14. Persisting Objects with Serialization

Contents

[1. Introduction 1](#_Toc12770)

[2. Java Serialization Overview 1](#_Toc12131)

[3. Being Serializable 3](#_Toc5582)

[4. Serializing/Deserializing an Object 3](#_Toc8117)

[5. Transient Fields 4](#_Toc31932)

[6. Demo: Serialize and deserialize an object 6](#_Toc7908)

[7. Summary 6](#_Toc20580)

# 1. Introduction

=>slides: Pg. 1

 In this module, we talk about persisting objects with serialization.

=>slides: Pg. 2

Now throughout this module, we're going to look at the aspects and issues involved in Java serialization. We'll first take a look at the purpose and capabilities of Java serialization. We'll then see what's involved in making a type support serialization. We'll then see exactly how we serialize an object out to a serialization stream, and then restore that object information back from that serialization stream.We'll then look at the issue of transient fields, in other words, the issue of having a type that supports serialization, but that type may have specific fields that we want to exclude from serialization. Okay, so let's get started.

# Java Serialization Overview

=>slides: Pg. 3

One of the really powerful concepts that's built into Java is the ability to persist objects. And what we mean by that is that we can take an object that exists in the runtime and store that object out to a byte stream. And then we can take that byte stream and completely reproduce that object. And in most cases, taking advantage of this capability requires very little programming because the Java runtime leverages reflection. By using reflection, it can know all the members of your type, and it can get and set the values of that type. Now by default, this capability only operates on instance members because it's really designed around restoring instances of objects, but the ability is highly customizable. So if you needed to incorporate statics, you could do that by customizing it.

=>slides: Pg. 4

Now having this ability to persist objects like this opens up a ton of possibilities. One thing is the simple idea of saving state. If you could actually have objects that represent the state of your application, you could save them out to the file system. When the application restarts, you can restore them back and have the state fully reproduced. You can also store your objects inside of a database. And so the idea is that rather than looking at individual fields on a table and manually setting those values in your object each time, you can actually store and retrieve entire objects. If you're working with a relational database management system , it will generally store these objects as a blob. With other systems like object‑oriented database management systems can actually directly store and retrieve objects using serialization. This also gives us the ability to pass objects across address boundaries. So you can actually take an object graph that's in one application space and then pass it over into another application using things like shared memory and so forth. Or you could even pass them across the network. So an object presentation on one machine can actually be passed to another machine and restore it over there.

=>slides: Pg. 5

Now this capability is referred to as serialization, and serialization really has two sides to it. On one side is the side of serializing, and that's storing out to the byte stream. And the idea is that you pass in an object that you want to store. But serialization is actually very intelligent that when you pass it an object, it doesn't just store that object. It stores the object and any other object that it points to. That's what we call an object graph. So when you pass in an object, the entire object graph referenced by that object is then passed out to the byte stream and has all the information about the types that are involved there and all their values. So that's the serializing side. The other side is deserializing, and that's the idea that we restore that object graph. So we can start out with the byte stream and then basically just take that byte stream and have the entire object graph completely rebuilt. So when we use the term serialization, we're really including both sides of this, the idea of serializing out to the byte stream and then deserializing from the byte stream.

=>slides: Pg. 6

Working with serialization really revolves around three core types. There's the serializable interface, and that's implemented by any type that wants to be serializable. And it simply indicates that that type supports serialization. Now the funny thing is is that the serializable interface has no methods. It's what we call a marker interface. It simply means that if I implement that interface that I have verified that my type is capable of being serialized and deserialized accurately. Now in order to be able to serialize the type, we use a class called ObjectOutputStream, and basically that has the ability to take an object graph and write it out to a stream. Then on the other side, we have ObjectInputStream. It actually has the ability to take the stream of bytes that was serialized and then deserialize that back to the object graph. Okay, so now that we've seen the terminology involved in serialization and the types that are involved, in our next section, we'll see exactly what's involved in making a type serializable.

