns that are associated with music? You're like, okay, now you're really getting weird. No, it's pretty simple. Things like a song, right, is a noun, that's associated with music, or an album, or an artist, right. And so what you want to think about is the things that are the nouns in the domain that you're dealing with oftentimes end up translating into what your classes are. So you may end up having a class that keeps track of information about a particular song or class that keeps track of information about a particular album.

So the good linguists out there tell me we not only nouns but we also have verbs, not unless you happen to be talking to my son, who seems to only have nouns, but that's a different story. And he loves jarens by the way. But like, why are you telling me this? Just cause it's fun, because I just spent a whole week dealing with it. In terms of verbs, these are oftentimes the methods that are associated with your classes, right. So when you want to do something, some noun takes some action, which is a verb, which is some class, has some method that operates on that class. So at an abstract level that's what you want to think about in terms of high-level principles of design. Now, there are some other sort of more concrete things that you might want to think about, things that have to do with what are the characteristics of the data you actually want to store so one thing that comes up, oftentimes, is thinking of the notion of having a unique identifier, identifier. What do I mean by unique identifier? All of you have unique identifiers, whether or not you like it or not, as a result of being at Stanford. Your Stanford University I.D. number is a unique identifier for you at Stanford. Every student has an I.D. number. Okay. So it identifies you and it's unique. No two students share the same I.D. number.

So you get issued this number when you show up here and you have it for life. When you leave it's still with you. I know, I left, I came back, I have the same student I.D. number. It just exists and this uniquely identifies you. And in different cases you might want to think about what are unique identifiers. Right. So in some cases, for example, if you had a social network you might consider the names of people and not the social network to be identifiers, or say the names of their profiles, for example. In other cases, you might have something different. If you're managing a store you might have some I.D. number for books, an ISBN number, or if you're keeping track of music, you might say that the combination of the songs' name and the band that plays it is a unique identifier for that song. In some cases the unique identifier can be a combination of things. But if you think about your data having a unique identifier that also gives you some insights about what kind of data structures you might want to use to keep track of certain things. On other unique identifier that some of you've already grapple with is saying Name Surfer. Right? If you think about the data in Name Surfer what's the unique identifier there?

Student: Name.

Instructor (Mehran Sahami): Name, right? Name is a unique identifier and for every name you have some list of values associated with it, which was the rank of that name over the last century in terms of how popular it was for names. But every name, well, I shouldn't say every name has some value associated with it, but every unique identifier in the system has some value associated with it and only one set of values. And so the important thing to keep track of there is when you actually are doing your Name Surfer assignment to fact of this thing is a unique identifier can potentially help you keep track of the data that you're using. And we'll sort of go into that as we go along in the class. Okay?

So some other principles we can kind of think about in terms of designing data structure, in terms of actually doing the design, there's some questions you want to ask yourself. And the questions you want to ask yourself is, are you keeping track of some collection of objects? Right? So there comes some collection of objects through data that you want to have. And if you have a collection of objects, say in an online music store, you might have a collection of songs that you want to keep track of, this word should be a tip off too, that perhaps there's an interesting collection that exist in Java that would be a way of keeping track of that information. It may not be in Java if you're programming in some other language. But the fact that Java has something called a collection and the reason why they gave the name of collections to a certain group of stuff is because they're used to keep track of a collection of objects. And the question that you ask yourself then is what collections do you actually want to use? Okay.

So with that said, what we can do is spend a moment, and it will be a brief moment, revisiting the collection hierarchy. Right? You've seen this picture before but I'm just showing it to you again, because the last time you saw it was like two weeks ago, which is a lifetime in a quarter. Right? I think it too, it's about a fifth of the quarter. You're like, oh, what was I doing two weeks ago? Was the break out, was I learning Printland? No, no it wasn't that long ago. But what you were learning about, a little bit, was collections. And o there are some collections, for example, like an ArrayList that going all the way up the chain of the hierarchy is itself a collection. Or there are other things, for example, like a HashMap. And a HashMap, if you said hey, I have some HashMap, the set of keys in that HashMap ends up actually being a set, which happens to be a collection. Okay. And so what you want to think about, do I have different things that I can keep track of? Like an ArrayList is one way to keep track of things. A HashMap may be another way of keeping track of things. When is the appropriate time to use one thing versus another? And so when you want to think about the appropriate times of one versus the other, you want to think about what are the methods that a collection provides to you. And it turns out all collections that implement the collection interface, like the ArrayList or the key set of the HashMap, have all of these properties.

