16. Java's Generic Collections and Friends

Contents

[1. Introduction 1](#_Toc20386)

[2. The Problem with Arrays 2](#_Toc15209)

[3. Lists 3](#_Toc28993)

[4. Sets 4](#_Toc16603)

[5. Maps 5](#_Toc10393)

[6. Conclusions 6](#_Toc12271)

# Introduction

=>slides: Pg. 1

In this module of the course, we're going to be looking at the Java Collections API. And the Java Collections API is the absolute most common use case for generics. It's a situation where people use generic collections all the time in day‑to‑day programming. So how to look through some of the basics of the Collections API will help us understand how generics are used in practice and how we can use them in our own code.

=>slides: Pg. 2

Now firstly, what is a collection? Well, the idea is that you've got this kind of bag, this kind of box with lots and lots of different elements in. So here's a collection of old roman coins with funny faces on them. And the idea with the Collections APIs is there's a bunch of common data structures that we might want to use for different kind of groups of things. And each of these different data structures has different kind of properties, but because they're so commonly used, we're going to put implementations of them into the core Java library so anyone can use them.

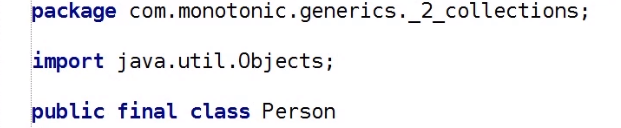
=>slides: Pg. 3

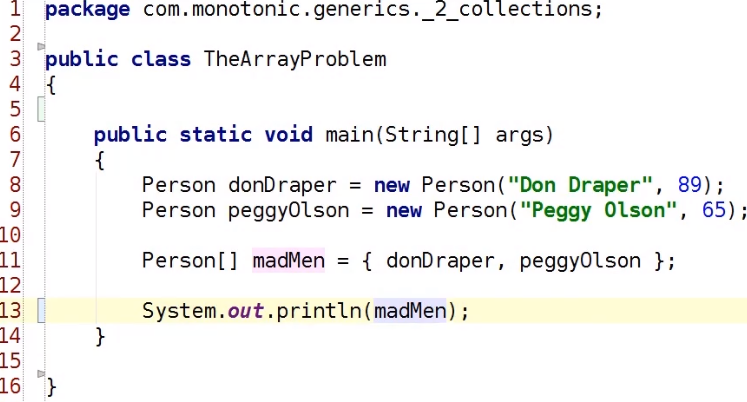
Well, what are going to be looking at that? The Collections API is actually very broad as a whole, and we're not going to be delving into the depths of all the different collections. We're just going to have a look at three different types of collections and cover a default implementation, an implementation which you might want to use with each of these collections. So we're going to have a look at a list, which is an ordered collection, a set, which is a collection with unique elements, and a map, which is a collection that's got keys and values and a one‑to‑one relationship between those keys and values. And in each case, we're going to be using an implementation of that collection interface. So an ArrayList for a list, a HashSet for a set, and a HashMap for a map. Now there are other implementations available in the core library, and it's, you know, a cool idea to go and explore lots of these things, lots of information out about that. We're just going to have a look at the basics.

=>slides: Pg. 4

So let's begin with looking at a list, which, as I say, is an ordered collection.

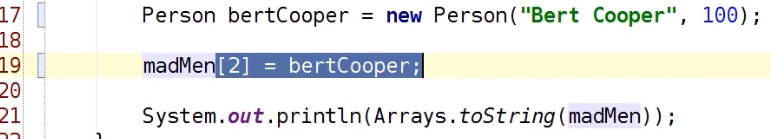
# The Problem with Arrays

So here's an example Person class. We're going to use this to understand how we can use the list interface and the Collections API and also how we can use arrays and what's wrong with arrays. So our Person class has a name and an age. And our name can't be null, and our age is an int. So it also can't be null. We've got getters for both the name and the age, and we've got implemented equals and hashCode methods and toString methods.

