Advanced Java 8 Programming

version 2.0

Lab Guide

7437 Race Road Hanover, MD 21076

IN1458-LG-03 / 20211005

Contents

Lab 1: Using Lambda Expressions

Overview

In this lab, you will use three different ways to define a method for implementing Java logic. We will implement the behavior with a nested class, a local class and anonymous class, and lastly using a **Lambda** expression.

This lab can be done using an editor with a Java command line or an IDE, such as **Eclipse**. IDEs are much easier, but it can be done using the command line as it is just Java code.

If you are using the command line you will need to know where the JDK is installed.
In Windows it is normally:
C:\Program Files\Java\jdk{the jdk version}.

Mac usually installs to:
/Library/Java/JavaVirtualMachines/jdk{the jdk version} or
/System/Library/Java/JavaVirtualMachines/jdk{the jdk version}.

The bin folder contains the executables such as **javac**. JREs install to a different path and usually are added to the PATH so they run without specifying the path. Opening a command window and typing "**java -version**" without the quotes normally will produce output like **java version "1.8.0_66"**.

1. Open Eclipse or a command window.

2. Create a **Project** named **LambdaExercises** as a **Java Project**. Or create a folder in the home directory named **LambdaExercises** to run the exercises in.

   Add a package named **com.example.lambda** to the **LambdaExercises Java Resources**: **src** folder.

   If using the command line, create a directory structure under the LambdaExercises\com\example\lambda for Windows or LambdaExercises/com/example/lambda for Mac.

3. Copy the file **Main.java** from the **Setup\Exercises\Lambda** folder to the package just created. This is an example of a class using a nested class, an inner class and inner interface.

4. Eclipse will automatically build the project and compile the code. If you look in the **bin** folder of the workspace you will see the directories built and several class files. If you are using the command line you would run **javac** in the folder **LambdaExercises** using the syntax **javac com.example.lambda.Main** which would

produce several class files in the **com\example\lambda** folder.

5. The class files produced by the compilation process are named:

    - **Main.class**
    - **Main$Customer.class**
    - **Main$CustomerProcessor.class**
    - **Main$ShowCustomer.class**

    The three inner classes **Customer**, **CustomerProcessor** and **ShowCustomer** get their own class files. Inner classes are compiler gimmicks to provide functionality that is not part of the Java Virtual Machine. An inner class is like a top-level class with a variable declaring the instance of the top level class. Let's verify this.

6. Open a Terminal Application (Mac) or Command Window (Windows) and change the working directory to the location of the LambdaExercises. Execute the following command:
    javap com.example.lambda.Main$Customer

7. Notice the **this$0** variable and its type. It is a final **com.example.lambda.Main** - which represents the top level class. Each of the other class files will have the same variable.

8. Execute the following in the same window:
    java com.example.lambda.Main

    Notice the amount of time it took to run the Java program.

9. Look at the code in either an editor or Eclipse, whichever you prefer.

10. We are going to illustrate using the **CustomerProcessor** as a Local class.

11. Move the **CustomerProcessor** class inside the **process** method at the first line of the method. It is now a **Local class**. If you look at the generated classes you will notice that the **Main$CustomerProcessor** is now **Main$1CustomerProcessor**. That is necessary because there can be more than one class definition of the same Local class in the same class, by declaring the same type in multiple methods or local scopes.

12. Run the class and notice the output didn't change. Now add a few more Customers to the initializer list as **new Customer("234","AZ"), new Customer("345","WA"),** etc. Run it again.

13. Next, we want to change the resolution of the timer to nanoseconds. Change the **System.currentTimeMillis** to **System.nanoTime**. Run it again. Now, instead of 4 we get something like 337436.

14. To simplify the program a little, we are now going to change the definition of the **CustomerProcessor** to an **anonymous class**. To do that change the **process** method to the code below:

```
public void process() {
        // create instance of anonymous inner class
        ShowCustomer show = new ShowCustomer() {
                @Override
                public void show(Customer t) {
                        System.out.println("Account "+ t.account+ " state
                                                                      "+t.state);
                }
        };
        String state = "TX";
        for (Customer customer : customers) {
                if (state.equals(customer.state) ) {
                show.show(customer);
                }
        }
}
```

15. Notice that there is no class defining the implementation method anymore. It is instead being defined by the use of an anonymous class with the method defined inline. If you look at the classes now defined in the output folder you will see that **Main$1CustomerProcessor** is now **Main$1**. It doesn't have a type so it is defined as the parent type with a **$1** to indicate it is the first inner class defined. If you look at it with **javap** you will see that it has the same bytecodes as the previous class - just the shorter type name.

