1. **Working with Collections**

[1. Introduction 1](#_Toc31139)

[2. A First Look at Collections 1](#_Toc2484)

[3. Collections and Type Safety 1](#_Toc6183)

[4. Collection Interface 2](#_Toc29384)

[5. Java 8 Collection Features 2](#_Toc19243)

[6. Converting Between Collections and Arrays 2](#_Toc23200)

[7. Collection Types 2](#_Toc11796)

[8. Sorting 3](#_Toc2865)

[9. Map Collections 3](#_Toc5790)

[10. Sorted Map Collections 3](#_Toc20597)

[11. Summary 3](#_Toc330)

# Introduction

Slides=> pg. 1-2

Welcome to the next module of the Pluralsight course Java Fundamentals. In this module, we're going to talk about working with collections. Now, in this module, we've got a lot of great stuff to cover. We'll first take a look at the role of collections in our applications. We'll talk about the issue of collections and ensuring type safety. Look at some of the most commonly used collection methods. Take a look at how the issue of equality of the entries in collection is handled. We'll take a look at some of the features that were added to collections as part of Java 8. We'll look at how we can convert between collections and arrays. Look at some of the most commonly used collection interfaces and classes. We'll then look at how certain collections deal with sorting behavior. And then finally, we'll wrap up by talking about map collections, also known as dictionaries. Okay, so let's get started.

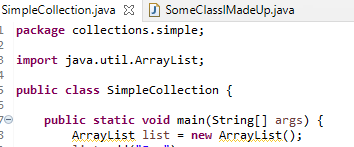
# A First Look at Collections

slides=>pg. 3

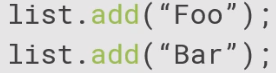
When we build our applications, something that we very commonly have to do is manage groups of data. Imagine if you're building an application to manage passengers on a flight, well you'd have to have a group of passengers for that flight. Or if you're building a social media app, you're going to have to have groups of messages. So we need a way to store these groups of commonly typed data. The most basic way to do that is to use arrays. And arrays work well for simple cases, but arrays do have some limitations. Probably one of the biggest ones we encounter is that arrays are statically sized. So when we create an array, we have to make sure we have enough room for everything we want to store in it. Or if we do run out of room, we end up having to do the work to create another larger array and copy things over. Also, arrays rely on explicit position management. In other words, if you put something into position 0, when you go back to position 0, what's going to be there is whatever you put there. If you want to any kind of higher‑level stuff like sorting or other things, you have to kind of do that more explicitly. So what it comes down to is that arrays are really little more than a bunch of values. Now that doesn't make arrays bad. It just means that they're limited in their usage. And sometimes we need something more powerful than arrays, and that's where collections come in. Collections provide a more powerful option for managing groups of data than is available with arrays.

Slides=>pg. 4

Now, basically, all a collection is is something that can hold and organize values. And so fundamentally, they are conceptually very much like arrays. They're iterable. So you can actually walk through what's in a collection. Collections can provide type safety. So just like you can you have an array of a particular type, you can have a collection of a particular type. But probably one of the most fundamental benefits of collections is that they tend to dynamically size. So you can just keep adding things to them, and they'll get bigger as they need to. Now there are a wide variety of collections available. Now most simply, a collection can be just a list of values. So again, kind of think of it as an array that just gets bigger as you need it to, and that alone is certainly valuable. But collections can provide a lot of optimization or sophistication because there's a lot of different kinds of collections. So some collection types provide ordering. They may automatically sort things for you. Some collection types will prevent duplicates. Some collection types allow you to manage things as name/value pairs. So you can put things in there and then go back and get them back by their name, a concept often known as maps or dictionaries. So collections are kind of this generalized concept with lots of different specific types of collections. Now look at some really simple code that uses a collection.



Let's say we're going to have a simple collection here. It's just a collection of objects. So I'm going to declare a collection called an ArrayList. And as its name sounds, it basically acts like an array. But it is a collection, so it gets bigger. Now notice as we declare this ArrayList, we're not saying anything about the type that holds, which makes this an ArrayList that holds the type object, which is pretty much all class types inside of Java.



So I can do things like just add in the string Foo, add in the string Bar. And notice when I declared the ArrayList, I said nothing about size. So the ArrayList is getting bigger as it needs to.

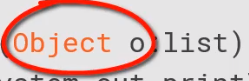


At any point, I can ask the ArrayList how many elements are inside of it by calling the size method. So this would tell me there are two elements inside of there. As I mentioned, it's iterable.

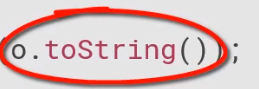


So I can do a for loop over it

and then write them out. But a couple things. Because this ArrayList doesn't have a type associated with it, it's an ArrayList of objects.



So the return type here is Object.



And so if I want to get to the string value inside of it, I'm calling the toString method on there. Now just as a note here. The call to a println would call toString for me automatically. I'm just kind of putting that in there to show it explicitly.

But if I do want to get the value out as a string, I can get to the value by calling the get method on the collection, but notice that would come back as an object. So I have to explicitly cast it to be a string.

Now let's say I go out and I just create an instance of some class I've just made up, some arbitrary class. Because I am restricted to the type of my ArrayList,



I can also just add that right into that same collection. Now there may be times I want to do that, but generally I don't. So if we look at what we've seen in this collection code we have here, we've seen the fact that I can add things to it and it'll grow dynamically. We can iterate over it in. In general, we're probably going to want to restrict the types that we put inside of a given collection. So in the next clip, let's take a look at how we can actually restrict a given collection to a specific type.