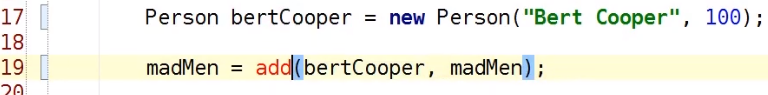


So let's have a look at using People with arrays. Bring up a main method so we can run it. And we're going to have a person called Don Draper. And just like the character from Mad Men, Don would be 89 years old if he was alive today. And his trusted companion, Peggy Olson. So we'll have a peggyOlson object, and she'd be 65 if she was alive today. So let's create an array of people, and these are going to be our Mad Men characters. And that array is going to contain our donDraper object and our peggyOlson object. And let's try and print out these Mad Men characters and see what happens. Wow. So the first problem we have with arrays is there's no easy toString implementation.

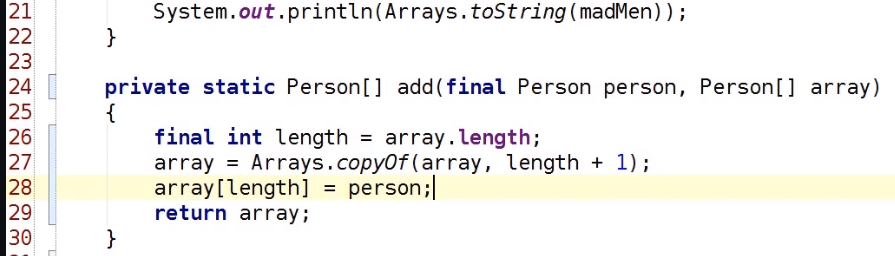
  
In fact, and this applies to a lot of the functionality on arrays, we have to use this Arrays static class and call to the toString method on it, passing in the array as a parameter in order to get a good toString. So if we run that, we can now see Don and Peggy being printed out on the console. Cool. Next thing, we've got our madMen list.



Let's try and add another element into our madMen list. So let's create a person, Bert Cooper, another Mad Men character, and we've got a bertCooper object. And Bert, well he must be like 100 or something by now. So we want to say madMen, array is a 0 index. So to add the third element, we do madMen(2) is equal to bertCooper, and then we'll print out that madMen array at the end. Well, here's the thing. We get an ArrayIndexOutOfBoundsException because we can't add a new element to an array. So what would we do?



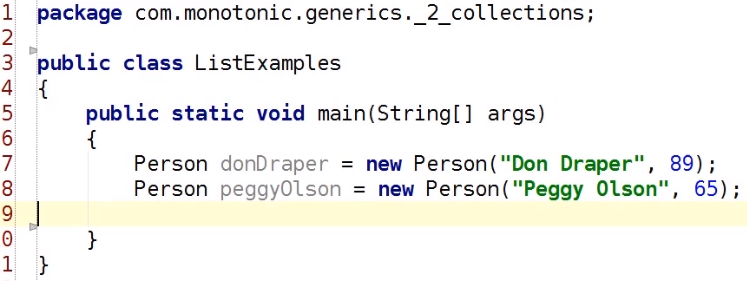
Well, we'd have to say madMen is equal to and have a method to say add our element bert to our madMen array.



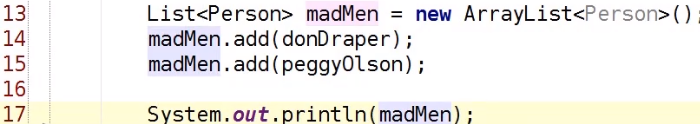
So let's create this static method with an add like that. So we're going to say take our Arrays class, and we're going to say give me a copy of this array with the new length one larger. We can't make that final anymore. And then we're going to return this array, and then we want to say‑‑‑ The original length is the index, which will allow the person out, so we do array of length is equal to a person. And if we rerun this, we'll see we've got Bert Cooper as the new element in the array. Lovely stuff. But it's not really lovely stuff, is it? Because we've had to go and copy the entire array. We've had to assume that it's the last element. We've had to write all this boilerplate code. It's confusing, and it's error‑prone as well. Hmm. So it's hard to expand arrays. It's hard to print them out. What about if we want to find out if element is in there? Well, there's no contains method. In fact, there's barely any functionality on arrays at all. They're an okay primitive construct in a language, but arrays are really not what you want to do or use very often when you're trying to work with data. So let's have a look at how we can use the collections list interface and see how this improves upon arrays.