16. Now we want to turn it into a **Lambda expression**. Change the **process** method to the code below:

```
public void process() {
        // create instance of lambda expression
        ShowCustomer show = (Customer t) -> System.out.println("Account
                                      "+ t.account+ " state "+t.state);
        String state = "TX";
        for (Customer customer : customers) {
                if (state.equals(customer.state) ) {
                show.show(customer);
                }
        }
}
```

17. What we have done is defined the method at the local level without the method signature. If you look at the output folder there is no longer a **Main$1** implementation class. See how much simpler the code is?

18. We have one more simplification to refactor the code. We can remove the local variable and use a **Lambda expression** inline. Change the **process** method to the

code below and add the **checkCustomerList** method:

```
public void checkCustomerList(List<Customer> customers,
                                 Predicate<Customer> predicate) {
       for(Customer customer:customers){
              if (predicate.test(customer)) {
                     System.out.println("Account "+ customer.account+ "
                                          state "+customer.state);
              }
       }
}

public void process() {
       // create instance of lambda expression
       checkCustomerList(customers, (n) -> "TX".equals(n.state));
       checkCustomerList(customers, (n) -> "WA".equals(n.state));
}
```

19. Run the refactored application. Notice how much cleaner the code is now! We now have a method to check our customer list against various conditions that can be separately maintainable using additional Lambda expressions.

Lab 2: Signing and Verifying a JAR

Overview

In this lab, you will generate a self-signing certificate so that you can sign a Java Archive (JAR). You will also sign and seal a JAR and verify that it was signed properly.

This lab can be done using an editor or an IDE to edit the file. The JAR signing and verifying must be done using the command line. To use the command line, you will need to know where the JDK is installed.

1. Open Eclipse or a command/terminal window. Create a **`Project`** named **`JarSigning`** as a **`Java Project`**. Or create a folder in the home directory named **`JarSigning`** to run the exercises in.

2. If using Eclipse, set the output folder to the same folder as the source.

3. Add a package named **`com.example.mypackage`** and **`com.example.mypackage.util`** to `the` **`JarSigning Java Resources: src`** folder.

4. Copy the file **`Main.java`** from the **`Setup\Exercises\JarSigning`** folder to the package **`com.example.mypackage`** just created. Copy the **`Utility.java`** from the **`Setup\Exercises\JarSigning`** folder to the **`com.example.mypackage.util`** package. These two classes are going to be signed in later steps.

5. Eclipse will automatically build the project and compile the code. If you look in the **`src`** folder of the workspace you will see the directories built and several class files. If you are using the command line you would run **javac** in the **`JarSigning`** folder using the syntax **`javac com.example.mypackage.Main com.example.mypackage.util.Utility`** which would produce the class files.

6. Open a Terminal Application (Mac) or Command Window (Windows) and change the working directory to the location of the **`JarSigning`** folder. Execute the following command:
   ```
   javap bin/com.example.mypackage.Main
   ```
   or if using Eclipse:
   ```
   javap src/com.example.mypackage.Main
   ```

7. To make things easier, we are going to set the **`JAVA_HOME`** variable to the Java install folder (if it is not already set).

   For Windows:

A. To check this, use `set JAVA_HOME.` If your JAVA_HOME is already set, it should say something like:
   `JAVA_HOME=C:\Program Files\Java\jdk1.8.0_66` or
   `JAVA_HOME=C:\Progra~1\Java\jdk1.8.0_66`
B. To set `JAVA_HOME` use:
   `set JAVA_HOME="C:\Progra~1\Java\jdk1.8.0_66"` (depending on where you JDK is installed).