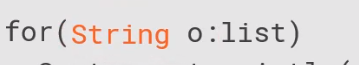
# Collections and Type Safety

Slides=>6

As we saw in the previous clip, when we worked with our collection and we didn't associate with any given type, that collection operated on the Object types. So what that meant was any return values that we got back from the collection were of type Object, which meant we had to actually convert them to our desired type. And the collection didn't restrict what types we could add to the collection. So in general, we want to go ahead and associate our collections with a particular type. We want to restrict the type that they operate on. Collections use the Java concept called generics to do that. And conceptually, generics are kind of simple. It basically means that when we create our collection, we specify what types that collection can operate on. Now generics are a very big and powerful concept. You want to know more about them. Later in the course, I'll give you some pointers to where you can learn more about them. But for our purposes, just understand that generics allow us restrict the types the collection operates on. And that type restriction is pervasive so that when you associate a collection with a particular type, the return values that you get back from that collection will be typed appropriately. They'll be typed based on what you've associated the collection with. And the collection will enforce those types when you add things to it. So the compiler will check to make sure that anything you're adding to collection is compatible with the type associated with the collection. So let's take a look at the code that we wrote earlier, but let's go ahead and type our collection this time.



So we'll have our ArrayList, but notice now after the collection ArayList, we actually have the less than the type String, then the greater than before we give the variable name. Well, this says that this is an ArrayList that holds the type String, and this can be any of our classes. So if we create our own custom class, our arrayList can be types associated with that. So now when we set up the ArrayList and we say its variable name is list, when we new up the ArrayList, we go ahead and we're typing the ArrayList again here. So we're saying that our ArrayList, it holds strings, the variable's called list, and we create an array list to hold strings. Now for convenience, on the right‑hand side of the equals sign, we can take that second occurrence of String and just remove that and just put the less than and greater than. So the compiler will go ahead and set that up appropriately for us. So we're still creating a list that holds strings. So we'll go ahead and add our string values, just as we did before. And we of course can ask the collection how big it is by calling list.size. And we can of course iterate through the collection. Now remember, though, this time, since we actually associated the string type with the collection, here in the for statement, rather than getting objects back each time, we will actually get strings back.



And since they are strings we don't need to call toString on the instance;



we'll just get the string value directly from the variable because it is a string. And when we call the get method to actually get an individual value out of the collection.



Previously, we had to cast the string, but now since our ArrayList is associated with the type string, the get will actually return the values as strings.



And again as we said, that enforces that so that we have some type here, SomeClassIMadeUp and this variable c, if we go ahead and try to add that to the collection, well, now the compiler will say, well, wait a minute, SomeClassIMadeUp is not compatible with type String. Therefore, you can't add that to this collection. So you see we're getting a lot of power now. Our collection is strongly typed. It's limited to holding just things that are strings, and we get this dynamic growth. And as we'll see, there's a lot more that we get with collections. So in the next clip, we'll take a look at the collection interface and those features that are common across collections.

# Collection Interface

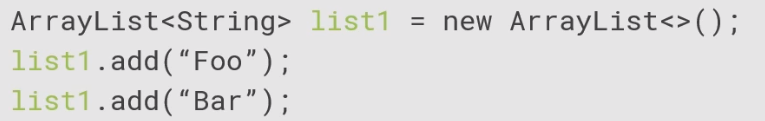
Slides=> pg. 8

Java provides a number of different collection types, and each of those collections have their own features. But many of the features are common across collections. Now these common features mostly come from the collection interface, and most collections implement that interface. Now, a notable exception are the map collections. They don't implement the collection interface. We'll talk more about them a little bit later in this module. Collection interface extends the Iterable interface, which is why we can use things like forEach to walk through collections. But of course the collection interface provides a number of methods that are specific to collections.

Slides=> pg. 9

And if we look at some of the most common ones, things like the size method, which we've already seen in code, tells us the number of elements inside a collection. The clear method will remove all elements in a collection. IsEmpty returns true if the collection has no elements in it. We've seen add before where we can add a member to a collection, and we have addAll, which allows us to add all the members from one collection into another. So let's take a look at using addAll.

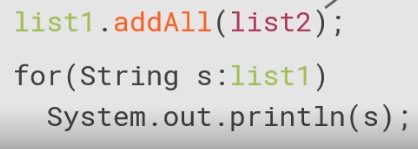
Slides=> pg. 10



So we declare an ArrayList here to hold strings, and we add a couple members to it.



Now let's use another collection type called a LinkedList, which also holds strings, and we'll add some members to that. Now these are two different types of collections, but they both implement the collection interface,

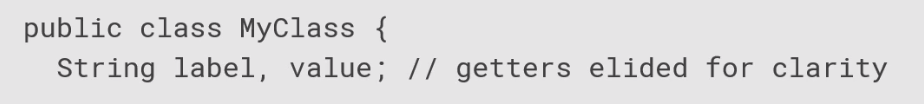


so we can do things like call addAll, so we'll addAll the members from list2 in to list1. Now this doesn't change list2, right, list2 will still have exactly what it had before. But now if we iterate over list1, we'll have the original members that we added directly to list1, as well as the members we added from list2.

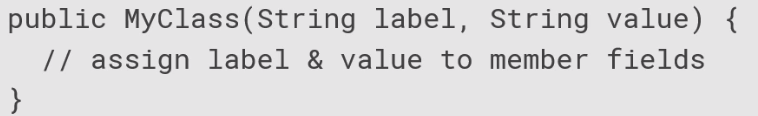
Slides=> pg. 11

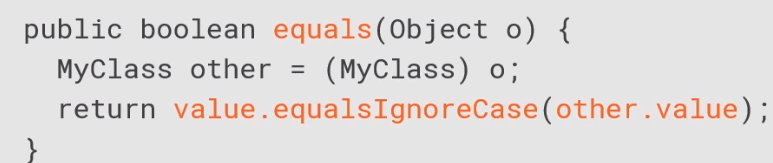
Now many of the methods of the collection interface rely on equality tests, so things like contains will return true if the collection contains a particular element. ContainsAll returns true if the collection contains all the elements in another collection. Remove allows you to remove a single element. RemoveAll removes all the elements from one collection from the current collection. And retainAll removes all the elements from the current collection that are not in another collection. The thing is though, each one of these require some kind of equality test, and that equality test is based on the class's equals method. Right, so it doesn't do a reference equals, \_\_\_\_\_ it's the exact same object. It uses to return value of the equals method. So let's see how that affects our applications.

