15. The What and Why of Java Generics

Contents

[1. Introduction 1](#_Toc10242)

[2. An Example Class: Circular Buffer 1](#_Toc30689)

[3. Boilerplate vs. Type Safety 1](#_Toc9886)

[4. Making Our Circular Buffer Generic 1](#_Toc28109)

[5. Summary 2](#_Toc21184)

# 1. Introduction

=>slides: Pg. 1

We're going to be looking at what generics are. We're going to be looking at all the advanced features. But we're also going to be looking at some of the introductory features, so we aren't assuming that you know anything about generics. But before we cover anything about the details or any of the nitty‑gritty or tricks that you can do with generics, let's first answer one simple question.

=>slides: Pg. 2

Why do we actually need generics? What are the problems they're here to solve?

=>slides: Pg. 3

Okay, so let's just take a quick look at this code. We've got a list which is being set up for us, and we think it's a list of strings, and we want to call .get and get the first element out of it, and we do that by calling .get(0), and then we assign that value to a variable called result. And result is a string, so we need to cast the value to a string and then we can print out the result. So this code is just printing out string, that is the first value in a list

=>slides: Pg. 4

If we run this code, you might encounter an error like this. It says ClassCastException, java.lang.Integer cannot be cost to java.lang.String. What that means is our value that we tried to cast to a string was actually an integer, and, obviously, an integer isn't a string so you can't cast it. Here is the reason why that might occur.

=>slides: Pg. 5

When we instantiate our list, we say List list is equal to new ArrayList. We add the string a, the string b, and the number 1 in. And the problem here is that a collection can contain any object. We could have added strings. We could have added integers. We could have had a people. We could've added horses. We could have added cars. Collections are inherently heterogeneous. They can contain any type of object. And that presents us with a type safety problem throughout our application. We don't know whether our list has only strings or only numbers. And in reality, most of the time, we want our list to only contain a certain type of object because we want to be able to do something with it, and we want to be able to do it safely.

=>slides: Pg. 6

Generics solve this problem. On this slide, we've altered the declaration of our list to be a list of string. Now don't worry about angular brackets. Don't worry about any of that kind of stuff yet. We'll look at the details and understand what it all means over the course of this module. What this really is just saying, this is a list that only contains strings. And when we have list.add(1) in red, we will get an error. We won't get an error at runtime. We'll get an error before our application runs.

=>slides: Pg. 7

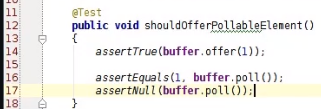
Generics stop runtime errors at compile time. They convert that problem which could blow up in our face in production when customers are using our application before it ever runs. They help us have type safety. Now I'm going to look at a more extensive example. We're going to write our own class that is going to be a collection that contains various values called a circular buffer. What we're going to do is explore our different possibilities because generics isn't the only way we could have had type safety. We could have created multiple different buffer classes. We could have done lots of different things. And we'll see how these other solutions aren't quite as good as generics, and we'll show you how you can actually code with generics.

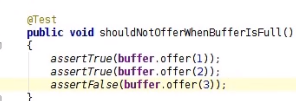
# 2. An Example Class: Circular Buffer



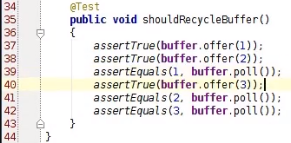


   
Why would we want generics? Well, first of all, let's have a look at some code that doesn't use generics, and we're going to see what the problems are with using that code. And then we're going to convert that code to be generic and see how our code becomes a little bit cleaner and a little bit more reusable as a result. So, well, first of all, we're going to have a look at this data structure called a circular buffer, or you might have heard of this as a ring buffer. Now a circular buffer is a kind of queue. So elements that go in first come out first, and it's a queue with a fixed size. So here's our test code. If you don't understand testing, you can have a look at my course Automated Testing in Java, which covers the basics of how to do unit testing with JUnit.

  
Now this circular buffer has a constructor that takes a size, which represents the fixed size of the buffer, and this is the kind of behavior we might expect when we interact with it. So there's an offer method and a poll method. And the offer method means try to write something to the buffer, and the poll method means try to read something out of the buffer. And offer will return true or false depending upon whether it succeeds. And if poll has a value that we can read out, it'll return that value. Otherwise, it'll return null.   
If we offer one element to the buffer, it should return true. If we then immediately call poll without making any other changes, it should return the number 1 back to us. And if we call poll again, it should return null.