For Mac:

A. Run `echo $JAVA_HOME`. If the variable is already set, it should say something like:
   `/Library/Java/JavaVirtualMachines/jdk1.8.0_66/Contents/Home`
C. To set JAVA_HOME, use
   `export JAVA_HOME=$(/usr/libexec/java_home -v 1.8)` or add that line to your `.bash_profile` file to make it persistent**.**

8. Before we can sign anything, we will need a certificate in our **keystore**. Use the following command (all on one line) to generate a public and private key pair and certificate in the keystore. (You can leave out **%JAVA_HOME%/bin/** on Macs.) There will be lots of questions. Answer them but have fun.
```
%JAVA_HOME%/bin/keytool -genkey -alias server-alias
-keyalg RSA -keypass changeit -storepass changeit
-keystore keystore.jks
```

9. Export the server certificate which we will use during the verification process.
```
%JAVA_HOME%/bin/keytool -export -alias server-alias
-storepass changeit -file server.cer -keystore
keystore.jks
```

10. Create a file in the project root named **Manifest.txt** and add the following content. We will use this to seal our jar in the next step.
```
Manifest-Version: 1.6
Main-Class: com.example.mypackage.Main
Sealed: true
```

11. Create the jar file with the following syntax. The `c` is to create the jar, `f` to say you are passing a filename, and `m` is to say you are providing the filename of a manifest file to use in that order. If using Eclipse again substitute **src** for **bin**.
```
%JAVA_HOME%/bin/jar cfm JarSigning.jar Manifest.txt
bin\com\example\mypackage\*.class
bin\com\example\mypackage\util\*.class
```

12. You can verify that the jar was built by opening it. In it you will find a `MANIFEST.MF` in the `META-INF` folder.

13. We are now prepared to sign the jar. Sign it using the following syntax.

For Windows:
```
%JAVA_HOME%\bin\jarsigner.exe -keystore keystore.jks
-signedjar SignedJar.jar JarSigning.jar server-alias
```

For Mac:
```
jarsigner -keystore keystore.jks -signedjar
SignedJar.jar JarSigning.jar server-alias
```

When asked for a Passphrase enter: **changeit**
There will be no feedback of whether or not it is correct.

14. First we are going to do a visual verification. Open the **SignedJar.jar** and look in the **META-INF** folder and notice that there are two new files. It can be opened with 7 Zip or WinZip or rename to **SignedJar.zip** and renamed back when done. **SERVER-A.RSA** and **SERVER-A.SF**. Open the **SERVER-A.SF** using any text editor and examine the content. There are digests for each file in the jar.

15. Now let's make sure the signing worked. Verify it using the following syntax.
    For Windows:
    ```
    %JAVA_HOME%\bin\jarsigner.exe -verbose -verify
    SignedJar.jar
    ```

    For Mac:
    ```
    jarsigner -verbose -verify SignedJar.jar
    ```

16. Output from the command will be:

```
s      339 Sun Dec 27 16:55:10 CST 2015 META-INF/MANIFEST.MF
       446 Sun Dec 27 16:55:10 CST 2015 META-INF/SERVER-A.SF
      1347 Sun Dec 27 16:55:10 CST 2015 META-INF/SERVER-A.RSA
         0 Sun Dec 27 16:47:22 CST 2015 META-INF/
sm     571 Sun Dec 27 16:47:16 CST 2015 com/example/mypackage/Main.class
sm     523 Sun Dec 27 16:47:16 CST 2015
com/example/mypackage/util/Utility.class

s = signature was verified
m = entry is listed in manifest
k = at least one certificate was found in keystore
i = at least one certificate was found in identity scope

jar verified.

Warning:
This jar contains entries whose certificate chain is not validated.
This jar contains entries whose signer certificate will expire within
six months.
This jar contains signatures that does not include a timestamp. Without
a timestamp, users may not be able to validate this jar after the
```

```
signer certificate's expiration date (2016-03-26) or after any future
revocation date.

Re-run with the -verbose and -certs options for more details.
```