Slides=> pg. 12

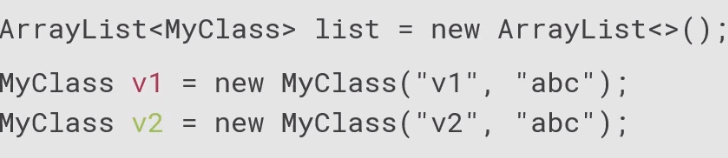


So let's say I have a class here we're going to call MyClass, and it has two member fields, label and value,

it has a constructor that accepts the label and value and puts them into the member fields, and we override the equals method.

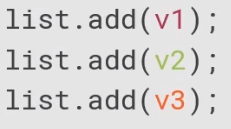
So what we do then is we accept an object, cast it to our type, and then we do an equalsIgnoreCase on the value. So notice that our definition of equals is simply if the value fields match, we don't look at the label fields.

Slides =>pg. 13



And so now we have some code that uses this. We'll have our ArrayList as our collection. We'll go ahead and create an instance of MyClass, which has a label v1 and the value abc. Another instance, the labels v2, values abc.

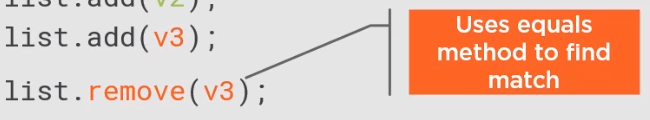
And another instance, labels v3 and the values abc.



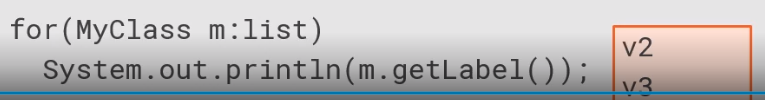
We add those three to the collection.



So now we say remove v3. Now, you might think that's going to remove the one we added, v3.



But remember, we're not doing a reference equals, we're using the value of the equals method, so basically it will walk through, in this case, in the first member it finds it matches or returns back and equals of true is removed.

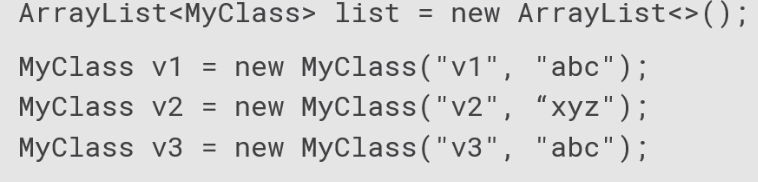
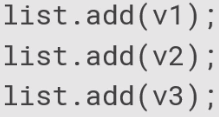
So if I go ahead and print this out now, I will get back v2 and v3. And this is because all three members had the exact same value and our equals test was only on the value. So just be aware of that as you're working with these methods. Okay, in the next clip we'll take a look at some of the features of the collection interface that rely on capabilities introduced in Java 8.

# Java 8 Collection Features

Slides=> pg. 14

Java 8 introduced a number of new features to the Java environment. One of those features is something called lambda expressions. Now, lambda expressions are a fairly involved concept overall, but fundamentally, what they do is they make it easy for us to pass code as arguments. The thing we want to understand is that collections have the ability to leverage lambda expressions to run code across its members. Now, one of the methods we'll look at here is what's called the forEach method, and that's actually inherited from the iterable interface. And what that allows us to do is simply pass in a block of code and have that code just be run for each member in the collection. Now, another method we'll look at here is what's called the removeIf method, and that takes a particular type of lambda expression called a predicate. Predicates resolve to either true or false. So what happens is that we can pass in that code, and then for every member for which that code resolves to true, that member is removed. Let's take a look first here at using our forEach method.

So let's go ahead and create an array list that holds the MyClass type we created just a little bit earlier.

We'll create three instances in MyClass, one with a label of v1 and a value of abc, another with the label of v2, value of xyz, and the last one with a label of v3, again, a value of abc. 

We'll go ahead and add each one of those to the collection. And so let's go ahead here and say we want to just go ahead and print the labels out for each one of these. Now we could write an explicit for loop, or we could use the forEach method, and we're going to pass it in the lambda expression.

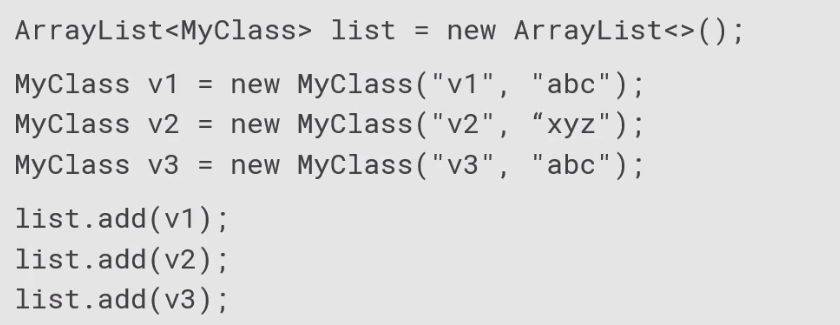
Now, the format of a lambda expression is first provide a parameter name, and that name could be anything you want; I'm just using the name m here. And then we use the arrow symbol, which is hyphen greater than. So we're basically saying we're passing m into this code, and that m will represent the current member in the collection, so m will represent each member in the collection once. And then the code we want to run is this System.out.println and then m.getLabel.