And some of you have seen them before, but just to review. You can add a value, right? So this is a parameterized values type. Like you can have an ArrayList of strings and you can add some value to it, and it adds it to the collection, and little did you know, or maybe you did know, but at the time we didn't' really care about it, was it returned a bullion. Most of the times we just returned the bullion, we didn't care about it. But actually returned true the collection changed. So in an ArrayList, it always returned true because when you were adding a value, it didn't care about duplicates, it would always just add them to the end and always return true. Some collections, like sets, actually don't allow you to have duplicates. So if you try to add something to a set that already has the value you're trying to add, it will not change the set and return false, because it says, hey, I already have that value and nothing changed.

A couple other things that you should know about, most of these you've seen. Remove, removes the first instance of an element as it appears and returns true if a match is found or returns false if it didn't find anything to actually remove. And clear, basically, just sort of nukes the whole collection. It just says get rid of everything in the collection. I'm done with that and the collection is dead. Actually, the collection is not dead to you, it still exists, it's just an empty shell of what it was before. There are violins playing in the background. And then size, you can get the size of the collection. You've seen this, you've probably used a lot of these before in your programs. Contains, that's an important one, right? You actually want to see if a collection contains some particular value, if a collection is empty. And here's one that's sort of interesting that we talked about a little bit but we didn't actually talk about the fact that a collection or all collections can give you one these. All collections can give you an iterator.

So we talked about, for example, having an iterator over the key set of HashMap. That's one thing we did before. We said we had some HashMap that lets a map from strings from some other strings. And we want, say, hey what I want to see is get a set of all the keys and I want to iterate all over those keys. That's great! You can do that and that's perfectly fine. When we used ArrayList we always had like a four loop and said, oh, from zero up to the size of the ArrayList do something. But if we actually wanted to, we could have an iterator over the ArrayList and then this would give us the elements of that ArrayList one at a time. So because an ArrayList is a collection it can also give us an iterator. And that's just something to keep in mind is that there's common patterns that get used in programming. One of the common patterns that get used is known as an iteration pattern, which again, is an iterator over some collection and you just go through and do something like printout the values for every element of that collection. And if you want to write it in the most general case, you don't care if that collection happens to be the key set of the HashMap, or an ArrayList, or whatever, you just say, hey, you're a collection, give me an iterator and I can go through all your elements one at a time and, for example, print them out. Okay. So there's just simple pattern's that we get into.

Now, you're like, okay, Marilyn, that's fine, you told me some design principles over here, you told me about some collections over here. Show me something concrete, like put it all together. So let's actually put it all together. Okay. And we'll view a little example, which is going to an online music store. And because many names for online music stores are already taken, our music store is going to be called Flytunes cause they're tunes that will fly. All right. Yeah, man, when you're like in your mid 30's you just can't be that cool. But trust me, it is. Okay? So we're going to make a little store that just keeps track of music and albums, and that music and actually lets us keep track of information and prices. And so what we want to think about is what are the things that we actually are going to do in that store, okay? So one of the nouns of that store is going to be a song, okay? So a song is some basic thing that we're going to sell. This is what we want to be able to do with the song. Now, you could say, well, what does that mean, do I have some method called sell? If we're doing inventory management we might not actually have a method called selling a song, but we might, for example, want to add for inventory to do things like add songs.

And similarly, songs, oftentimes, are put together into albums. Okay, so we may also want to keep track of albums and do things like add albums to our inventory. Now, the interesting thing with an online music store that differentiates it from say a physical music store, is you can do interesting things, right? You can actually have songs that are not on any albums. And that works, right? It's kind of like thinking of a single, right. When you go and buy a single somewhere. In the days of yore, you could actually buy a little record single that had two sides on it so you got two songs, so it wasn't really a notion of a real single, single. I guess now, there are like CD singles. But who wants a CD single when it comes down to it? You can get songs that are on albums. At the same time, you can have the same song be on multiple albums, right? That always happens. There's a band, I won't mention their name, but I remember from the early '80s, they had two albums. They had their first album and they had their best of albums, which were half the songs from their first album. Just anything you can do to milk the consumer. But basically, what that meant was songs can show up on multiple albums. Okay, so we want to begin to think about how that might actually affect our design.

Now, if we think about putting the information together, right, nouns become our classes. So if we're going to have song as a noun, we're probably going to have some class song that's going to keep track of all the information associated with a song. And so just for the sake of brevity, I'll tell you what information's going to be associated with songs that we care about in our store. There's a notion of the name of the song, the band or artists that perform that song, and then a price, because we're going to allow for songs to be sold individually, so individual songs, as opposed to whole albums, have prices. Okay. And you can think about these things and think about, oh, what data types do you want to have for them. Right, so what type data type makes sense for a name, for example, string type, or if you want to have a band name, this would probably be a string. Price is always an interesting one. You could sort of say, well, now, and there's multiple things I could have it be. I could have it be an [inaudible], for example, if I was going to have it be the number of cents. In the simplest case, I'm just going to have it be a double. Even though we know there's no fractional money unless you're a banker, in which case there is fractional money. But we won't talk about that right now.

