CS 240: Java Collections – Using Collections Correctly Transcript

This video shows a split screen of Professor Wilkerson on the right and a PowerPoint screen on the left. The left screen will often switch to a web browser. Any text displayed or action performed that is not verbalized will be included in italics as visual descriptions.

[00:00:00] **KEN RODHAM:** Next, we want to discuss some considerations that you should think about when you're using the collections classes.

[00:00:07] It's easy to use these classes wrong, and I want to talk about some gotchas that frequently trip people up.

[00:00:18] I want to talk about how to use these correctly and how to use them successfully.

[00:00:24] We're going to focus on the equals, the hashCode, and the Comparable interface.

[00:00:29] A lot of these issues revolve around when you choose to use a hashing base structure like HashMap or HashSet, or if you use a tree-based structure, like a TreeMap or a TreeSet.

[00:00:41] Then you need to think hard about your equals method, your hashCode method, and your Comparable interface.

[00:00:51] The first consideration is that when you use these collection classes, that means you're going to insert objects into the collections.

[00:01:02] Whatever these objects are that you're inserting into the collections need to implement the equals method properly for the equality that you want.

[00:01:14] We recently discussed that when you write a class in Java, you need to think about, "What kind of equality do I want for the objects of this class? Do I want the objects to be compared by identity or address such that an object's only equal to itself? Or do I want to compare these objects based on value so that two different objects could in fact be equal if they contain the same data." Whenever you're writing a class, you need to think about that.

[00:01:41] It's especially important if you're going to take those objects and put them in one of these Java collections classes because these Java collections classes call the equals method a lot.

[00:01:53] For example, one of the methods on the collections is contains, so you can query a collection and ask if it contains a particular object or a particular value.

[00:02:03] Well, how does the collection figure out if a particular value is in the collection? Well, it iterates over all the elements and it calls the equals method.

[00:02:12] If your equals method doesn't work the way that you really want it to, then the contains method won't work the right way either.

[00:02:18] Make sure you're always thinking about your equals method and making sure that it has the implementation that you want it to have.

[00:02:26] We did say also that the default implementation of equals in Java that you inherit from the Object class compares objects by address, so you get identity-based equality by default.

[00:02:40] If that's not what you want, you need to override your equals method.

[00:02:47] If you're using one of the hashing-based collections, HashSet or HashMap, not only do you need to override the equals method, but you're also going to need to think about your hashCode method and make sure that it works properly as well.

[00:03:01] Now, as we've discussed, your equals method and your hashCode methods need to be in sync with each other.

[00:03:07] They both need to work the same way, they both need to implement the same kind of equality, meaning that if you're using address-based equality, then you would also want to use address-based hashing, and that's what you get by default in Java.

[00:03:24] The default hashCode method that you inherit simply returns the address of the object, which is perfect.

[00:03:31] If that's the equality that you're using, then you really don't need to override hashCode or equals, but if you do want to do value-based quality, you'd need to override your hashCode method to implement that.

[00:03:44] Whatever variables are being compared for equality, those would be the same variables that you would want to use in your hashCode calculations, and so just make sure that you're thinking about this.

[00:03:57] Anytime you use a hash-based data structure, the class needs to implement the equality that you want and hashing that you want.

[00:04:10] Now the good news is a lot of times, when we use data structures, we use built-in Java datatypes that already implement hashCode and equals the right way, like String, or the Date class, or in a lot of times the objects that we're putting into our collections are provided by Java.

[00:04:27] They already implement the equals and hash code, you don't need to worry about it.

[00:04:31] For example, if you're using a set of strings, or a map where the the key type is string, you're not going to need to worry about this because they've already done it for you.

[00:04:40] It’s really when you writing your own custom classes that you need to think about how to do this.

[00:04:49] Let’s talk about the Comparable interface for a minute.

[00:04:54] Whenever you use a tree-based data structure, TreeSet or TreeMap or PriorityQueue, it turns out, you need to make sure that the objects that you’re putting into these structures are sortable.

[00:05:10] Meaning, if I’m going to have a set of strings, for example, well, they can’t build a binary search tree from string objects unless the string objects can be compared to see which one is less than the other one.

[00:05:24] That’s just inherent in a binary search trees: you have to be able to compare the objects to see who’s greater than or who’s less than the other one.

[00:05:32] This is certainly something that TreeSet and TreeMaps do because they use binary search trees.

[00:05:38] A PriorityQueue doesn’t use a binary search tree, but it does use a data structure underneath that requires that objects be comparable so that they can be sorted properly.

[00:05:48] You can imagine that we can’t return the highest priority element in the queue if we can’t compare the elements in any way.

[00:05:57] In Java, there’s an interface that you should implement on a class if you want objects of that class to be sortable, or to have order, if you will.

[00:06:12] The way you do that is you just implement this interface called Comparable.

[00:06:17] Now the Comparable interface in Java—maybe we should go look at it. *(Rodham opens the Java documentation website in a browser window.)*

[00:06:21] If you look at the Comparable interface, it’s only got one method on it; it’s called compareTo.