17. We will get a verification message. Notice in the beginning of some lines there is an **s** and an **m**. Rerun the above command but add the **-keystore keystore.jks** and notice the addition of a **k** to the beginning of some of the lines. The legend tells you what that means. Without the keystore, there is no validation of the key against the keystore. Certificate may or not may be valid, but the signature is valid. Since the signature we signed with is in the keystore, when we verify against the keystore, the certificate is considered valid.

```
     454 Sun Jan 10 19:52:34 CST 2016 META-INF/SERVER-A.SF
    1373 Sun Jan 10 19:52:34 CST 2016 META-INF/SERVER-A.RSA
       0 Sun Jan 10 19:51:50 CST 2016 META-INF/
smk  571 Sun Jan 10 17:00:40 CST 2016 bin/com/example/mypackage/
Main.class
smk  523 Sun Jan 10 17:00:40 CST 2016 bin/com/example/mypackage/util/
Utility.class

  s = signature was verified
  m = entry is listed in manifest
  k = at least one certificate was found in keystore
  i = at least one certificate was found in identity scope

jar verified.

Warning:
This jar contains entries whose signer certificate will expire within
six months.
This jar contains signatures that does not include a timestamp. Without
a timestamp, users may not be able to validate this jar after the
signer certificate's expiration date (2016-04-09) or after any future
revocation date.

Re-run with the -verbose and -certs options for more details.
```

Lab 3: Working with Exceptions

Overview

In this lab, you will experiment with Exception Handling using JDK 7 and 8 syntax.

This lab can be done using an editor and the command line or an IDE.

1. Create a **Project** named **ExceptionHandling** as a **Java Project**. Or create a folder in the home directory named **ExceptionHandling** to run the exercises in.
   A. If using the IDE, set the output folder to the same folder as the source.

2. Add a package named **com.example.mypackage** to the **ExceptionHandling Java Resources: src** folder.

3. Copy the file **Main.java** and **ShowNumbers.java** from the **Setup\Exercises\ExceptionHandling** folder to the package **com.example.mypackage** just created. This will serve as the starting point for the exception handling example.

4. Copy the **numberslist.txt** file to a <u>new folder</u> in the **ExceptionHandling** project named **resources**. Add the resources to your **classpath** when running to access the file from the JVM.

5. There is minimal exception handling being done at this point. You will refactor the code to provide better exception handling. The program should not end just because it encountered invalid input!

6. Fix the errors so that the application runs. Run using program arguments below.
   Program Arguments:
   1 2 56 999 2334567 3900021 dkfr "" 218.32

7. Refactor the reused piece of code that handles the **NumberFormatException**.

8. The reading of the file is done totally "old school". Convert the IO to **try-with-resources** and see how much cleaner the code is.

9. Finally, create a custom **Exception** that will allow you to catch the **NumberFormatException** and throw either a **DataFormatException** or an **InvalidDataException**.

   A. You can decide which is appropriate based on the input **String**. Both should subclass an **InputDataException,** which should subclass the **java.lang.Exception**, since we want to force it to be handled.

B. **InputDataException** should have in instance variable to store the input deemed invalid.

C. The subclasses should set the message on the superclass using a specific message for each class.

10. Run the application again. Notice the improved exception handling.

Lab 4: Threads

Overview

In this lab, you will use threads to invert the colors of an image file and output the image to a new file. You will use the old style **Thread** and the new way of doing things with the **java.util.concurrent** package. You will use the **ThreadPool ExecutorService** and then use the **ForkJoin** to see how simple it is to create subtasks and allow Java to manage tasks for you.

We copy and invert the pixels 1000 times to get an average of the time it takes to convert the image. It changes the colors from teal and orange to red and blue. Your times will vary based on CPU but should be around 3 seconds. That averages out to about 3 milliseconds per pass for the entire image, which is 1024x1024 pixels.

1. The beginning source for the exercise is the **`ImageInverter`** class in the **`Setup/Exercises/Threads`** folder. You will also need the JPG file that is in the same folder. Create in a new Project. Create a package **`com.example.mandelbrot`** and put the Java file there. Put the JPG file in the main project folder.

2. Run the application. Refresh the project and you will see a new JPG file. Open the new one. View the original JPG. They colors should be different, but the image should be the same otherwise.

3. Our first challenge will be to use the **Thread** class by implementing **`Runnable`** on the class. That will mean you need to provide a **`public void run()`** method. It can call the **`compute()`** method which we will need later, so don't rename it.