It was just like Superman III. Anyone see that movie? No. It's not worth watching, trust me. But fractional money does exist outside of movies, bad movies in Hollywood. So that's the information we want to keep track of for a song and then we want to think about what are some of the things that we want to be able to do in relations to those songs. The other thing we also want to think about is our friend the unique identifier. Is there some unique identifier for a song? And this is one of those things you really need to think about the application that you're using, what assumptions you can make. We might like to say that the name is a unique identifier for a song, but unfortunately, there are many songs that have the same name. Okay.

But I would venture to guess that the combination of the name and the band would perhaps be a unique identifier for a song. The only thing is we don't have one string that we keep track of that keeps name and band in it. So that's another thing that we need to think about, and we'll get into code when we get into code, that we need to think about the design of that particular object. The other thing we need to think about is what changes in an object during its lifetime and what doesn't change. Like, so if I have a song its name and the band that made it for a particular song, like, some band can go uncover the song they learn, but that's a different song, the name and the band name don't change for a song. But hey, it can go on sale and you know, I can jack up the price at the holidays and all that kind of stuff. So the price is something that's malleable.

So another thing you think about in terms of the principles of design is, of the data that I have associated with a particular object, what's going to remain static when that object's created and what's going to be potentially changed? And that's what gives you some insight about what's some of the data, for example, that you only get from an object, what's some data that you can potentially set in the object, and if you think about what potentially uniquely identifies that object, what data do you actually need at the time that you construct the object, right. To say this object is actually some particular unique thing that I care about. Okay. So let's turn that into a little bit of code just to make it a little more concrete. So we'll get rid of our friend, Power Point, and we fire up our friend, Blitz. Ah, and look, a song, how convenient. So here's the information to keep track of a song. It's just a class called song. And what we want to do is keep track of song, the song's name, the band name, and the price.

So when we create the song, one of the things we might do is say, hey, give me all that information to start with. Because if you're going to put some song in your store and you're going to sell it, it better have some song name and band name that I can use to refer to it by, because that's going to be its unique identifier and give me some initial starting price. Now, we might necessarily not require an initial starting price, because it's something that's going to change during the duration of the program, and isn't in support of our unique identifier. But in this case, we're just going to ask for an initial price. The thing we do care about, in terms of the malleability of what's actually in this data structure, is thinking about song name, band name, and price. So song name, we only have a getter for there. There's no setter. Once the object's created, you can't change the band name for that song. You can't say, oh yeah, you know, that was "In Your Eyes," by Peter Gabriel and now it's going to be like, "In Your Eyes," by Kanye West. Like, that's a different song. And I don't know if that's happened. It's probably not a good idea. But the song remains the same, if you're a Led Zeppelin fan, right? And the band name, actually, the band name is also going to remain the same for that particular song. But the price has both a getter and a setter. Right? Because it's something that's malleable. After we create that song, yeah, we might change its price. And because we know that we provide both of those things in the definition of the class.

Now, as we talked about, in days of yore, whenever you create any class it should also have a method called Two String. And Two String just returns a string representation of the data in that class. So this just prints out inside double quotes, which is why we have this backslash, quote, that's a single double quote character, the title of the song in double quotes by the band name, and then it says cost, and it has the price associater with the cost. So it just returns a string to baseline caps lets the data. And here's the private instance variables of that particular class. Right. There's a title, a band, and a price for the title of the song, the band that made the song, and the price of the song. And that's all the information that's in there. But it captures and encapsulates the notion of having a song and what parts of the song are static or can't change, and what parts of the song are mutable or can change. Okay. So besides songs, we also have this thing called albums. Any question about the song portion? If you're sort of feeling good with song, nod your head. All right, good times. If you're not feeling good song, shake your head. If you're awake nod your head. There's a few that's not nodding, but that's okay. That's cool, too. So let's do the class for an album. So the class for an album is another thing we care about. And albums become a little more interesting because an album not only has a name, right, so this is going to be a name, and yeah, the name will probably be some string. And there's also a band, potentially, that produces the album. Now, the interesting thing is the band – you might say, but Marilyn, isn't that redundant? Like, don't I have some album and it’s going to have a bunch of songs on it, and so I already have names for the band for those songs? So why do I need the name of the band for the album?

