Contents

[Esoteric Approach to Saving Objects into a File 1](#_Toc483676907)

[Using -D Option 2](#_Toc483676908)

[Throwing Runtime Exceptions – From Service to Business Layer 4](#_Toc483676909)

[Best practices around catching specific exceptions? 4](#_Toc483676910)

# Esoteric Approach to Saving Objects into a File

Typical approach, in persisting objects to a File would be to use the ObjectOutputStream to write an object and use the Object InputStream to read an object.

A cleaner approach is to use a data structure such as a HashTable and save your objects inside this HashTable with a unique key and then persist this HashTable into the file system.

**Note: Java doc on Hashtable demonstrates how the Hashtable works - https://docs.oracle.com/javase/7/docs/api/java/util/Hashtable.html**

In other words, the flow at a high level for persistence would look like this:

**Creation of Hashtable and saving it to File steps:**

1. Map the Application Objects into a HashTable, with each application object identified by a unique key.

*Hashtable<Integer, Customer> customerHashtable = new Hashtable<Integer, Customer>();*

*customerHashtable.put (1, customerOne); // customerOne is mapped to 1*

*customerHashtable.put (2, customerTwo); // customerTwo is mapped to 2*

1. Persist HashTable into a File

**Retrieval steps:**

1. Extract HashTable from the File
2. Now locate the object of interest by the unique key.

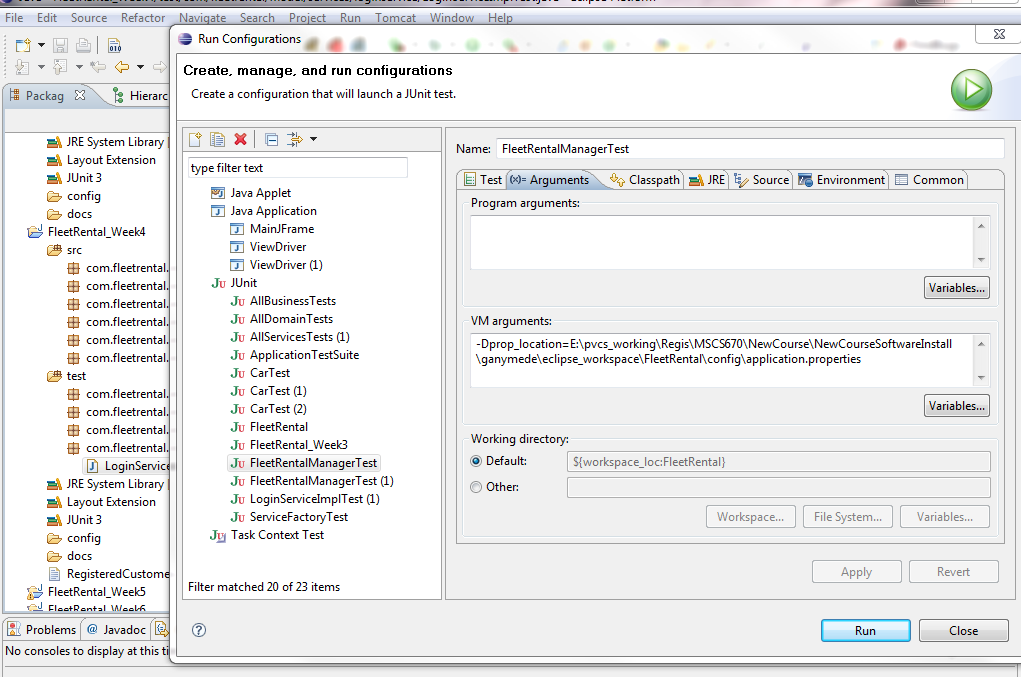
*Customer customer = customerHashtable.get(1); // retrieves customer tied to key 1.*

To apply this technique, read-up on HashTable first and draw your design on paper first to confirm the approach before doing any coding.