# Being Serializable

=>slides: Pg. 7

So if we have a type that we want to be serializable, well what do we have to do? We know that one of the key requirements is that we implement the Serializable interface. But now as we mentioned, that interface doesn't have any methods. It's a marker interface, meaning that if we implement that interface, we're saying that we know our type is serializable. But what does that really mean? Well, in general, for a type to be serializable, all of its members need to be serializable. Now for primitive type members, that's not a problem. Those are all serializable by default. But for other members that are not primitive types, in order for them to be serializable, they themselves have to implement the Serializable interface. =>slides: Pg. 8

So let's take a look at some code and see what this is like. So let's take the BankAccount class that we've been using throughout much of this course. Remember, we had two core fields here. We have a balance field, which is an int, and then we have an id field, which is a string. We have a couple constructors along with the other members. So now if I go off and I new up an instance of this BankAccount class, passing in an id value of 1234 and a starting bounce of 500, we know that allocates out the memory for our BankAccount. We have our balance field, which will have an initial value of 500. And then we'll have our id field, which will have an initial value of 1234. Now again, we want our BankAccount to be serializable. Well balance type is int. That's a primitive type. So our balance field is indeed serializable. But now what about the id? The id field is of type String. String is not a primitive type. String is an object. So what that means is that in terms of what the memory looks like for our BankAccount, the 1234 is not actually stored inside of the memory of the BankAccount. Actually, there is separate memory allocated, which is the String object. That's what contains the 1234. Inside of our BankAccount class, it's simply a reference to that instance of the String class. So what that means is that our BankAccount class is a simple object graph. The root of the graph is the BankAccount object. The node of that graph is the 1234, the string. So since our id field is not a primitive type, it's a string, which is an object, that means that we need to verify that that class String is serializable. So what we need to do is go out and check the documentation on the String class. And we'll notice that the documentation of String class says that the String class implements the Serializable interface. So that's saying that that class String is indeed serializable. So that means that our other member id is indeed serializable. So now we know our BankAccount class is serializable because all of its members are serializable. So what we'll simply do now is in our BankAccount class implement that Serializable interface where there are no methods. But now by implementing that, we're indicating that yes, our BankAccount class is itself serializable. Okay, so now in the next section, let's see what code we need to write in order to serialize and deserialize instances of our BankAccount class.

# Serializing/Deserializing an Object

=>slides: Pg. 9

So now we know our BankAccount class is serializable. So let's go ahead and create an instance of the BankAccount class, again we'll give it an Id of 1234 and an initial balance of 500. Let's go ahead and make a deposit of $250, so now our current balance would be $750. Let's say we want to go ahead and serialize this out to a file called account.dat. So we'll have a method we write called saveAccount that accepts a reference to the BankAccount to be serialized and the file name. So saveAccount will have those two parameters, alright, the BankAccount to be serialized and the file name. And now the work of actually serializing your BankAccount will be done by the class ObjectOutputStream, and that can serialize out to any stream. What we'll do is go ahead and create an OutputStream over that file that was passed in. So now when we're done there, we actually get a reference to an ObjectOutputStream that knows how to serialize instances and objects into that file. We'll go ahead and put this in a try with resources. So we close the stream as soon as we're done. We will need the handle IOExceptions because we are dealing with the file system. So now how do we actually serialize the object? Well all we do is call the writeObject method against our ObjectOutputStream, passing in a reference to the object we want to serialize. And so that easily, it writes out the type information for the object being serialized, the value of its members, as well as any type information required for the members that it contains. So it does that for the whole object graft, meaning that anything that our object references, any other objects that those objects reference. So then how do we actually read it back in?

=>slides: Pg. 10

Let's go ahead and create a method here called loadAccount that accepts a file name that we want to deserialize from, and it returns back a reference to a BankAccount. We'll create a local variable here that's of type BankAccount, and now we're going to use ObjectInputStream to do the work of deserialization. Again, we'll open up a file stream over that file name, which then give us back a reference to an object input stream that can now deserialize the objects from that file. Again, we'll put this in a try with resources and handle any IOExceptions. So now to actually get the object back out of that stream, we simply call the readObject method against our ObjectInputStream. Now remember that the serialization information contains not just values, but the type information of what's been serialized. Well, it's possible that you try to deserialize a type that's not in your class path. If that were to happen, the readObject would actually throw a ClassNotFoundException. But now in our case, we do have access to all the types, so the readObject actual returns back a reference to the BankAccount class that has all the values that were contained at the time that were serialized. So we can simply return back that BankAccount reference. So that means that we can then call this loadAccount method, passing in the file name, again, this is the same file name that we serialized to in the last slide. That gives us back the reference to the BankAccount in the exact state it was in at the time we serialized it, which means that if we now write out its Id and balance, we'll get out an Id of 1234, and the balance at the time it was serialized, which was $750. And so you see that easily, we were able to capture the state of an object, save it, and then later restore it. But now there is one potential issue here. What if we serialize our BankAccount class with our two fields in it, but then later changed the fields that are inside of our BankAccount class, we actually change the class definition, how does the serialization system handle the scenario of deserializing a type that has changed since it was serialized? And that's what we'll take a look at the next section.