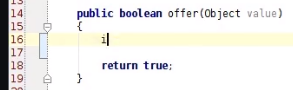
  
As I say, our buffer is fixed size. So if we offer two elements in, they should both return true. And when we try to offer the third element in, it'll be full, so it will return false.

  
If the buffer is empty, then our poll call should return null.

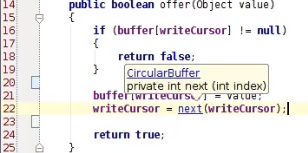
  
And, obviously, we expect it to be able to reuse slots within the buffer. So if we offer two elements, it's now full. But then if we poll an element out and then reoffer another element, we can add that element in. So we can keep on going reusing the buffer with as many elements as we want, but only two at a given point in time. So how would we implement this circular buffer?

  
Well, we're going to keep an array of objects, which is going to represent the elements in our buffer. If we don't have a full buffer, then the slots in that array, which are empty, we're going to use the null value to represent.

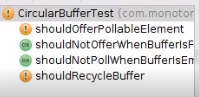
  
So we've got two fields here, a readCursor and a writeCursor. And the readCursor is going to be the point where we want to read data out to the buffer at, and the writeCursor is the point at which we want to write data into the buffer at. Okay, there's quite a lot to think about with circular buffers. But don't worry, you'll see where we're going.

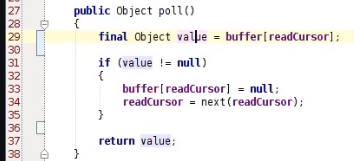
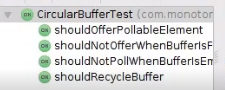


So when we try and offer a value into the buffer, the first thing we need to check is whether we can offer the value into the buffer.

  
So we want to say if the buffer's writeCursor is not equal to null, i. e. if there's not a slot available for us to write into‑‑‑ Now if we're still going in that method, then we know that we've got a slot in the buffer, and we can write stuff into it. So we're going to say the buffer's writeCursor is equal value, and then we're going to take our writeCursor, and we're going to move it on to the next element in the buffer.

  
And this next element has got a trick in it. So what this next element is doing is saying add 1 to the index of the buffer. And when you get to the end of the buffer, loop back to the beginning. And the way we do that is we increment the index by 1, and then we modular it with the length of the buffer. Okay, so that means if it's a multiple of the buffer, it goes back to 0.

  
Now if we run this code in our test again, we'll see we've got two passing tests because our offer method works. But our poll method still just returns null. So how do we implement the poll method? Well, let's have a look at this.

  
Firstly, we want to have the same logical cases we had in the offer method, which is to say we want to say take our buffer,  
  
look at the readCursor, and see if we've got a value to read. And if that value is available to read, then it's not null. I'm just going to extract a local variable, so that's the value we want to return. Now if it's not null and there is something to read, then what we want to do is we'll take our readCursor, and we want to make it the next element. So the next time we call poll, we look at the next elements to read out. The other thing we want to do is we want to take our buffer and our readCursor, and we want to null out that element in the buffer because bear in mind now we've read the value, we've got a value, we've read it. Then, we don't want to read that value again. We want that value to be removed from the buffer. It's just a temporary storage thing after all. So at the end of that method, we're going to return the value we've read. Now note that if this element is empty and it's null, we're going to return null from the poll case, that's one of our tests. So it works either way. You can return value regardless of whether this if statement passed or not.   


If we call that, we can see all our tests pass. So there was a bit of a whistlestop tool, but there we go. We've implemented a nice, simple, circular buffer data struct. So how do generics come into play?

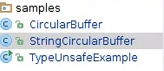
# 3. Boilerplate vs. Type Safety

  
So let's have a look at this example code, which uses our CircularBuffer. We've got a buffer, and we're instantiating it with 10 elements. And we're going to add some strings in, a, b, c, and d, and then we're going to call our concatenate method. So our concatenate method repeatedly polls the elements of the buffer and appends them to a string builder, and then it will print out the result.