   Change the **for** loop in **`invert()`**. Use the new instance of the **`ImageInverter`** as the **`Runnable`** object in a **`new Thread`** and then start the **`Thread`**. Break the work up into 4 pieces by doing a section per thread. Hard code it or use a for-loop. Start each **ImageInverter** at an offset ¼ of the way through and invert ¼ of the **int** array.

4. Run it and you should get the same output.

5. Next, instead of creating a **Thread** explicitly, use the **`ExecutorService`** to execute the computation on a new implicit **Thread** from the pool. You can get a **`fixedThreadPool`** from the **`Executor`** static implementation.

   Use the same instance of the **`ImageInverter`** for the **`Runnable`** reference needed to execute on the thread. You will probably want to wait for the threads to finish before looping on the outer loop. That can be done using the **`shutdown()`** method on the pool. It will wait until all the threads complete before shutting down.

6. A more robust threading strategy would break the work into more discreet units of work and make use of the `ForkJoin` subtasking ability of more recent versions of Java. To implement this, we need to change our class from implementing `Runnable` to subclassing either `RecursiveAction` or `RecursiveTask`. The choice is decided on whether the **compute** method needs to return a value, which means the task is an accumulating task or not. This task is not accumulating so it would be appropriate to use the `RecursiveAction`.

7. Change the `ImageInverter` to extend:
   `java.util.concurrent.RecursiveAction`.

   Change the `invert()` outer **for** loop:

   A. Remove the inner **for** loop and do not split the work. That will be done in the **compute** method.
   B. Instead of getting the `FixedThreadPool ExecutorService`, you will get the `ForkJoin ExecutorService`.
   C. Rather than using the `execute(Runnable)` method, we can now use the `invoke(ForkJoinTask)` method since the class implements the `ForkJoinTask` interface.

8. The `compute()` method also needs refactoring. It will need to see if the work left to do is decomposable.

   Add a threshold variable to define how much work is the smallest unit of work to do. 500 would be fine. It is arbitrary and, in this application, isn't based on how much computation is being done versus the overhead of thread management. Add an **if** check to see if there are still more pixels to process than the threshold. If so, decompose further. Otherwise, perform the calculation.

   To decompose further, split the work by creating a two new `ImageInverters` with half the remaining work for each and passing them to the `invokeAll(ForkJoinTask)` method. There is a method that takes a variable number of `ForkJoinTasks`. Otherwise, do the normal calculation for loop.

9. This will run in about the same time. Some things could be fixed but don't directly affect the output, as they would in the real world.
   - The destination buffer is being overwritten on each iteration.
   - All the threads are working with the same destination array.
   - Concurrency should be being enforced around the array, but is not.

   Optionally, if we want to add concurrency, we would need to add a synchronized block around the access to the destination array. If you choose to try that, reduce the

for loop from 1000 to 100 as there is significant overhead to use synchronization. To fix that delay, use a unique array per iteration of the **for** loop. Only save one to write the image from. Since there aren't any synchronization conflicts there is little delay in this example.

10. To finish up the topic, add a `LongAdder` to accumulate how many decompositions were executed. Increment it in the **compute** method, since it will be being executed in the context of a different thread. Add the results to the end of **run println**.

Lab 5: Networking

Overview

In this lab, you will create a server application that listens for a text string sent by a client. You will also create the client that sends the text string to the server. This will demonstrate how easy it is to create a networking application in Java.

1. Create a project for this exercise. Create two packages and two Main classes, one for the server and one for the client **`com.example.server`** and **`com.example.client`**.

2. In the server, create an instance of the **`ServerSocketChannel`** using the static **`open()`** method. Many frameworks use factory methods to create the objects they need. This is one example.

   Use **try-with-resources** to simplify the code.

3. Using the **`socket()`** method on the server socket channel returned by the open, bind the socket to a new **`InetSocketAddress`** on port **9999**.

4. Loop forever and accept a connection on the server socket channel which will give you a **`SocketChannel`** to communicate with the client over.

5. The client is going to send one text string. The server will need to read it in and log it.

   Use the factory method **`allocate()`** on the **`ByteBuffer`** to get a **`ByteBuffer`** to read the data into. Then invoke the **`read()`** method on the socket channel. Create a new String object using the **`byte[]`** returned from the **`ByteBuffer`** array method using **`"ASCII"`** for the encoding.