# Transient Fields

=>slides: Pg. 11

There are some scenarios where you may want to type the support serialization, but you may not want all the fields in that type to be part of the serialization. This is useful for scenarios where you have fields in that type that could be derived from other fields and that's helpful because it avoids taking up unnecessary space in the serialization storage. Now this is a case we can actually use the transient keyword to identify the fields that we don't want to be part of the serialization, and only what we'll then do is that when we restore that type, we'll go ahead and set the value of the field marked transient manually. One of the ways we commonly do that is by customizing the serialization.

=>slides: Pg. 12

So let's look at an example. So I have a class that we'll call my AccountGroup. The AccountGroup has a map inside of it that can hold bank accounts, so the key will be the id of the bank account, which is a string, then, of course, the value will be the bank count itself, and we'll use this account group for scenarios where maybe a person has multiple bank accounts, want to keep those grouped together or maybe a family has the parents with accounts and children with some accounts. So AccountGroups allow us to keep those together. So one of the things we'll have is a field that tells us the total balance of all the accounts in a group, of course, we'll have a getter for that. Then what we'll do is that each time we add an account to the group, we'll go ahead and take the balance from that particular account, add it into the total balance, and then take the account and added into the map. So now, if we look at our AccountGroup, the only field in it is the accountMap field. Well the accountMap field is a map and we know that the bank accounts inside of that map can be serialized, we've already determined that the String class supports serialization, and if we were look at the documentation on hashMap, that also supports serialization, so that means that the only field in our AccountGroup is serializable. So that means, of course, then we can take our AccountGroup and mark that as Serializable, but we know as this is written, our total balance field can be determined from getting the balances from the accounts contained in the group. So what we'll do then is take our totalBalance field and mark it as transient. So by marking it is transient, what that means is that when we serialize instances of AccountGroup, the value of the totalBalance field won't be written to the serialization stream, and then when we deserialize instances, there will be no value read back for the total balance field.

=>slides: Pg. 13

So let's see what it's like now to use our AccountGroup class. What we'll do is we'll start out by creating instances of our BankAccount class that has a balance of $500. We'll create another instance of our BankAccount class that has a balance of $750. We'll then create our AccountGroup and then add both of those accounts into the AccountGroup. So the total balance in our AccountGroup now will be $1250. So what we'll do is we'll call a method called saveGroup that will actually take care of serializing an instance of our AccountGroup into a file called group.dat. Now our saveGroup method will look very much like the serialization we wrote much earlier in this module. So what we'll do is go ahead and open up an ObjectOutputStream over the file, we'll then put a try catch in place to handle the exceptions, and then we simply will take our ObjectOutputStream and call its writeObject passing in the object to serialize and that easily we've now serialized our AccountGroup instance.

=>slides: Pg. 14

So now to go ahead and deserialize our AccountGroup instance, we'll create a method here called loadGroup that accepts a filename, returns back the newly loaded AccountGroup instance. We'll have a local variable for the AccountGroup, in this case, we'll go ahead and open up our ObjectInputStream over the filename, we'll handle our exceptions, we have one for the IOException, one for the ClassNotFoundException, and then we'll take our ObjectInputStream, call readObject to read back the AccountGroup, and then we'll simply return that back from the method. So let's go ahead and call our loadGroup method. What that'll do now is give us back the deserialized version of our AccountGroup, so this will be AccountGroup that actually contains the 2 bank accounts we added on the previous slide or that 1 had a balance of $500, the other had a balance of $750. So if we now go ahead and take our AccountGroup and call getTotalBalance and write that out, what do we get? Well, we get back 0 because remember that we've excluded total balance from the serialization process.

=>slides: Pg. 15

Alright, looking here at our AccountGroup class because of that transient modifier on totalBalance means it doesn't get serialized out or read back. So we'll need to do here is go ahead and customize our serialization so that we can go ahead and set the value of totalBalance. We don't need to make any changes in terms of how we serialize AccountGroup, it's only the deserializing we need it to handle. So we'll go ahead and add a method readObject that takes that object input stream. Alright, so this is now us providing our own custom deserialization. So what we'll do now is just go ahead and call the default readObject because we wanted to deserialize the type just as it always does and that'll take care of all the details of deserializing the hashMap that corresponds to our accountMap field, but what we want to do now is go ahead and manually set the total balance. Now we can do that by simply looping through the accounts in the accountMap and then getting the balance from each one of those and adding them into the total balance, alright, and that easily now we'd go ahead and had our totalBalance field, which was transient had its value properly set using other values we had access to as part of the deserialization. Now notice that we only have an implementation of readObject. We don't have a implementation of writeObject. Because we don't need to modify the way the actual serializing occurs, there is no reason for us to provide the writeObject because the only thing we would do inside of there is call the default writeObject method. So in cases where you only would be calling the default writeObject method, there is no reason to provide the writeObject implementation. That's worth noting because in our previous section, we did provide a writeObject that the only thing it did do was call the default writeObject and that was just to demonstrate how you provide both methods, readObject and writeObject. In a case like here like we have with AccountGroup, because we only need to change the way it's deserialized, we only have to provide implementation of readObject.