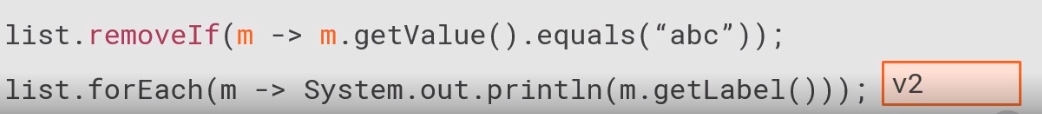
So what that means is that this method, System.out.println will be called once forEach member, and for that member we'll call it getLabel, which then, in turn, will print out the value of the label.

Slides=> pg. 15

So, it prints the first member's label, second member's label, and the third member's label. Right, so that was a really simple lambda expression. It's important to note that lambda expressions can be much more involved than that, but fundamentally, it's always the same. It's just kind of running the blocks of code once for each member.

So let's take a look at using removeIf. So let's go ahead and create that same array list we did before holding the MyClass.

We'll create those same three members we created earlier, and we'll go ahead and add each one to the collection, and let's say that we want to go ahead and remove each one that has a value of abc.



So what we'll do is we'll go to our list, we'll call removeIf, again, we're using a lambda expression. So we'll have a parameter name, again, I'm going to call it m, I could call it anything I want to, the arrow symbol, and then the code. Now, remember, this code is a predicate, so it has to resolve to true or false. So, first I'm going to say m.getValue, so for each member we'll call getValue and we'll do an equals comparison to abc. So, it'll walk through the first one and look, it'll get that value abc. That is equal, so that one's removed. Second one has a value of xyz. That's not equal, so it's not removed. Third one, again, has abc, so that one is removed.

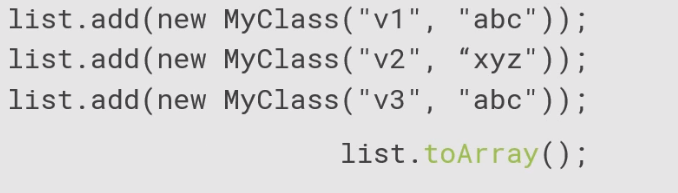
Slides=> pg. 16

So now if we go through and run through the code and the collection we write out everybody who now remains in the collection, the only one left is that second member, v2. All right, so the idea is lambda expressions make it just really easy for us to run blocks of code across members of our collections. Okay, now in the next clip, we're going to take a look at converting between collections and arrays.

# Converting Between Collections and Arrays

Slides=> pg. 17

Now there are times in our code where we need to be able to convert between collections and arrays, and what it comes down to is just very often maybe dealing with a particular API that requires an array even though you may be working in a collection. And also that's because you're dealing with legacy code or just some library code that you don't control. Now thanks to the methods on the collection interface, we can easily get an array from our collections. Now we use the toArray method. Now there's one version of toArray that takes no parameters. If you use that version, you'll get back an array of type Object containing the members from the collection. There's another version of toArray that accepts an array as a parameter, and that version of toArray will return back an array of the same type as the array passed in. Now it's also worth noting, though, is that if you have an array and you need it as a collection, the Arrays class has a static method called asList that will give us a collection containing the members of a particular array. Let's take a look at some code here. So first let's go ahead and get our collection as an array. So we'll go ahead and create an ArrayList again to continue as members of type MyClass.



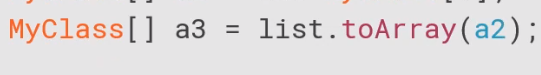
We'll go ahead and add three members to it. So now if I call list.toArray passing no parameters,

what I'll get back is an array of objects with each member being the three MyClass instances from my collection. Now that's useful, it's powerful, but often we want typed arrays, not just arrays of Object.

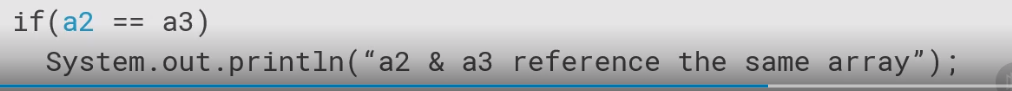
So what we can do right there is call list.toArray, but what we do now is pass in a typed array. Now a common technique we use here is we create up an instance of the MyClass array with 0 members, with a 0 size array. So this array we're passing in here is just used to indicate the type of the array we'd like back. So what toArray will do now is create a brand‑new array containing MyClass members and return that back. So a1 is a newly created array containing references to the three MyClass instances that were inside of the collection.



Now, if I go out here and I create an array that actually has room for the members, so a2 is a brand‑new array that has three elements inside of it,

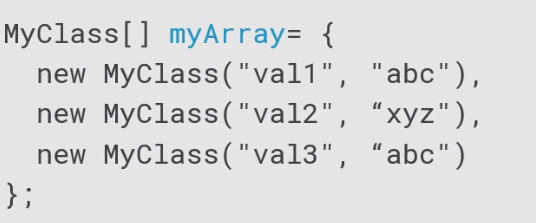


and I pass that array into toArray, toArray will return back a reference to the same array I passed in. In other words, if I pass an array into toArray that has room for the members in the collection, toArray will go ahead and use that array.



Slides=> pg. 18

So in this case, the variables a2 and a3 point to the exact same array. The toArray method used the array that I passed in because there was room to hold the members from the collection.