6. For the client, loop reading a string from the console. Read from the console using a **`BufferedReader`** from the **`System.in`**.

   You will need a **`SocketChannel`** to write to the server. You use **`open()`** on it the same as the server side did. On the **`SocketChannel`** use **`connect()`** and provide **localhost** as the host and port **9999** as the port.

   To write the data on the channel, you will use the **`ByteBuffer`**'s **`allocate()`** to get an instance, then put the string you got from the console in the **`ByteBuffer`**. To tell the **`ByteBuffer`** you are done adding content, use **`flip()`**. Write the **`ByteBuffer`** to the **`SocketChannel`** and close the channel.

7. That is all there is to it. Run the server first. It will wait for the client messages and log them to the console. Run the client next and type a string in the console to send. The server will wake up and print the string that you entered.

Lab 6: JDBC

Overview

In this lab, you will gain experience using JDBC features. You will create a database, populate the database and query the content of the database using several JDBC interfaces.

1. Create a project to work in called **JDBC**. Create a **Main.java** and a main method to execute.

2. JDBC support is built into Java, but there is no driver to connect to a database. To develop a JDBC application you will need to pick a database to connect to. JDBC is mostly transparent at the code level and can be connected to different databases without too much effort. Our lab is going to use **Derby**.

3. Using a **try-with-resources**, create a connection to the database. Use the static method on the DriverManager to get a connection to the url **"jdbc:derby:myDB;create=true"**.

   If you get stuck you can use the first JDBC snippet code from the **Setup/Exercises/JDBC snippets** file and add to your main method.

   Run the application. You will get an error that there is no suitable driver.

   The code only logs the message on the **SQLException**, but there are more details that can be provided by logging the **SQLState** and the **ErrorCode**. These can help to diagnose the error better.

4. Add the JAR **derby.jar** from the **Setup/Exercises/JDBC** folder. That will be where you installed the JDK on your machine. It isn't necessary for compiling so you can add it to the **classpath** on the **Run Configuration**.

   Run again after adding the derby.jar to the classpath. You will get a successful run.

5. JDBC supports both **DQL** and **DML** on a database. To use the new database we created, there needs to be data. Databases store content in tables.

   Create a table called **BOOK** with **five columns**:
   - An **id column of type identity**
   - Four columns of **whole strings** for **ISBN**, **NAME**, **DESCRIPTION** and **AUTHOR**.

   Derby can use a **varchar** with a length to hold text data. Snippet 2 can be used if you are unsure what syntax to use. Add that inside the **try-with-resources** block added with the first snippet.

Run the application. The **drop** will fail but **create** will succeed. Run a second time and the **drop** will succeed and **create** will succeed.

6. Insert a row into the table using the connection to get a prepared statement with position markers for the values of the four columns.

   Use **PreparedStatement setString** to provide values for the position markers.
   Name=The Cat in The Hat
   Author=Dr. Seuss
   ISBN= 978-0-7172-6059-1
   Description=Some description

   Snippet 1 and 2 used a regular **Statement** to perform the **execute()** which is passed a string of **DML** to execute. This snippet will use a **PreparedStatement**. PreparedStatements use a **?** to substitute for a value being passed. Snippet 3 can help with syntax if you are stuck.

   In this case we are going to provide 4 values. There will be four
   setString(int, String) invocations to provide the value to the row being inserted.

   Run the application. There will now be one row in the table!

   7. Get a regular **Statement** from the connection and use a **query** which queries the BOOK table **NAME** column. Get the results and log the result rows. It will run and echo the row that was added in the prior step.

   This code uses **executeQuery()** instead of **execute()** or **executeUpdate()**. That is because there needs to be a **ResultSet** to hold the query results. The **executeQuery()** method returns a **ResultSet** whereas the other methods return a **boolean** with true for success, or an exception is thrown for failure, or a false for not updating. If you get stuck, snippet 4 has a solution.

8. Update the database using the **FilteredRowSet**. You can get the **FilteredRowSet** from the **RowSetProvider**. Set the command to the desired SQL to retrieve all the columns but the ID. Set the URL.