# Lists

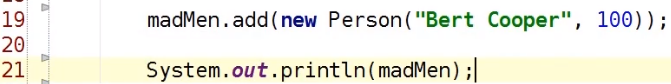
=>slides: Pg. 3



Cool. So now, let's think about how we can use the list interface as a better solution to the problem that we encounter with arrays. Well, the first thing to do the list is to declare a variable. So list is an interface in the java.util package, and importantly for this course, it's a generic interface. So we're going to say create a List of Person. Again, they're going to be Mad Men characters. And in this case, we might think that we could just do a new List of Person, but we'd be wrong. List itself is an interface. There are different ways you could implement a list. All a list means is that it's a collection with an order. So we could implement a list using an array underneath and implement those kind of manual add methods. We can implement a list using a linked list. We could do all sorts of different things if we wanted to. Now our List of Person, we're just going to use an array list.



This is the most commonly used implementation of list, I would say, and it provides very efficient access for random indexing and for iteration over it. Okay, so we're going to say a List of Person is a new ArrayList of Person. We'll import the ArrayList class at this point. So ArrayList is our implementation of list that uses an array underneath to implement the functionality. And you'll notice that our ArrayList class is also generic. So we've got a List of Person, madMen, is equal to a new ArrayList of Person. Nice and simple. Now we can add our elements into the list. So here we're going to say madMen.add(donDraper) and madMen.add(peggyOlson). And then we can print out our madMen list and run the code, and we'll see we have an ArrayList implementation printed out on the console. There's a nice toString method. We can see it, and we can read what's going on. Lovely stuff. Now the problem we had with arrays was we had to implement our own expansion and resizing strategy. And lists conveniently solve this problem for us.



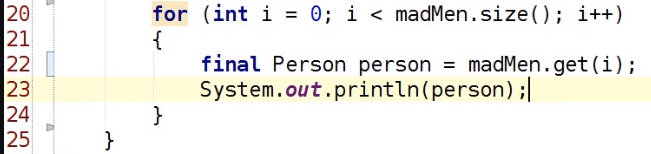
So our madMen object, we can say add new Person. I'm going to call him Bert Cooper, and we're going to give him an age of 100. So we add Bert into the list, rerun the code again, and actually print Bert out, and we can see that he's been added to the list. So that add method automatically resized the list when we added it. The other thing to bear in mind is that because these methods are generic, we have some kind of type safety. So if I say madMen.add and I try and say new Object, I'll get a compile error. It'll say, you know, we've got this method called add and add should only take a person, and you're trying to pass an object in. That's not safe. Everything has to be a person. Nice and simple, nice and easy to make correct code. It's also possible to query the list a number of other ways.  
  
So, for example, we can ask for its size, and that gives us 3. Or we can also ask our list for elements at different points.



So as I said, a list has order, and that means we can say madMen.get(0), and that would give us the first element in the list. So that print out Don Draper. Or we could say madMen.get(2), and that will print out the third element in the list, and that will be Bert Cooper. Fantastic.

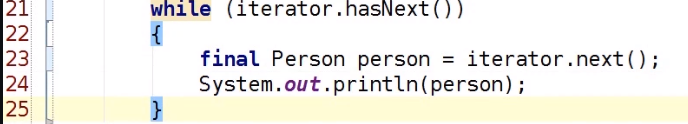


Now if I just extract a variable here, you can see that this get method also returns a person as well. Because our list interface is generic, the list can have a generic return type for methods like get as well. So that is another example of how generics are being used in this Collections API. And finally, let's take our get and get out all the elements in the list. So we know we can get size and see the number elements, and we know we can get get and pass in an index to get out one element.

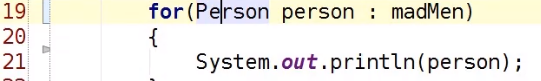


So let's remove these other printouts and loop over the list. So I'm going to say for int i = 0; i less than madMen.size; i++ and then madMen.get(i) for person and then print them out. So there's just a very simple example of me printing them out calling get, getting the correctly type person, and then printing them out. Now lists like many collections offer actually a more advanced function for being able to loop over them like this. You don't have to use the index and call it by index on every time.