Now there are cases where I may have code that has an array, and maybe I want a collection. So let's say I have an array here of type MyClass. We have three members inside of it. If I need to get that as a collection, I can call the Arrays class's static method asList, pass that array in, and that will then go ahead and I'll get out a collection that has references to the members that were in the array. And that's a full‑blown collection, so I can go ahead and now use that as a collection,

so in this case I'm calling the forEach method and running code across the members in it. So you see, Java makes it very easy for us to move between collections and arrays.

Slides=> pg. 19

Okay, next let's take a look at some of the common collection types that we have available to us in Java.

# Collection Types

Slides=> pg. 20

Now, as I've mentioned, Java provides a wide variety of collections, and the reason is is that they each have their own specific behaviors. Now, the types of associated with collections are interfaces and, of course, actual classes. Now, the interfaces, those define the contracts for a particular collection's behavior, and then the classes, well, they're the actual implementations of the collection. And each of the collection classes implement one or more collection interfaces.

Slides=> pg. 21

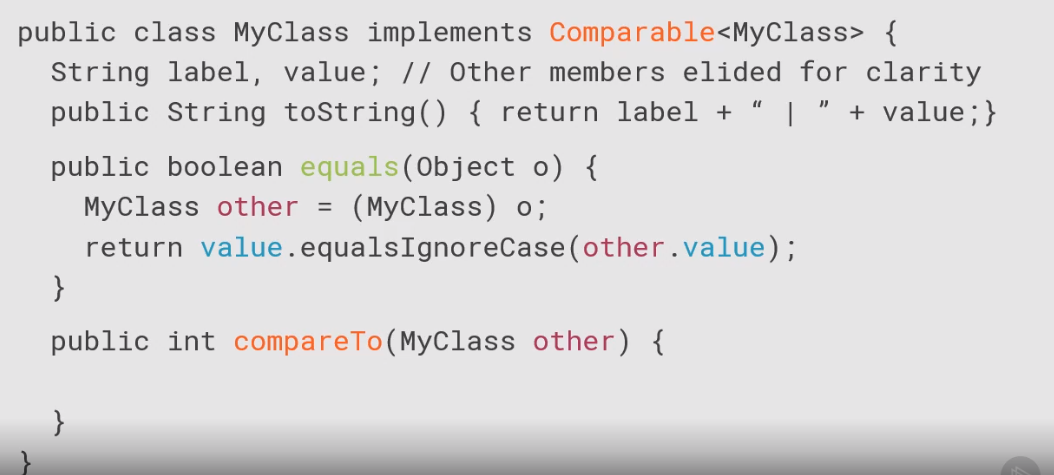
Now, if we take a look at some of the commonly used collection interfaces, well, one we're familiar with is the Collection interface itself. We've been talking about that one a bunch throughout this module. And that provides our basic collection operations. Another interface is the List interface, and that is used by collections that maintain some concept of order, whatever that might mean. There's the Queue interface, and that for collections that have a concept of order and a specific head element, kind of the one that's kind of next, in a sense, Then we have the Set interface, and that's for collections that have no duplicates, so you know that each value inside of there is unique. And then there's the SortedSet interface, and so that's for sets, meaning that there are no duplicates, and its members are sorted. So these are the contracts for behavior.

Slides=> pg. 22

Some of the types that implement those interfaces are things like the ArrayList. We've been talking about that guy a lot. So, an ArrayList is a list that's backed by a resizable array. Now it's efficient for random access, and what we mean by that is that it's very efficient to go I want the 7th element, the 200th element, the 15th element. But it's inefficient for random inserts. So if you've got a large ArrayList and you want to insert something between the 200th and 201st element, that can be expensive in an ArrayList. Now there's the LinkedList class, which is a list and queue backed by a doubly linked list. Now it's efficient for random inserts. So inserting a new member between two of the existing members is actually very efficient. But it's inefficient to do random access, so things like jumping to the 150th element, the 300th, the 15th. That's inefficient in a LinkedList. Now the HashSet, okay, that's a set that's implemented as a hash table. So it uses each object instance's hash value, and it's a good kind of general purpose hash table. Then there's the TreeSet, which is a SortedSet that's implemented as a balanced binary tree. So, it's a set, so each of the members are unique, and it tends to be inefficient as you're modifying it or you're searching through it, but it makes it easy to access the members in a sorted order. So there's lots of these collection types available. They each have their own individual benefits. So as you're working through your applications, take a look at what your individual needs are, and pick the right collection class for you. Now, in the next clip, I'm gong to talk a little bit about how those collections that rely on sorting actually handle sorting the members.