[00:06:31] It’s similar to the equals method.

[00:06:34] You can implement the compareTo method on a class and the parameter that is passed in is the object that you’re being compared to, so like equals.

[00:06:46] Then the return value is an integer.

[00:06:49] In this case, if I want my class to be sortable, I’ll implement the Comparable interface, I’ll put a compareTo method on it.

[00:06:58] The compareTo method, when it’s called, it’s going to compare itself with the object that was passed in, and it’s going to return an integer that indicates the relationship between those two objects.

[00:07:11] In specific, if the two objects are equal to each other, compareTo returns a 0.

[00:07:21] If the current object is greater than the parameter object, it returns an integer greater than 0.

[00:07:30] If the current object is less than the parameter object, it returns an integer less than 0.

[00:07:36] This method either returns 0, positive, or negative, depending on the relationship between the two objects that are being compared.

[00:07:44] This is an idiom that you’ll see in just about every language that you’ll use is that when they implement sorting of objects or values, this is how they do it.

[00:07:56] You have to write some comparison method that returns 0, positive, or negative, depending on the relationship there.

[00:08:05] The big idea to take away is that if you're going to use a TreeSet, the objects that you're putting into the set have to implement the Comparable interface.

[00:08:16] If you're using a TreeMap, the class that you're using as the key type has to implement the comparable interface.

[00:08:23] If you're using a priority queue, the elements or the objects that you're putting into the queue have to implement the Comparable interface.

[00:08:29] Now there's also a notion of a Comparator in Java, which is useful when let's say I have some objects that I want to put into a TreeSet and they don't implement Comparable.

[00:08:42] Maybe it's a class that I didn't write, so I can't implement Comparable on it myself because I didn't write it.

[00:08:48] There's also this notion of a Comparator, which is another way to provide a comparison algorithm that the TreeSet can use.

[00:08:57] Let's do an example or look at an example of the Comparable interface, just so you can have a more concrete idea of what I'm talking about.

[00:09:08] What I've done in this slide is I've implemented a class called TimeOfDay.

[00:09:13] TimeOfDay object just represents the time of day.

[00:09:16] It's got an hour component, it's got a minute component.

[00:09:23] You'll see here that it does implement the comparable interface.

[00:09:29] Now the comparable interface is a generic interface, so it can be specialized for different data types.

[00:09:35] What this means is I'm implementing the comparable interface for TimeOfDay objects.

[00:09:40] It means I know how to compare myself with a TimeOfDay object.

[00:09:45] You'll notice down here, when we implement the compareTo method, that the parameter is actually a TimeOfDay object; it's not a generic object.

[00:09:53] This is a little bit different than the equals method for equals.

[00:09:56] The parameter is a generic object, but in this case, you can specify what parameter type you want for your compareTo method.

[00:10:05] In this case, I've got an hour and a minute and I've got getters and setters for both of those. We default to zeros.

[00:10:17] That would be like midnight.

[00:10:20] But let's suppose I want my TimeOfDay objects to be comparable.

[00:10:24] This would allow me to create a TreeSet of TimeOfDay, for example, or I could use TimeOfDay as a key type in a TreeMap.

[00:10:34] What I'm going to do in the compareTo method is I need to be able to compare two TimeOfDay objects and see which one is less than the other.

[00:10:42] Now as the creator of this class, I get to define what it means for one object to be less than or greater than the other, but in this case, what we're going to do is we're going to base the comparison first of all on the hour components.

[00:10:59] If one time of day has an hour value that's less than the other one, then the one that has the less hour would be lesser.

[00:11:08] Now if the hours are equal in both of the objects, then we're going to compare the minutes.

[00:11:13] In that case, the object with the lesser minute value would be lesser than the other object.

[00:11:20] This is pretty typical that when you have a class and you want to implement the compareTo method, you have to define how am I going to compare these objects.

[00:11:30] If you have multiple variables that are being compared, then you have to decide which one's going to be compared first, which one's going to be compared second.

[00:11:38] That's what this compareTo method down here does.

[00:11:41] What it does, first of all, is it just takes the two hour values from the two objects and it compares them.

[00:11:49] Now to do that, it calls this compare method on the integer wrapper class.

[00:11:54] Conveniently, the integer class has a compare method on it, which will give me back a 0, a positive, or a negative value, based on the relationship of those two values.

[00:12:04] I just call the integer compare method, compare the two hour values, and it gives me back an int.

[00:12:11] Now if that result is not equal to 0, then I can just return it directly.

[00:12:17] I already know the answer.

[00:12:18] But if that first comparison comes back 0, then I need to compare the minute values with each other.

[00:12:23] Then the comparison of the minute values would be the answer at that point.

[00:12:29] That's a very typical implementation of a compareTo method.

[00:12:34] Anytime you want to do any sorting in Java, you got to implement the compareTo.

[00:12:39] The Java library also has sorting algorithms built into it.

[00:12:46] If you want to sort objects using their built-in sort, you're going to have to implement Comparable for that as well.