Anyone know? Want to venture a guess? Anyone have an album that’s like this, ‘80s compilation is the critical word? Right. You can have an album that’s band isn’t actually a real band name. Its band name could just be something like compilation. And it’s going to have a bunch of songs on it, each of one which has a distinct band. Okay. So that’s perfectly fine. There’s no reason why an album, especially in the online world when you can sort of create mixes all the time, needs to have a single band. And so there wouldn’t be a need for having bands associated with songs. We still need to have bands associated with the songs. And potentially, at a higher level, we might want to be able to say, is this whole album by one band, or one artist, or is it actually a compilation. Okay? Now, the interesting part though, is that an album not only has a band and name, but it has a list of songs. So how might we keep track of that list of songs? What would be a reasonable data structure we could use?

Student:

[Inaudible].

Instructor (Mehran Sahami): An Array, our friend an Array. Well, the only problem with an Array is, right, it needs to have some fixed size. There’s some albums out there that are very short, like “In A Gadda Da Vida,” Iron Butterfly, there’s one song that’s one side of the album, if you were back on the LP days, and what a fine album it is. And there’s other albums that are just like, oh, look there’s like 300 songs on here. Okay.

So an Array with just a fixed size might potentially waste a lot of space. What’s the more malleable version we could use?

Student: [Inaudible].

Instructor (Mehran Sahami): Oh, yeah. I love it when it’s just all around. All right.

Student: [Inaudible].

Instructor (Mehran Sahami): [Inaudible] one, I think. Like that post Thanksgiving. It’s like the tryptophan, still like working its way. Yeah. You know.

– albums, to begin with. How do we actually add some list of songs on it. We need to have a way to be able to add songs to this album, and once we actually add songs to a list of songs on the album, we need to have some way of being able to list the [inaudible], or perhaps, iterating over them. The only thing with an ArrayList is enter implements collection interface so that it actually provides you enter it. Okay. So let’s look at the code for that, just real quickly and then things will become more interesting, afterwards. Okay.

So here's an album. Inside an album we have an album name and a band, those are the things that are going to start off by constructing an album. So we say here's the initial album name and band, and what I want to do is build up the contents of that album. So it lets you get the album name and get the band name but you can't set them. Those things are fixed. Okay.

The other thing that I'm actually going to assume here, which is something I didn't assume for songs, is that the name of the album is a unique identifier for the album. Because if I can potentially have compilation albums that's a compilation of multiple bands, so the band name is just something like compilation or maybe the band name is empty string, the album name by itself should be a unique identifier.

Now, you might say, but Marilyn, that's not true in the real world. I have multiple albums that have the same title on them. We're just going to assume that for the purposes of what we're doing here, and it'll be okay.

How do we build up the album? We have a notion of adding a song to an album and getting an iterator over the songs on the album. And so the way we do that is we're going to have something called songs. Let me show you songs down here. Songs is just an ArrayList of songs. Okay. And so if I want to add a song to the album, I pass it in an actual song object and it adds it to its ArrayList. And if I want to list out all these songs that are on the album, I ask for an iterator over all the songs on the album. So what I actually get is an iterator over song objects. Okay.

Two stings just returns the title and the band, it doesn't actually list out all the songs. It just says, hey, it's just this name of this title and this band, and that's all that's in an album. Okay. Again, we think about what's mutable and what's not mutable.

Now, to put the whole store together, this is where things get a little more interesting. To put the whole store together, you need to think about what's the store going to do. So let me show you a simple store running and this is the basic text interface for a store. It's kind of like online store circuit of 1995. Okay. So I can list out all the songs, I can list out all the albums, I can add a song, I can add an album. When the store starts, I have not songs or albums in the store. I need to add them all. I can list all the songs on a particular album and I can also update the price for a song. Okay.

So if I list out all the songs. It says all songs carried by the store and says nothing, because there's no songs that the store currently has. And list out all the albums carried by the store and list out nothing here, because there're no albums. But I can go ahead and do something like add a song. And let's say the song I want to add is "In Your Eyes," Peter Gabriel. Any Peter Gabriel fans out there? No? A little bit? Come on. Oh, man. I give up. It's all over. I just don't believe it. All right. We'll say the song is, I say, okay, it'll be 99 cents. Go get it. All right.

So we add a song and if we list all the songs, now we have here's the string representation of a song, "In Your Eyes," by Peter Gabriel, cost 99 cents. We still have no albums, rights, we just have a particular song that we can potentially sell by itself and we don't have any albums. So we'll come back to this. But this is the basic idea. We want to be able to list all the songs and albums, add songs, add albums, and then list the information for a particular album. Okay.

So if we think about that, what we need is a bigger data structure to keep track of all this information about multiple songs and multiple albums. Okay.

Now, if we want to manage an inventor, the two things we have to keep in mind are also what I mentioned before. A song can exist in our data that is not on any particular album. So as a result it's not sufficient to just say what albums are carried by the store, because some songs may not be on any album, but we still sell them individually. So we need to have some notion of keeping track of a list of songs.