# Using -D Option

To pass values using the –D option to your unit tests

1. select the unit test
2. Right click and go to Run as -> Run Configurations
3. In the Dialog that ensues (below), go to Arguments tab and add your –D option



Analyzing Stack Trace during Debugging

From a debugging standpoint, I’m sure we all concur that proper logging (using System.out.println in this course) aids in isolating problems. If we reflect on what we’ve done so far, we are already logging at different parts of the code especially in the catch block. This logging is invaluable during debugging and it’s important to pay close attention to the stacktrace when it’s generated as it reveals the precise location and the nature of an error.

For example, during a debugging session with a fellow student last night, in less than few seconds of looking at the stacktrace was able to isolate and get past the initial hurdle caused by an exception.

Granted that the Text has details (and the supplemental material) on exceptions, detailed below is a simple example just to complete the picture.

Hope it helps! Please let me know if you have any questions.

**Analyzing Stack Trace During Debugging**

Following is a made up stack trace for purpose of our discussion.

1. Exception in thread "main" java.lang.NullPointerException
2. at com.fleetrental.manager.RegistrationServiceImpl.registerCustomer(RegistrationServiceImpl.java:72)
3. at com.fleetrental.manager.FleetRentalManager.performAction(FleetRentalManager.java:36)
4. at com.fleetrental.Driver.main(Driver.java:18)

**What does this tell us?**

As a ‘stack’trace, and we know a **stack** data structure is a LIFO (last in first out). Which means that the first line (line 1 above) in the log was last entry that was pushed into the stack and hence considered the source of the problem.

**To break things down:**

**Line 0:** Tells us that it is a NullPointerException (NPE). Quick search on meaning of this shows us that we are referencing an object that hasn’t been instantiated.

**Line 1:** Tells us (based on the definition of a stack) that line 72 located in registerCustomer method of class RegistrationServiceImpl is the source of error. **Here is where the fix needs to be applied. Invariably with NPE the quick fix is to ensure that the object is instantiated.**

**Line 2:** Tells us that line number 36 of performAction method of FleetRentalManaget is invoking the line above it.

**Line 3:** Which is at the bottom of the stack (or first entry pushed into the stack) as the original line of code that initiated a call that led to the NPE.

# Throwing Runtime Exceptions – From Service to Business Layer

Study the attached code, in the sequence as shown below:

ThrowingRuntimeExceptionsService.java

ThrowingRuntimeExceptionsManager.java

ApplicationIllegalArgumentException.java

# Best practices around catching specific exceptions?

FileNotFoundException inherits from IOException.  ClassNotFoundException means a different class was serialized or the file read was corrupt.   Either way, Java could not read the object instance that you need.  That is definitely bad but not actionable by your application or user.

There are many exceptions that might be thrown.

What is the best practice to decide which explicit exceptions to catch?

What value would be lost if we did the following:

public processMemberShip(Member member) throws MembershipException

try {

…

}catch (Exception ex)

{

ex.printStackTrace();

throw new MembershipException(...);

} ?

Catching more types of exceptions might add code complexity but at some point it could stop making the method more robust or maintainable and not be necessary?  How do you find the right balance?

**Answer**

Excellent question!

Yes, based on the calls being made within a method, multiple exceptions could be thrown.

1. Off the bat, catching just the exception(called the catch all) is highly discouraged, especially while we are learning best practices or even otherwise it's frowned upon in the real world.

try{

    .. number of call outs

    } catch (Exception e) {  // this is bad!

    }

    finally {}

2. Catching specific exceptions would allow one to handle and provide meaningful action to the calling methods and the logging of the errors aid during debugging. Ending the catch with an Exception is considered fair approach as this enables catching any missed specific exception.

try

{ //code that might throw exceptions}

catch (MalformedURLException e1)

{ // emergency action for malformed URLs}

catch (UnknownHostException e2)

{ // emergency action for unknown hosts }

catch (IOException e3)

{ // emergency action for all

 // other I/O problems

}

catch (Exception e4) {}

finally {}