# Sorting

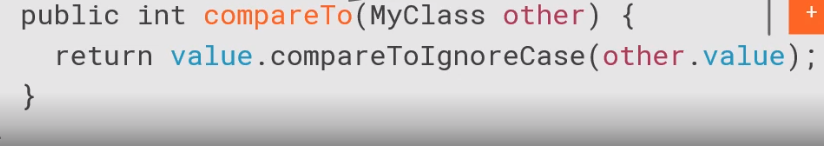
Slides=> pg. 23



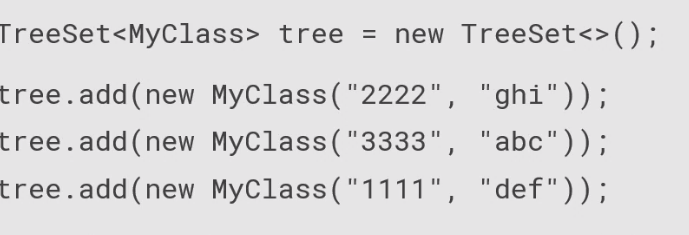
Now as we've seen, some collections rely on types being sortable. Now there are two ways for us to specify sort behavior. One is to use the comparable interface. Now that's implemented by the type to be sorted, and basically what that means is that a type is able to specify its own sort behavior. Now something to be aware of is that if a type implements the comparable interface, the behavior of the comparable interface should be consistent with the implementation of its equals method. In other words, if comparing two instances of the type using equals returns true, comparing those same two instances using the comparable interface should also indicate that they're equal, in other words, the types should behave consistently when compared for equality. Now there's another interface called comparator, and that's implemented by a type that performs a sort. In other words, it allows one type to specify the sort behavior for another type. This interface is useful for cases where maybe you want to sort a type that doesn't implement the comparable interface, or you have situations where you want to provide an alternate sort than that provided by the comparable interface. So let's take a look at these guys in code. So let's first look at using the comparable interface. So remember we've got this class, MyClass, and it has two fields, the label and a value field. Let's go ahead and add a toString method to it, so it just returns about the concatenations of the lable, and vertical bar, and value. And remember earlier when we implemented this class, we implimented the equals method. Alright, and the equals method relied on comparing the value fields of two instances. So since that's what our equals method uses, if we go ahead and impliment the comparable interface on this type, the comparable interface should provide the same behavior.

Slides=> pg. 24

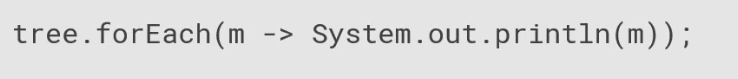
So now the comparable interface has one method, compareTo, and again it receives one value, other, and this method is responsible to indicate whether the current instance is less than, equal to, or greater than the other instance. And it does that by returning a negative, zero, or a positive value. So again we want to use the value field in the compareTo because that's when we also use an equals.



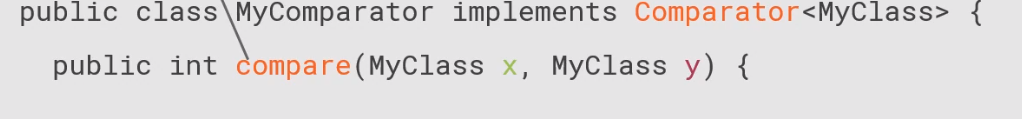
So we'll just add the value.CompareToIgnoreCase to the other. So now we have the ability to sort instances of my class, and those sorts will be depended upon the value field.



So now if we want to go ahead and use this with a collection that does sorting, we'll use it with a TreeSet, we'll go ahead and add three instances of MyClass to the TreeSet. Alright, the first has a label of 2222 with a value of ghi, next is a label of 3333 with a value of abc, and the last one has a label of 1111 with a value of def. So remember that the TreeSet maintains sorting.

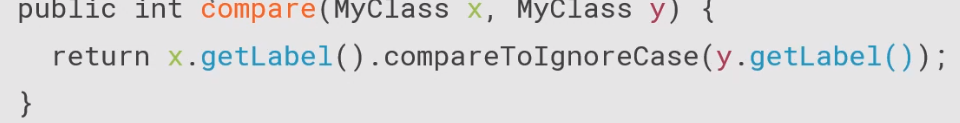
So if we go ahead to the TreeSet, and we walk through them one by one, and we print them out, we will now get them back based on the implementation of the comparable interface. Slides=> pg. 25

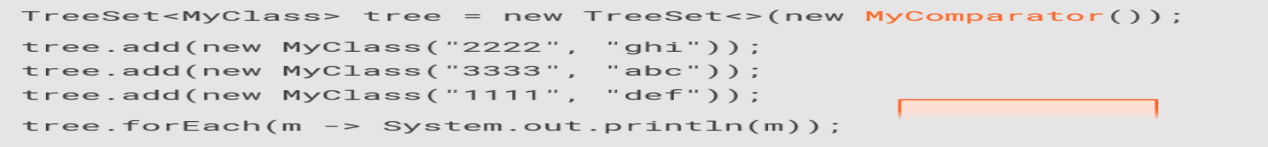
So the first one will come back as the value abc, so the next one comes back with the value def, and the last one comes back as the one with the value ghi. In other words, TreeSet used the sort behavior provided by the comparable interface. Now let's say we want to implement an alternate sort, and so we'll use the comparator to do that.

So we'll create a new class, I'm just going to call it MyComparator, and it implements the Comparator interface. So this class is responsible for providing sort behavior for another class, in this case it'll be the MyClass. It has one method, compare, and it accepts two instances. So it's responsible to indicate how those two instances compare to each other, basically, how the first one compares to the second one.

Slides=> pg. 26

So if it returns back a negative value, the the first one is considered less than the second one, 0 says they're equal, positive values says the first one is greater than the second one. So in this case, we'll provide a sort based on the label.

So the Comparator sorts them based on the label fields of each instance.



So we can also use this in a TreeSet. So we'll go ahead and create an instance of our TreeSet class, but when we construct this one, we're going to pass in an instance of our comparator. We'll go ahead and add instances that have the same labels and values as we had last time. But now remember this is using the sort provided by my comparator. So now when we loop through them, we'll now get the values back sorted by their label.

Slides=> pg. 27

So we'll come back with the first one labeled 1111, next will be labeled 2222, and the last one will be labeled 3333. So again, we were able provide sorting on the type itself with the comparable interface or sorting from an alternate type with the comparator interface. In the next clip, we'll take a look at working with map collections.

# Map Collections

Slides=> pg. 28

Let's look now at map collections. Now map collections are a little different because in a map collection, each entry is composed of two parts, a key and a value. People often refer to these as dictionaries as well, and the idea is that the key is used to identify or locate the value. Now in a map collection, each key is unique, but the values can be duplicated. So multiple keys can point to the same value if that makes sense in the problem that you're solving. Values can even be null.