Now there's different things we could think of for a data structure to keep track of songs. One thing is an ArrayList, right. That's what we're using in albums to keep track of a whole list of songs. Another thing we could consider is a HashMap of songs. And so if we think about a map versus an ArrayList, what question that you want to think about gets back to this identifier question, right. Because if you want to have a map, say for example, some string to song, and you want this string to uniquely identify a song, this string needs to be something that is a unique identifier. But a song doesn't have one string that's a unique identifier, it's unique identifier's a combination of a name and a band.

And so all kinds of funky things that are things that people consider. Oh, how can I connect those two strings together? People actually do that in real applications. We're not going to do that here. We're just gonna say, there's too much complexity in dealing with this, we're going to go for a much simpler approach and just say we're going to have an ArrayList of all of our songs and not worry about the unique identifier issue. So here we have an ArrayList of type song, and we'll just call this songs, that's all the songs in our database. And so here we create a new ArrayList of song and we call it constructor. Okay. Now, life in the album world's a little bit different. Besides just keeping track of a list of songs, we also need to keep track of albums. But in the album world the name is actually a unique identifier. And if we want to be able to look up albums quickly, it might make sense to use a HashMap. So part of doing this whole example is to actually show you both ArrayList and HashMap in one application.

So what we could do is have a HashMap that maps from stings to albums where the map, this string, is in some sense the name of the album and this is the actual album object. And we'll call this albums and we can do all the new, you know, la de da HashMap we actually created. Okay. So now we have these two big data structures that actually keep track of stuff for us.

Now, here's where things get a little bit funky. And when things get funky, what you're going to need when you deal with big data structures, you need a guide. And you'll see this in just a second because you're going to see some of the code that we write gets very long when we deal with big data structures. So I'll be your guide. All right. So in the days of yore, I almost bought the whole outfit. But it's a little hot in here, under the lights. So in order to actually think about how you get the information and store the information when you have a large data structure, paper and pencil is your friend. Right If you spend all your time just staring at a computer screen it doesn't really allow you to internalize what is your data structure really look like and what's going on. So break out some pencil and paper, not right now but when you're working on data structures, and draw out, potentially, what things look like.

So here's songs and songs, and songs is an ArrayList. And it's going to have multiple, let's say at this point, three songs in it. And over here we have albums and albums is a HashMap, albums, that maps from names of an album to a particular album object. Now, the important thing to keep in mind in objects, and this is kind of the whole key to big data structure, is all objects, when you refer to them in Java, are references to objects. Remember when we talked about that. When you pass an object to a particular method in some application, you're passing a reference to the object. You're passing where that object lives. Okay.

Which means that when you have an ArrayList of songs, which what you really have here are a bunch of references, which we can think of as pointers that refer to the actual objects that contain the songs. Okay. So over here there's a "In Your Eyes," by Peter Gabriel and it was 99 cents. And over here we might have say, "Ramble On," tell me there's some Zeppelin fans out there. All right, good, good. We will not have to end lecture early. And "Ramble On" is such a great song, it's like $12.99 by itself, single son. That's probably why most people don't listen to it. And over here we have the master, "Stairway to Heaven," Stairway to H, we'll just abbreviate it. Because it's that good, we'll just have a moment of silence, also, by Led Zeppelin, and we'll just say that one should be like, 49 cents so everyone can listen to it. It's just kind of like the bonus tune. All right. And so that's what we have in a list of songs. Now, here's the interesting part, right. If I'm going to have some albums, so I add some albums. So let's say add some album on here like "Soul," by Peter Gabriel, and Soul actually has the song, "In Your Eyes" on it. Okay.

Now there's two things that come up we will need to think about when we actually do this. We need to say, hey, this has got some ArrayList associate in there, and so I can create a new object that is a song for "In Your Eyes" and set my ArrayList to be a reference to that object. And that's a reasonable thing to do in some cases. The only problem is what happens if I go into my store and say, hey, I want to change my song "In Your Eyes" from being 99 cents, because no one's heard of it before, to 9 cents. Okay. So if I go thought my list of songs I say, oh, here it is, I'll change it's cost to be 9 cents. Now, unless I go through all of my albums and find for every album go though every song that's listed on the album and see if I can find that same song duplicated, I'm going to create an inconsistency in my data. What I really want to have is say, hey, there's only one object that is that song. And if that song happens to be a song that's sold individually, or it's a song that's both in my list of songs and on some albums, there's only one object ever that I refer to for that song, which means, I never create the second object out here for that same song. What I do, is when I'm creating the album "Soul," and someone tells me, oh, it's got the song, "In Your Eyes," on it, I say, hey does that already exist in my store. If it does exist in my store, I'm going to add that object to my ArrayList. I'm not going to create a new object, which means each song only ever gets created once, but it can potentially get added to multiple ArrayLists. And it's the same single underlying object that has multiple references to it.