=>slides: Pg. 16

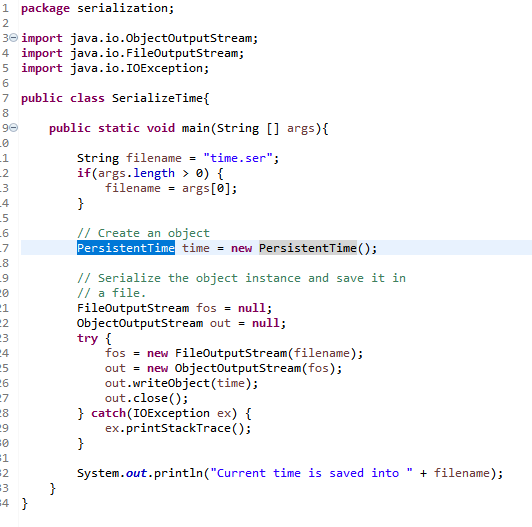
So now if we call loadAccount, deserializing our AccountGroup contained the 2 accounts, right, 1 with $500 balance and 1 with $750 balance, now if we go ahead and write out the total balance from the AccountGroup, now we get the proper value $1250.

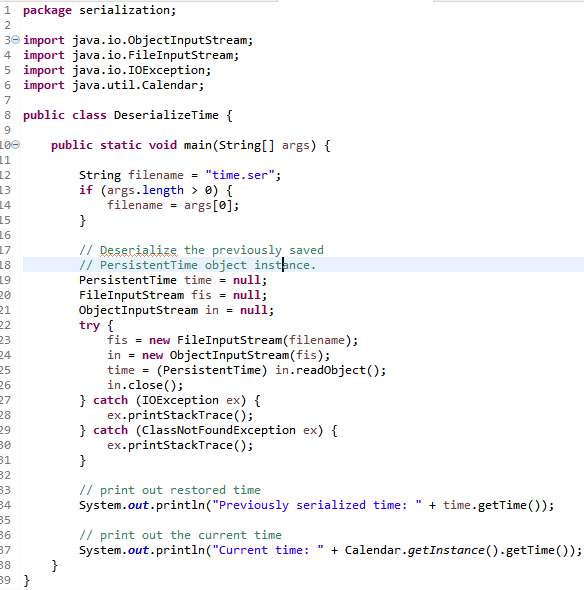
# Demo: Serialize and deserialize an object

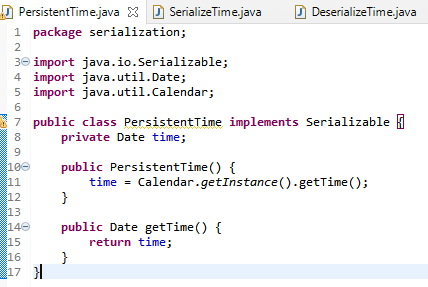
=>slides: Pg. 17

In this exercise, you will learn how to do serialization and deserialization of the an object.  You will also learn how to use transient keyword.











# Summary

=>slides: Pg. 18

That wraps up our discussion of serialization. So here are some of the key things you want to remember from this module. First, I'll remember that serialization provided a way for object persistence, a way to store and retrieve our objects. We could do things like store them in the files or databases and pull them back, and we could even pass objects between processes or across the network. Now, when it comes to a type being serializable, remember that primitive types are implicitly serializable, but classes must implement the serialized interface. Now remember that interface has no methods on it, it's just a marker interface, but basically by implementing it, our class is saying that it knows how to be properly serialized and deserialized. Now when it comes to the task of serializing and deserializing our types, remember that there were two core classes here. There was the ObjectOutputStream class, which took care of actually using reflection to look at our class instances and writing that content out to a serialization stream, and then there was the ObjectInputStream that could take that serialization stream and then rebuild our objects from that information.

=>slides: Pg. 19

We looked at this idea of marking fields as transient and it was a way to exclude a field from the serialization process and that was used when we had values that could be derived from other values so we didn't waste space in the serialization stream, serializing it and deserializing it. And during the deserialization process, generally, what we would do is use custom handling to then go off and set the value of those fields that were marked as transient. So now we've reached the end of our course.

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