Slides=> pg. 29

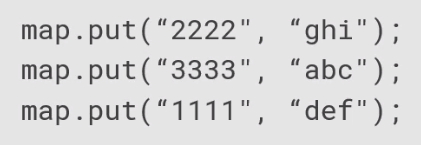
Now there are a few basic types that we associate with maps. First, let's look at the interfaces. Now there's the Map interface, and that provides your basic map operations. There's also a SortedMap interface. Now that extends the Map interface, and it's used for maps that have an idea of their keys actually being sorted. Now in terms of classes that implement these interfaces, one is HashMap, and that's just an efficient general purpose implementation of the Map interface. There's also a TreeMap, which is an implantation of a SortedMap interface. This is a self‑balancing tree. And so, in this case, the keys are sorted, and sorting can be handled either using the Comparable interface, in other words a type can have its own sorting, or the Comparator interface where one type sorts for another, just like we saw when we we're looking at the tree set.

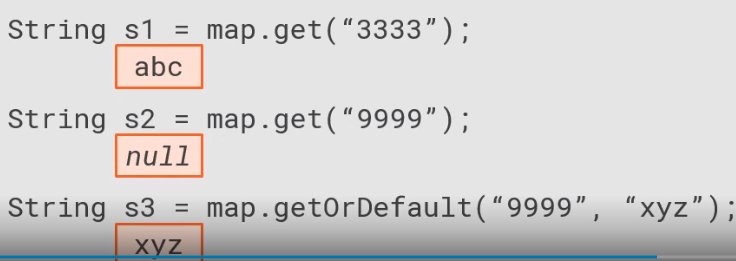
Slides=> pg. 30

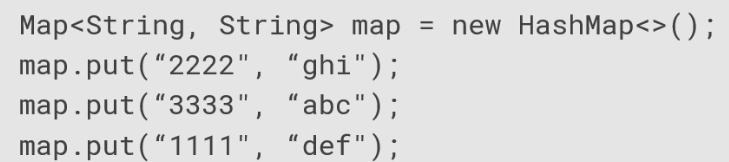
Now the Map interface has a number of methods on it, but let's look at some of the more commonly used ones. One is the put method. That's how we actually add a key and value to the map. There's a putIfAbsent method. What that says is I want to provide a key and a value, and I only want you to use this value if the key doesn't already have a value. So in other words, the key is not in there at all or the key is there, but the associative value is null. There's a get method where I provide a key and I get back the value. If that key is not found, it returns back a null. There's also a getOrDefault where I provide the key. If it's found, the value comes back. If the key is not found, then the value returns whatever I provided as my default. The values method allows me get back a collection containing all the values in that map. The keySet method gives me back a set containing all the keys in the map. Now map collections also support the idea of lambda expressions. So there is a forEach method that allows me to provide a lambda expression and perform the associated action for each entry in the map. And there's a replaceAll method that allows me to actually provide a lambda expression and modify the values associated with each key. So let's see these guys in code.

Slides=> pg. 31

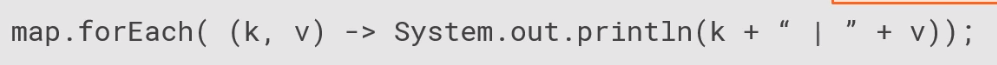


So let's set something up here. We'll go ahead and we'll declare a map, and we'll go ahead and create an instance of the HashMap. Now when we declare a map, we have to provide two type specifiers. The first one is the type of the key. So I'm going to say I have a map here whose key is a String. And then I have to provide the type of the value. I'm going to say that's also a String. Now they don't have to be the same. It can be any type I want them to be. But in this case for simplicity, we'll do a key and a value that are strings. So I want to go ahead and start putting some entries inside of here. 

So I'll say map.put, and it says that I'm putting an entry in whose key is 2222 whose value is ghi. I'll put another one in whose key is 3333 whose value is abc. And the last one we'll put in is 1111 for the key. The value is def. So accessing the members is really straightforward. So if I say get 3333, that will give me back to value associated with that key, so s1 would have the value abc in it. Now if I call map.get, passing in this key 9999, which is not in my map, well s2 would get back a null. But if I use getOrDefault and I say I want the key 9999, but I want the value to default to xyz, well if that key was there, I would get the actual value. But since there is no key 9999 inside my map, I then get back the xyz. Pretty straightforward.

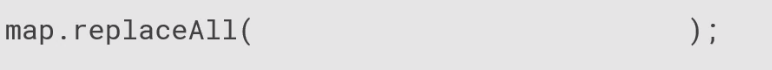


Now let's look a little bit at working with the lambda expressions. So let's go ahead and create an instance of our HashMap again. We'll add those same three entries.

So now if I want to go ahead and call this forEach method, so I'm going to go through and iterate over the whole collection. Now as I specify this lambda expression, I'm going to have to provide the parameters. But in this case, the lambda expression is what's called a biconsumer. And basically what it means is that it accepts two parameters. Now the parameter name can be any name I want. So I'll use k for the key and v for the value. So for each entry, it's going to call into this expression, passing in the key and the value for that entry. I'll go ahead and use the arrow notation. And then the code I want to execute is the System.out.println. And I concatenate the key, a vertical bar. and then value.

Slides=> pg. 32

So basically now that'll output out 2222 vertical bar ghi, then the next entry, and then the next entry, so really straightforward. But now let's say I want to go through and I want to apply some logic to modify all the values inside the map. So what I'll do is I'll use this replaceAll method.

Now again, this uses a lambda expression, but it uses one called a bifunction. Now a bifunction accepts two parameters, but also returns back a value.