Why is that cool? That's cool because now, when I come along and a whole bunch of people start listening to "In Your Eyes," and I'm like, Peter Gabriel, he just deserves a lot more money, we're going to make this $9.99. It's $9.99 everywhere by changing it once. And that's the real key to large-scale software engineering. You think about not only reusing – you remember for a long time we talked about having methods that you reuse and how you generalize your methods, this is about reusing your data. Thinking about your data, sort of, if it's only one thing, exists in one place ,and everything refers to it. Okay. So any questions about that idea? This is what we refer to as a shallow copy, because what you're getting, after you've created that song once, when you want to add that song somewhere else, you're just setting a reference to it, you're creating a shallow copy, there's only one copy. The thing we did before, where we actually created a whole separate structure, is referred to as a deep copy. And sometimes, deep copies make sense in some particular cases. Most of the time they actually, well, I won't say mot of the time, they don't, it depends on the application, but most of the time what you'll actually be using is your friend, the shallow copy. Okay.

So what does that actually look like if we try to turn that into some code? Well, what does that mean in the application? Let me show you what that means in the application. So we're going to add some songs. We're going to go through another example. All right, let me add the song and I'll just abbreviate, "In Your Eyes," Peter Gabriel, $1.99. Then I'm going to add "Ramble On," oops, "Ramble On," Led Zeppelin, and we'll make that, oh, I don't know, $2.99. Okay. Now, at this point I have two songs. Now, I'm going to add an album. So I add a particular album and the album I'm going to add is "Soul," by Peter Gabriel and it says enter a song name. It's going to have "In Your Eyes" on it. And it asks me because the unique identifier is both the song and the band name, it still needs to ask me for the band name, and the band name I give it is Peter Gabriel. And it says, hey, that song is already in the store. It's just letting you know, hey, I found that song in my store, so when I add it to the album, I'm adding that same object that's also in my store to the album. And then you could say, well, there's other stuff on there like there happens to be a tune called, "Red Rain," which is also by Peter Gabriel, and you know it's a fine tune, but let's just say it's 1 cent, okay. And it says new song to add to the store. What did it do here? What it did in this case, it says, hey, you want to have a new song called "Red Rain," by Peter Gabriel. That song costs 1 cent, you want to add it to your album.

Well, if you want to add it to your album, it's also a song that I'm going to see in the store. So it actually adds it to the store and adds it to the album. And there's still only one copy of that object ever. It just needs to make sure that when it creates a new song to add to an album that's not already in the store, it adds it to the store, as well as to the album.

If the song already exists in the store then it just adds a reference to the album. Okay. That’s the critical idea here. All right. So now, if we sort of list – I’ll hit enter quit – and if we list all the songs, right, the song “Red Rain” has now been added to the store and costs 1 cent. And if I list all the albums that are sold by Peter Gabriel, and if I list all the songs on that album, it has the songs “In Your Eyes” and “Red Rain” so it matches the picture that I think.

That’s why having a piece of paper, where you draw pictures, is useful. Because you look at what you’re application is doing and you say, does it match what I actually think should be happening in my picture. And if doesn’t, then you know one of two things is wrong. Either your picture’s wrong or your code that’s supposed to be dealing with that picture is wrong. But in either case, you’ve already figured out a bug, even though the program hasn’t crashed or anything, you just know there’s an inconsistency. Okay. And so now, if I update the price for a song, like I update the song, “In Your Eyes,” by Peter Gabriel, and I change it’s price to, I just go crazy, no one’s going to buy the song anymore, the price is updated. Now, if I list all the songs, that song is $999.99 in the store, and if I also list the songs on any album – five, so lower case, the price is also updated on each of the individual albums, because there’s only one object. Okay. That’s where the consistency comes in. That’s why the consistency’s key. Okay.

So what does this actually look like in code? How do we do this? Let me show you what the actual application looks like for our little friend, the Flytune Store. Okay. So there’s a bunch of stuff at the beginning that just asks for the user selection, basically print some stuff out to allow you to make a selection, and then gets you’re selection for you. And then there’s a big case statement that calls an appropriate method, depending what selection you made. So I’ll go though some of the simple ones pretty quickly. You can list out all the songs carried in the store. In order to be able to do that, we need to keep track of how this information’s actually stored, it’s exactly in these data structures I just showed you. Song is kept track of in an ArrayList of songs and albums is kept track of in a HashMap that maps from the name of the album to the actual album data structure, itself. Okay. Any questions about that, hopefully, that’s all clear. I will take off the hat.