So what we would expect to see if we run this is abcd all on the command line. Perfect, works exactly as expected. So just to recap, we offer three elements in, we concatenate all the strings in the buffer, and we return the result on the command line.



But there's a problem with this, which is that if I take my buffer and I offer the number 1 in here, we get an error. There we go. ClassCastException. Integer cannot be cast as string. So why does that happen? Well, because we're trying to append strings into our string builder, we cast the result of the poll method into a string. A bit of a nasty one that. Now our offer method was just taking an object as a parameter. So there was no way we could say all the elements in our CircularBuffer were strings. And as a result, we had a compile error. Well, what could we do to solve this problem?   
  
Well, we can take our CircularBuffer, and we could force the parameter to be a string. So we could say StringCircularBuffer.



Now our Object array can become a String array.



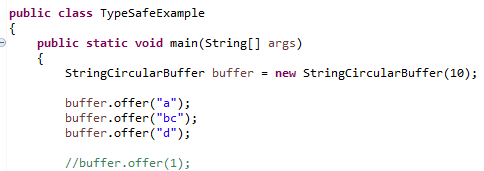
So we're going to instantiate a String array there.

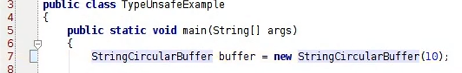


Our parameter for our value can become a string,



and our poll method can return a String. And this variable value has to become a string as well. Okay, that's all right. We've now got a StringCircularBuffer.





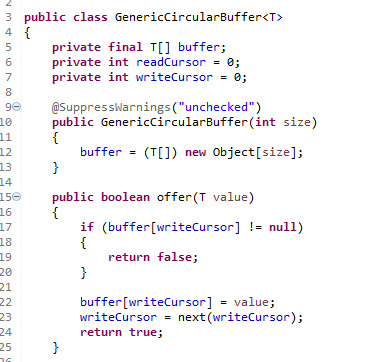
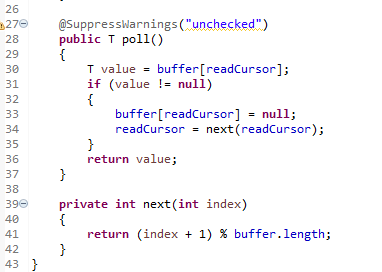
And if we take our example code, and we replace our CircularBuffer with a StringCircularBuffer, we can   
see, first of all, we can remove this cast because we don't need a cast anymore. The poll method just returns string.



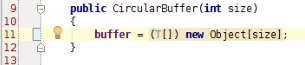
And we can see we get a compile error here saying you're trying to pass an int in where a string is required. So we've converted a runtime error into a compile time error. We've made use of the fact that Java is statically typed. Okay, fantastic. And we now can delete that code and rerun our example, and we're back to printing out abcd. Now you may not think this kind of thing is the kind of stuff you do. Why would you ever try and offer a 1 into a string buffer? But it's the kind of error that programmers easy make. Especially as the systems they develop become larger and more complicated, it's easy to typo or add something to the wrong variable. And static typing allows us to catch those errors early before they bite us in production. So we've got this StringCircularBuffer. What's the problem with just using a StringCircularBuffer? Well, the problem is that we might want to have circular buffers of all sorts of things. You don't have strings or integers, or maybe we want to have a CircularBuffer of people or any class that we have in our domain, all sorts of different things. And in order to implement that StringCircularBuffer, I had to copy and paste the implementation of my CircularBuffer and add in as a StringCircularBuffer. Now this is really quite bad because it means that any errors I made in my CircularBuffer, I might have copied and pasted into my string CircularBuffer. It means that if I want to say do something like support having multiple threads offering elements into the buffer, I need to go update all of my copies of that CircularBuffer. It's just generally a bad idea and leads to harder‑to‑maintain code. So we've linked to one approach, which is to have a CircularBuffer that takes

objects and another approach, which is a string specialized CircularBuffer which is type‑safe, but requires us to do copy and paste. How do we get around this problematic choice of either needing copy and paste code or needing to have unsafe code?