  
What we could do is we could call the iterator method on madMen, and that would give us a Person iterator. And you'll see that that's also another generic interface. And an iterator has two methods on it.   
One of them is hasNext. It's got a few more than two, but two important ones. And the other one is next. And hasNext returns true or false whether there's another element remaining, and next gets that element out.



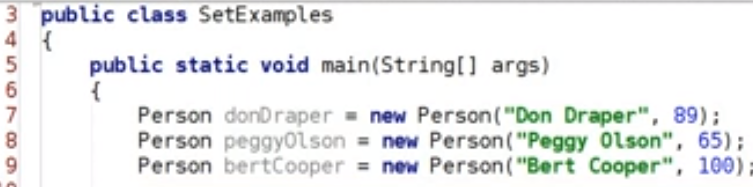
So we can rewrite this loop by saying while iterator.hasNext, and then we can say get our person, delete that because we don't need it anymore, call the next method on the iterator, and run this code again. So you print out Don Draper, Peggy Olson, Bert Cooper, and they're all in the order that we've put them into the list. So if I swap these two adds around and put Peggy in before Don and rerun it, we'll see when we print it out, Peggy comes out before Don. That ordering is respected. So we've said we've got this iterator construct. It's actually possible to go one better than that because Java has a foreach loop, which was actually introduced at the same time as generics in Java 5.

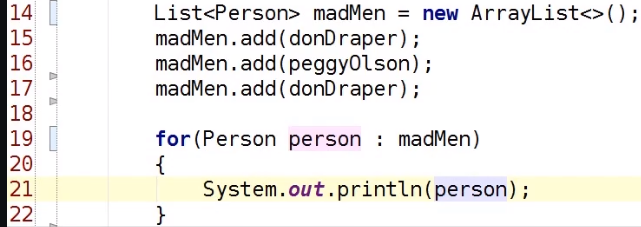
  
And so we can just say for each Person of type person in madMen, print them out. There we go. That's even simpler and even less error‑prone than our iterator approach. There's one more thing we can do just to make this even better, and that is, in Java 7, the concept of the diamond operator was added.

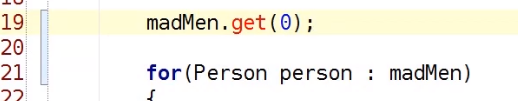
So this here is a diamond. Our diamond means don't add the generics in explicitly, just infer from the context. So you know this is a List of Person, and you know that ArrayList implements list. So you know you can construct an ArrayList of Person and assign it to the madMen variable. We'll be using diamonds quite a lot moving forward. And all it means is just take the generic parameter from the left‑hand side and fill it in as the generic parameter in our constructor. And if we run that again, we can see it does exactly the same thing as if we'd put a person in there explicitly. Cool, that's lists.

# Sets

=>slides: Pg. 5

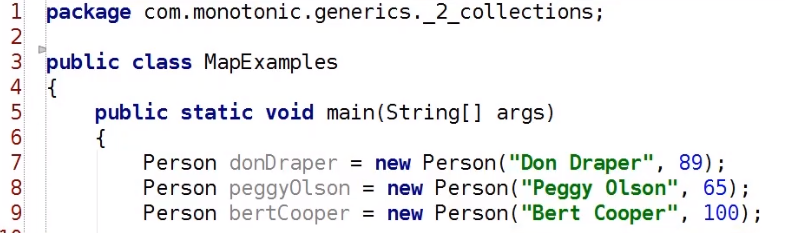
So we've had look at lists. Let's now have a look at sets. A set is a collection that has unique elements. What do I mean by that? Well, let's suppose I had my list, my generic List of Person, my Mad Men characters,

and I added Don Draper to the list, and I added Peggy, and I added Don Draper again. And I said for each person in my list, let's print out that person, and I ran this code. I would see, as you can see in the console at the bottom, that Don Draper is in that list twice. Array lists don't ensure uniqueness of elements, and they have ordering. So it makes sense to have two Don Drapers in the list. But sets only allow unique elements.