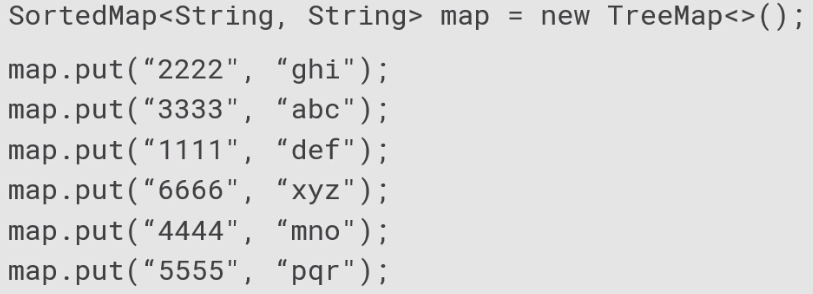
So what'll happen is that the code that I provide will run, receive the key and the value, and I'm returning back what I want the new value to be. So I'm going to have the same parameters, the key and the value, the arrow. But now what I'm going to say here is v.toUpperCase. Now what this is implying is that it's returning back v.toUpperCase. In other words, it calls toUpperCase on each value, takes a result of that, returns that back, and then the map applies that as the new value for that entry. So after I run this replaceAll,

if I then do a forEach again and print them out, it'll print each of the entries out, but now the values are all uppercase. This replaceAll method allowed me to provide the lambda expression that was then executed across all the entries in the map. and it allowed me to modify all those values using whatever logic I desire. Okay, now in the next clip, let's take a look at working with sorted maps.

# Sorted Map Collections

Slides=> pg. 33

Let's look now at some of the commonly used methods on the SortedMap interface. Now remember the SortedMap interface extends the Map interface, so it has all those methods, as well as some additional ones. So one it has is firstKey, which will give me back the first key in sort order that's inside of this map. LastKey gives you back the last key that's in sort order in this map. The method headMap, this will give me back another map that contains all the entries that have keys whose values are less than some specified key I pass in. TailMap kind of does the opposite. It gives me back a map that has all the entries whose keys are greater than or equal to the specified key I pass in. So that's important to note that headMap is exclusive of the key I pass. In other words, it doesn't include it, where tailMap is inclusive of the key I pass, so it does include it. There's also a subMap method where I actually provide a starting and ending key. So that would produce a map that contains all the entries whose keys are greater than or equal to the starting key I pass and less than the ending key I pass. So let's take a look now at working with our sortedMap.



So I'll go ahead and declare a SortedMap here, but I'll go ahead and instantiate this as a TreeMap because a TreeMap implements SortedMap. So this one, again, will have string keys and string values. I add entries to a using put just as I did with the regular map. But let's go ahead and put a bunch more entries in here. So we're going to go ahead and add six total entries in here, and they're in no particular order.

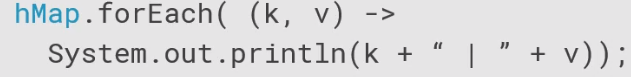
As I mentioned, SortedMap extends the Map interface, so I can call methods that are on the Map interface, like forEach.

Slides=> pg. 34

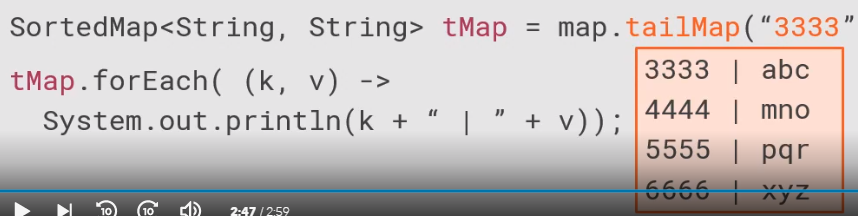
But now when I call forEach on this TreeMap, because it is a SortedMap as it displays the values out, notice the values come back in order by the key. So it takes care of managing that sort for me, and that's one of the real powers of the TreeMap is that it manages the sorting for us. But now let's look at some of the features that are specific to a SortedMap.



So I'll go ahead and create my TreeMap again. And we'll go ahead and add those same six key value pairs we did on the last slide. But let's use the headMap method. We're going to pass in the key 3333. So what happens now is a hMap contains a map that contains all the entries whose key is the last end 3333

So if I'm now on hMap, call forEach, and write all the entries out, I'll get the ones with keys 1111 and 2222. Again, it does not include the key I passed in.

Slides=> pg. 35



But now if I use tailMap, passing in the key 3333, and I go ahead and write out all the entries in the map produced from that, it will start with 33 through the end of the map. So again, it's just an important thing to understand is that headMap is exclusive of the key passed, tailMap is inclusive of the key passed.

# Summary

Slides=> pg. 36

To wrap up, here are some of the key things you want to remember. Remember that collections hold and organize values. Collections are iterable, and they tend to dynamically size. There are a number of different types of collections, but different collections can provide optimizations or certain sophistication. Now, by default, a collection will hold its entries as object, but collections can be type restricted, and we just use the Java generic syntax to do that. We associate a type with our collection, it will take care of returning our values appropriately typed, and that will enforce typing on any values that are added. Slides=> pg. 37

Remember that we can convert between collections and arrays. That's useful because you may be working with a collection, but then need to interact with an API that expects an array, or vice versa. Remember that the Collections interface provides the toArray method. You can get back the array as an array of object or as a typed array? And if you have an array and you want to do collection operations on it, you can use the Arrays class toList method. Remember that some collections provide sorting, and these collections can do it either using the Comparable interface, which allows a type to define its own sort, or using the Comparator interface, which allows a type to specify a sort for another type.

Slides=> pg. 38

Now remember the last collection we looked at was the map collections, and those store key value pairs. In a map, the keys are unique, although the values can be duplicated if need be. Remember that some maps support sorting their keys. Okay, so that wraps up our discussion of collections. In the next module, we'll look at how we can configure application execution and configure application environment.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*