[00:12:52] That's actually one thing that I should show you real quick, is Java does have built-in algorithms. *(Rodham opens the Java documentation website in a browser window.)*

[00:13:01] If I go back to the java.util package, in the documentation, there's a couple of classes here that you should be aware of.

[00:13:14] There's a class called Arrays. That's plural.

[00:13:21] This is a class that has a bunch of static methods on it that implement algorithms that are useful for arrays.

[00:13:29] For example, it has a binary search.

[00:13:31] You've learned about binary search before and Java has a built-in implementation of it.

[00:13:36] You can convert an array to a list, which is something you often do in programs; they have an asList method.

[00:13:45] Binary search. What else? They've got algorithms for comparing the elements of an array.

[00:13:55] If you want to do a comparison on an array level, you can do that.

[00:14:00] You can make copies of arrays.

[00:14:01] You have the sorting algorithm down here as well.

[00:14:06] If you need to sort an array, you can do that. There, sort.

[00:14:15] Now the other class that's in the collections library to be familiar with is there is a class named Collections, plural.

[00:14:25] This is a class that has algorithms for the built-in collections classes.

[00:14:33] The Arrays class had algorithms for arrays, the Collections class has algorithms for the Collections classes.

[00:14:38] It's similar to the Arrays class. It's got binary search, it's got sorting, it's got randomization, it's got max and min and shuffle and all kinds of stuff.

[00:14:49] Just be aware that those algorithms are available; you don't need to write those yourself.

[00:14:58] Now the last thing I want to talk about is that when you're using the Collections classes, you have to be really careful.

[00:15:11] What I mean by that is, let's suppose you create a HashSet and you insert a bunch of objects into the HashSet.

[00:15:23] Well, let's suppose that you want to actually modify one of those objects while it's in the HashSet.

[00:15:29] Let's say you go to one of the objects that's in the HashSet and you change its variables.

[00:15:35] Well, that's a problem actually, because if the changes you made to the object would actually give it a different hash code, that means that that object is now in the wrong spot in the HashSet.

[00:15:49] If you modify an object that's in a HashSet in a way that changes its hashCode, that means you've ruined your hash table because that object's now in the wrong place.

[00:15:59] It means you'll never be able to find that object again because it's got the wrong hashCode.

[00:16:06] Be very careful that if you're going to modify objects that are in a collection.

[00:16:12] Ask yourself the question: "If I modify the object while it's in the collection, am I going to mess up the collection?" Hash tables are a perfect example.

[00:16:20] If you're going to modify an object in a hash table in a way that changes its hashCode, you can't do that.

[00:16:27] What you would have to do instead is you'd have to take the object out of the HashSet, modify it, and then put it back in.

[00:16:34] That would make sure that it's always in the right place in the hash table.

[00:16:38] The same goes for tree sets.

[00:16:42] If you're going to insert a bunch of objects into a TreeSet, you're not allowed to modify the objects in a way that would change their position in the tree while they're in the TreeSet.

[00:16:54] You'd have to take the object out of the TreeSet, modify it, and then put it back in.

[00:16:59] Otherwise, that object is going to get lost.

[00:17:01] It's going to be in the wrong place in the tree.

[00:17:04] Those bugs can be really hard to find.

[00:17:09] I suggest you think very carefully whenever you modify an object that's inside a data structure of some kind, make sure that you're not going to corrupt the data structure by the changes that you're making.

[00:17:23] I think the last thing to talk about with collections is iteration.

[00:17:33] We talked about how collections support iterators.

[00:17:36] Of course, you can iterate over any collection.

[00:17:40] One of the gotchas that a lot of people run up against though is if you're iterating over a collection, you're in a loop that uses an iterator.

[00:17:50] If you try to modify the collection while you're iterating over it, that usually causes a problem.

[00:17:56] Sometimes you'll get an exception thrown that...what do they call it? They call it a concurrent modification exception.

[00:18:05] For example, if you're trying to iterate over a binary search tree and you actually modify the tree while you're iterating over it, typically, the iterator is going to complain.

[00:18:17] It's going to throw an exception, say you can't do that.

[00:18:19] You can't modify the collection while you're trying to iterate over it.

[00:18:23] If you want to modify the collection later, that's great, but you got to wait till you're finished iterating over it.

[00:18:29] That goes for any collection potentially.

[00:18:33] It could apply definitely the hash sets, HashMaps, TreeSets, TreeMaps.

[00:18:40] Lot of times, the lists can handle that thing better, but even in those cases, they might throw an exception like that at you.

[00:18:46] The principle is, don't modify a collection while you're iterating over it.

[00:18:50] Any changes you need to make to it, wait and do those later after you're done with the iteration.

[00:18:56] For example, if I'm going to iterate over a collection and I might need to keep a list of all the objects that I want to delete from it later, rather than deleting those objects while I'm iterating, TheI can do all the deletions after the iteration is over.

[00:19:13] That's our basic overview of the collections in Java.

[00:19:19] We're going to use those on the Evil Hangman lab, which we will talk about next.