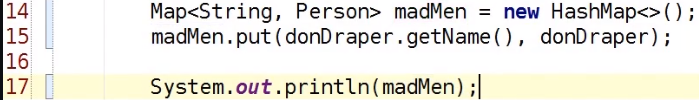
So I've got a Set of Person, and the implementation that we're going to use with respect to a set here is a HashSet. So I can say new HashSet. I'm again going to use the diamond operator. Again, both set and HashSet are generic, so I can say a Set of Person, and I can say a new HashSet of Person. Or I can remove that person and have the diamond operator on the right‑hand side. And if we rerun this code, we'll see that we've got Don Draper followed by Peggy Olson. So we don't have two Don Draper in our madMen set. Perfect, unique. The other thing this code demonstrates is the idea that you can iterate over a set just in the same way as you can iterate over a list. So I've got the exact same code here for our foreach loop, and it's working on set just like it's worked on a list. But sets don't have order.   
So if I say madMen.get(0), that's a compile error. There's no get method. And in fact, there's no defined ordering for elements in a set at all. So this foreach loop, it could be Don, then Peggy. It could be Peggy, then Don. It all just depends. So what other things does a set have that a list is not so good at?

Well, we can also check a contains methods. So if we do madMen.contains(donDraper), and if we run the same code with madMen.contains(bertCooper), we'll see true or false is printed at. So we added our donDraper element into our set. So we call contains. It returns true. We didn't add Burt into the set. So when we call contains, it's false. Another nice property of sets. So just to recap a set, we can still iterate over it, just like in a list. It doesn't allow duplicates in there, only unique elements. And we've got some convenience methods, like contains that help us efficiently check membership of hash sets.

# Maps

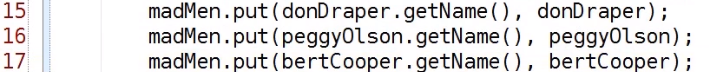
=>slides: Pg. 6  
So now we're going to look at the third and final type of collection, and that is a map. Now a map is a bit like a dictionary. You know a dictionary has got words in, and for each word, it's got a definition. And the idea with a dictionary is you can look up the definition given the word, and maps work exactly like that. But instead of restricting to just words and definitions, you can use them for any types of values, and we refer to those things as keys and values. So keys are unique, and for each key, there is a single value associated with it. What does that look like in code?   


Well, let's take, again, our madMen example. We've got our map interface, and we can see from the definition of our map interface that it has two generic parameters.

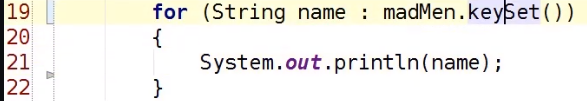
So I'm just going to make it a String and a Person. Well let's import the map interface, again, java.util, just like the other collections. If we look at that interface, we see it's got two generic parameters. One's called K, and the other one's called V. And K is the type of the keys in our map, and V is the type of the values in our map. So here, we're going to have a map from strings onto Person objects. And again, we're going to use the diamond operator on the right‑hand side. We could have said String and Person there, but we don't need to. We can just use the diamond. What do we actually mean by a map of String and Person? We actually mean that we want to be able to look up Mad Men characters by name. So if we take our madMen map, instead of an add method, there's a method called put, and the put takes the key and the value. So we can say donDraper.getName and donDraper. And if we were to print out this madMen map with just this one element in, we'd say donDraper=Person blah, blah, blah, blah, blah. So we've got a mapping from Don's name onto his other properties.

And in fact, we can look up that name so we can say madMen.get Don Draper, and that would just return us the value that's associated with that key. Now you can associate one value with multiple keys, but each key has to be unique.

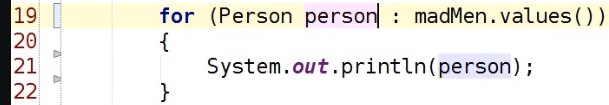
So if I come along here and say madMen.put donDraper.getName peggyOlson, and I rerun this code, it'll overwrite the previous entry. And when I try and look up the entry for Don's name, I'll get Peggy returned back as the value. So we don't want to do that.

What we really want to do is put Don in by name, Peggy in by name, and Bert Cooper in by name.