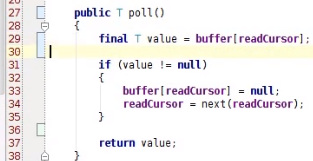
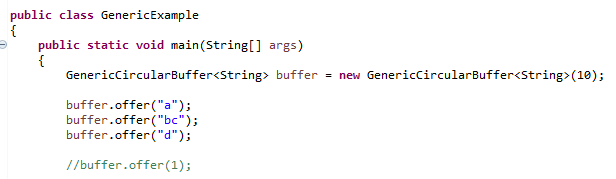
# Making Our Circular Buffer Generic

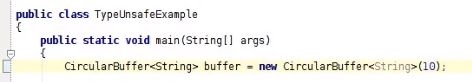
  


  
Well, this is where generics come into play. So how do we solve this conflict between type safety and boilerplate? We have generics. So let's take our CircularBuffer, and what we want to say is this CircularBuffer works on a certain type. Every element in the buffer is of this type. I want to call that type T. In our class definition, we put these little angular brackets after the class name and just T, and we say that's a type parameter, so like a parameter to a normal method, but this time a parameter to a class, and it's a type rather than a value. So we could say T is string. We could say T is an integer. We could say T is a person. And it can be any any Java class or interface that we want. Our buffer array is also going to be of type T because that means every element in the buffer is going to be a T.

  
Now what I would like to be able to do at this point is say new T of size, but Java won't let me do this, and I'm going to explain why that's the case later. So for now, what we're going to do is we're going to say cast our new object array to a T array so we can put it in our T array buffer.

  
Our offer method needs to take a value parameter that's of type T, so that gives us the requirement that anything that we write into the buffer is of type T.

  
And when we poll, we want to make sure that anything we we return is also a T. And what that means is the value that we return from our buffer also has to be a T. Okay, nice and simple. And the rest of the code can all remain exactly the same. In our type‑unsafe example before, which is now going to be a type‑safe example, we had a StringCircularBuffer.  




What I'm actually going to do is I'm going to say have a CircularBuffer and make it a CircularBuffer of Strings. There we go. So we're saying instead of having a CircularBuffer, it's a CircularBuffer of String. And to instantiate it, we call a constructor parameter that says new CircularBuffer of String, and then we offer the strings in,



and our concatenate method also needs to have a CircularBuffer of String. So this acts exactly like our StringCircularBuffer, except we haven't had to copy and paste it everywhere in order to implement it. We've just had to add this generic parameter in that says it's a CircularBuffer of String.

  
And importantly, we can also see that when we try and add the number 1 into the buffer, we get a compile error with the exact same compile error saying you're trying to pass an int in and it has to be a string. So there we go. We've used generics, and we've got code that's both type‑safe and doesn't have to have boilerplate everywhere. And if we run it, we can see it prints out abcd, so it does the exact same thing. Fantastic.

# 5. Summary

=>slides: Pg. 7

So we've now answered the question why do we need generics and what are they?

=>slides: Pg. 8

We've got this situation where we want to use our compiler as a bit of a lock, a bit of a safeguard to make sure that we can't do things like adding an integer into a string buffer. And we decided that we don't want to do that by just copy and pasting our buffer again and again and again because it leads to maintenance errors. So we've used generics to try and get the best of both worlds, type safety, but without copy and paste.

=>slides: Pg. 9

The rest of this course is going to cover a number of different topics. So we're going to look at generic collections, first of all, because really the most common use of generics in the Java API and in the Java ecosystem is to write collections‑oriented code. And we're just going to cover the basics of lists, sets, and maps, so you'll get a great idea of how to use generics in conjunction with collections. Then, we're going to move on and look about how generics work with interfaces, classes, hierarchies, how these things all fit together. Then we're going to talk about the fact that the generics we saw in this module were at the class level. We have a circular buffer of T. We can actually have generics at the method level. So we could have, for example, a sort method that's generic on the type of things we're going to sort. Then we'll look at wildcard generics. So this is a bit of a more interesting, more esoteric feature, but still somewhat commonly used, a more advanced feature maybe I should say. Then we're going to look about how generics offer compatibility. So even if you've got some code still left around that doesn't use generics, you still have compatibility with that code, and you can still migrate to using generics incrementally throughout your code base. And finally, we're going to talk a little bit about how reflection interplays with generics and how Lambda expressions, a new feature in Java 8, also work with generics. So that's the course outline. Hopefully you'll find all those topics interesting, and hopefully you already know how to use the most basic case of generics for simple classes. Thank you very much.

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