So how do we print these things out? To list all the songs, we just go through our ArrayList up to its size, and this is why you want to think of data structure as your needed guide, because you’re going a journey. At any given point, when you’re dealing with a data structure, you want to think, what is the type I’ll dealing with right now? What does that mean? It means, when I want to print something out, what I need is a string that prints out. How do I get a string? If I started at songs, songs is an ArrayList. I don’t have a string I can print out. But from an ArrayList I can get an individual element. When I get an individual element of that ArrayList, what do I have? I still don’t have a string I have a song. What can I ask the song for? I can ask to get the string version of the song and I have a string to print out. Okay.

So you always want to think of it as you’re going on a journey. Where do you start your journey? You’re journey starts at the data structures you have available to you. In this case, we have a data structure called songs, another data structure called albums, that’s what's available to us. And what we want to do is go from that starting point through a series of steps to get to the thing that we actually care about at the end, hat little piece of data that we want to display or interact with somehow. So here's another example. If I want to list all the albums, how do I list all the albums? Well, to list all the albums, albums is a HashSet. So in order to do something with a HashSet I need to say, hey, I want an iterator over all the keys of that HashSet. So albums is the HashSet, I get the keys of the HashSet, which is a collection, and I get an iterator for that collection, which is an iterator over all the keys of the HashSet. And now, as long as my album iterator, which is just my iterator over the keys, has an element, what do I do? I start at albums. I need say I need to get a particular album. Okay. Get. Which album am I going to get? I'm going to get the album whose name is associated with the next elements of the iterator. Right, because it's an iterator over all the names of albums. So get, gives me a particular album. Then, when I have the particular album, I can call two strings on it to get the string form of the album. Okay, any questions about that? Because they're going to get even longer, so if there are any questions about sort of the chain of things we call.

If it's making sense, the chain of things we call, nod your head. All right, and if it's not making sense, shake your head. And if it's kind of making sense, just keep looking and ask a question if a question comes to mind. All right. So how do I find a particular song? This is something where I'm going to use the helper method, so it's private to find a particular song. Songs, our unique identifier, is a combination of both the band name or the name of the song and the band name. So how do I check for that? I'm going to go through all my songs, it's an ArrayList so I can count through all the songs. Here's where things get long. How do I check to see if a song, that's actually in my data set, matches on its name with the name that's passed in? I start at songs, get the I song, and I have one particular song. For that particular object I get the song and name. Now, I have a string. I want to check to see if that string is equals to the name that's passed in. Okay.

And I do the same thing with band names. Song, get the I song, get the band name of that song, and then check to see if that's equal to the band. And if both of these are equal, then, hey, I found the song, and so I'm going to return an index, which is the index location of that song in my ArrayList, and I can just break out of the four-loop, here. Because once I find it, I say, hey, I found that, I don't need to keep looking, so actually this is one of the rare cases where you'll see a break in a four-loop, is you don't need to finish the loop. You got to what you were looking for and get out of the loop. If you manage to get through this whole loop without ever finding something that matches on both, the name and the band, well, your index remains negative one. So you return negative one to indicate, hey, I didn't find it, because you know negative one's not a valid index for an ArrayList. So if you return it that means you didn't find a valid element. Okay. How do we use find song? Here's how add song works. Okay. When you want to thing about add song, you want to think about this property that we're only ever going to create an object once, and everything else is going to be references to that object. So the way add song is going to work is it's going to return a song object. Okay. And what it's going to do, is it's going to ask us for the name of a song, if the user enters blank line that means they want to stop adding songs so it just returns null to say, hey, you want to stop adding songs, I didn't create a new song, here's a null to indicate you are done. But if they don’t impress enter quick, I also ask for a band name, and then I ask to find the song. Okay.

I call that find song method I just wrote and I say, does that song exist. If the song exists, the song index is not going to be minus one. And that means, that song already exists in the store. So you told me to add a song that already existed in the store. So I’m not going to create a new song because it’s already an object in the store that encapsulates all the information for that song, I will return to you a reference to that object, which means I just returned from the songs ArrayList whatever song happens to be at the index that that song actually lives at. Okay. So this just returns an actual object. It actually returns a reference. If you can, think of it as returning a pointer to the object. If I didn’t find it in there, then, hey, I need to create the new song, right. It’s sort of like “Red Rain” at the end. You wanted to add a song. It didn’t exist in the store, let me get the price for that song. I’ll create a new song object and now. Here’s the funky thing, I will add that song to my ArrayList of songs for the whole store, write out to that the new song was added to the store, and I’ll return that new song to you so you can do whatever you want with it.