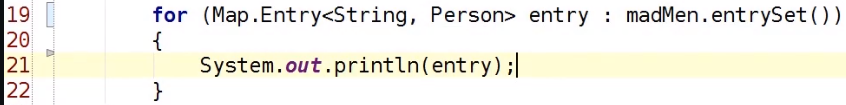
  
And then if we print out this whole map, we'll see we store multiple entries. Don maps to Don, Peggy maps to Peggy, and Bert maps to Bert. Okay, lovely stuff. So we've looked at get, and we've looked at put. The other key property that we have is iteration. So with other lists, there was just a single value. So if we wanted to iterate, we would just iterate over the members of the list or the members of the set. With a map, we've got keys and values. So we've actually got three different iteration combinations.



The first one is the keys. So we can say madMen.keySet. And what that method does is it returns us a set with all the names in it, so I could loop over them and print out everyone's name. So let's run that code. We see Don Draper, Peggy Olson, and Bert Cooper. Cool.



The other thing I could do is I can call the values method, and that returns a collection, which is the generic interface that sits above list and set. And this is a person. So now if we loop over the values, we get the Person objects, so name is Don Draper, age is 89 etc., etc. But maybe you actually want to do both. Maybe you want to have both the names and the values.

Well, there's also an entrySet method, and the entrySet method returns us an entry element. And an Entry element is generic. So it has two type parameters. One of them represents the key, and one of them represents the value. And an entry is just one element in the map. So if we look over these entries, we see we get one for Don, one for Peggy, and one for Bert. Fantastic, lovely stuff.

Entry, because it's a generic method, can call a getKey method that would return a string, and we can also have a getValue method that would return the object. So if we run that out, we see Don's name, then his object, then Peggy's name, then her object, then Burt's name, and then his object. So we can individually access all the keys and values as we're looking over them. We can access the values by calling get on the madeMen object, and we can call put on the map to add entries. So that's maps, which are quite a cool collection in and of themselves and also an interesting thing from a generics point of view because this is the first time we've seen that syntax for having two generic parameters, the first one, then the comma, and then the second one. An in actual fact, there's no hardcoded limit. You can have as many generic parameters on a class as you need to, though my general recommendation would be not to go above two or three and to try and keep it as simple as possible. But there we go. That's maps and hash maps. Thanks.

# Conclusions

=>slides: Pg. 7

Let's summarize what we've learned about collections and how they interact with generics.

=>slides: Pg. 8

The first thing we talked about was that for each type of collection, we had an interface with multiple implementations, and we looked at one specific implementation. So we had a List, which was an interface and an ArrayList, which was an implementation of that interface. And our ArrayList had a specific data structure backing it. So in the case of an ArrayList, it was just an array underneath the hood. And that had certain performance characteristics, such as order one time insertion speed and order end time increment speed because you might have to resize the array. ArrayLists and specific implementations are also concrete classes, so we can instantiate them. So whenever we have an assignment, we always see the interface type on the left and the implementation type on the right. From a generic point of view, they both have the same type parameter. So if you have a List of Person, you should have an ArrayList of Person as well. But if you've just got the simple assignment case, you can use the diamond operator for the ArrayList to say List of Person is equal to new ArrayList of diamond. And it's always called a diamond because you see the less than and then the greater than angles next to each other, which looks a bit like a diamond. Interfaces define functional characteristics. So a List was ordered. A set was a collection with unique elements. And you should always try and prefer the interface as the type for your variable. And the reason why you want to do that is because it means you can change your List of Person from being an ArrayList of Person to another implementation, for example a LinkedList of Person. And that makes it easier for you to pick the right data structure and migrate and maintain your code over time.

=>slides: Pg. 9

The other thing to note is that we've introduced the Collections API. We haven't gone into much detail, and there's a lot more that we could cover. But we have understood the basics. Lists and sets are both generic collections, and they're generic on the elements type that they contain. Maps have keys and values, and we have two generic parameters accordingly, one for the key and one for the value. Collections are the absolute most common use of generics. So understanding collections is really important when we use generics because it's a really useful use case. Cool. So now you've understood generics, the basics at least, and you've understood collections. In the next module, we're going to go on and think about how we can have classes and interfaces and implementation hierarchies that use generics. So we saw the our ArrayList of Person implemented List of Person. How does that work? What's going on? That's the kind of thing we're going to have a look at next. And I hope you'll continue listening and enjoying the course. Thank you.

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