And so now, you might ask, okay, Marilyn, if I just added a song to the store I don’t really care about doing anything with that song, why are you returning the song to me? And that’s true. If I just add a song to the store, if that’s all I care about, I ignore the return value. That’s actually what I do up here, which is very funky. Right. If you want to add a song, I just call the add song method, it goes ahead and adds the song to the store, if it doesn’t already exist, and it returns reference that song object. If all I’m doing is adding a song, I don’t care I just ignore it. I don’t assign it to anything, I just say, yeah, thanks for returning that object, that was fun, whatever, and just get rid of it. Okay. But the reason why I’ve written it this was is if I’m adding an album, what do I do? I ask for the name of the album, and I check to see if that album’s already in the store. If the album’s already in the store I’m not going to do anything because the album’s already in the store. If the album’s not already in the store, then I ask for the band name and I create a new album. And then I put that album in the store. So album is my HashMap. I put in that HashMap the name of the album is going to be the key and the actual album object is the object. So I add, you know, the album “Soul” to my HashMap.

Now, I’m going to add all the songs. So I have a Y-loop that goes through and keeps adding songs until I get a null from add song to indicate that the user wanted to stop adding songs. But here’s the funky part, every time the user adds a song, right, it comes along and says, hey, you want to create some new album? So let’s say I actually want to create some new album over here when I create the album “Soul,” so none of this stuff exists yet. Okay. So to create a new album, I say, hey, I want to create the album “Soul.” It says, okay, that’s fine, create an object for the album “Soul.” It has the name “Soul,” it’s by Peter Gabriel. And it says, okay, what songs are on going to be in there? And it starts asking me for songs, because it’s going to add them to my ArrayList in here. And so the first song I say is “In Your Eyes” in on that album. It goes and says, hey, find that song, it already exists. It returns a reference to that song, as a pointer that reference is what gets added to my ArrayList. Now, I go and ask for another song. Do you have any more songs? I say, yeah, there’s another song. The song is called “Red Rain.” When I go to create “Red Rain” it comes up here to add song, add song comes along, asks for the name and the band, it tries to find the song and says, hey, that song isn’t already there, so I’m going to create a new song. It creates a new song called “Red Rain,” by Peter Gabriel, has some price associate with it, and adds it to the list of songs for the store. And then it returns this object, which means it returns a reference to this object, and that reference to the object – oops, sorry this got blocked – all right, this is where it is creating a new song, and it adds the song to the store.

Right, it adds it as a song, which is that ArrayList up there, and then it returns the object. So when it returns the object, I went too far, the add object does not know I add that song to the album. So this album, we’re going to add a song, and the song we’re going to add is that same object. It’s “Red Rain.” Okay. So that’s the important thing to keep in mind. That object we only created once, and we passed around references to it or we can return references to it, and assign them other places. And that’s how you get consistency in a much bigger date structure. Now, there are a bunch of other things we could do in here. I won’t go through all the excruciating details down here. But we can list the songs on an album, we can update the song’s price. And by updating a song’s price, all we do is we ask for the song and the band, we find the song in the data set if it existed. If it doesn’t exist we just say, hey, it’s not in the store, and if it does exist then we read in it’s price, and then for the songs in the store, we find the song at that index and set its price. And we know that whatever other albums contain that particular song, if we happen to update the price over here to you know, $6.99, we only update it once and all the places that refer to it automatically will see the updated version, because they point to the same object. Okay.

Any questions about that? So I know it’s a lot of complexity to kind of deal with a big data structure like this. But now it’s one of those things like, now you’re old enough to kind of see the big honking data structure. Because in the real world, when people think of software engineering the large, these are the kinds of things they need to worry about and that’s where the complexity comes in. It’s keeping track of all your objects and thinking about what objects you actually need to design and build, in order to actually build an application that’s kind of successful to keep track of and makes thing consistent with all the data you have. So any other questions? Uh huh.

Student: [Inaudible].

Instructor (Mehran Sahami): Oh, can you use the mic, please?

Student: Sorry. So in this application, do all the songs and albums, they’re also, I guess, singles, or – cause the albums are never priced, right? It’s just the individual songs

Instructor (Mehran Sahami): Right. So the albums don’t have a price. You could imagine the cost for an album is the total of all the songs on the album. Or you could actually do something funky. Like this is one of those places you can make a policy decision and say, an album is 90 percent of the cost of all the songs on it. And then all the individual song prices can change and any time you just say, what’s the price of the album, total up all the prices of the songs, and take 90 percent of that. So it also allows for very dynamic album pricing.

Student: Thank you.

Instructor (Mehran Sahami): All righty. If there's any more questions, come on up. Otherwise I'll see you on Wednesday.

[End of Audio]

Duration: 49 minutes