   Perform an **execute** to get the **FRS** to synch up with the database.

   Move the current row to the insert row. Provide values for the columns using the appropriate update method.
   Name=Green Eggs and Ham
   Author=Dr. Seuss
   ISBN= B00ESF277M
   Description=Good book about green eggs and ham.

Perform the **insert**. Move the **FRS** to the current row and accept the changes. Use the same technique to enter a second book.
Name=Charlotte's Web
Author=E. B. White
ISBN= 9780064400558
Description=Good Book about a spider, a girl and a pig.

You can use the snippet 5 if you are stuck.

9. Create an **AuthorFilter** class for the **FilteredRowSet** to use as a filter, with three attributes:
   - **String[]** for authors
   - **String** to store the column name in the database that has the author
   - **int** for the column number in the query that has the author

A. It will implement the **java.sql.rowset.Predicate**. Implement the three **evaluate** methods.
   1. One takes a **RowSet** and will loop through the authors and check if the object in the column by name or number matches the author name. Use the **rs.getObject**.
   2. The second **evaluate** method takes an **object** and a **String**. It will look at the string and see if that matches the column name we stored. If so, compare the value against the authors and if a match is found return **true**.
   3. The third **evaluate** method takes an **object** and an **int**. It compares the column number against the column number instance variable. If they match, it will check all authors against the object that was passed and return **true** if one matches. They are used to determine if the filter criteria are met.

B. Provide two constructors. One takes a **String[]** for authors and the column name of the database. The second will take a **String[]** for authors and an **int** indicating which column in the query is the author column.

C. Copy the solution **AuthorFilter** from the **Setup/Exercises/JDBC Snippets** file to the folder the Main java file is in if you get stuck.

10. Run the application. Add "**E. B. White**" to the array of authors passed to the filter constructor and see if the filter allows "Charlotte's Web" to display in the list by running again.

Lab 7: Performance

Overview

In this lab, you will use two options for diagnosing performance-related characteristics of an application. **HPROF** produces a text file that can be used to investigate the application performance characteristics at the end of the execution. **HAT** can be used to view the results as well. You will also use **JVisualVM** to monitor the performance of the application more real time. You will see how coding changes can have a significant impact on performance.

1. The first tool that will be used is **`HPROF`**. Create a Java project named **`Performance`**. Create a package in **`src`** named **`com.example.mandelbrot`**. Copy the file **`ImageInvert.java`** and **`Pixel.java`** from the **`Setup/Exercises/Performance`** to the package just created. Copy the **`mandelbrot.jpg`** image file to the project folder. Review the imported code.

   Open a Command Window and change directory to the location of the project. Run **java** to run the application to verify that it runs without error. The program prompts for an **enter** key. We use this later to wait while we connect with the profiler.

   The output will be the run time and the number of decompositions. When performance testing, it is a good idea to repeat the operations several times to get a representative set of timings. No two iterations will be exact, but an average of the times will be helpful.

2. The example program reads in an image and blurs it in 4 different degrees based on the **width** parameter passed on the constructor of the **`InvertImage`** in the **invert** method. The first loop will do nothing after the blur output to the file. The second iteration of the loop will invert the image top to bottom. The third will flip the image left to right and the last will flip and invert. You can look at the output files to see that the content is as expected. The swap is easy to see, but the invert top to bottom not so much.

3. The first thing to notice is that the first of the 4 processes is under a second. Each additional pass on the data is taking over a second. So, the last one of the 4 is around 4 seconds. Why is it taking a second to flip or swap the image?

4. Change the **java** command by adding this parameter **`-agentlib:hprof`** which will invoke the JVM and tell the JVM to generate the **`HPROF`** data to a file **`java.hprof.txt`**. The execution is about 10 times longer.

5. Refresh the project. **F5** will usually do that or select **Refresh** from the project's context menu.

Open the **`java.hprof.txt`** file in a text editor. You can find the format of the file at the beginning and some help about what options to use when running with the **hprof** agent. Look for **`com.example.mandelbrot.Pixel`** and notice the count - if you even get a count. There are a lot of **`Pixel`** objects being created.

6. Now we want to know where the time is spent. Change the parameters for the **agentlib** from **`-agentlib:hprof`** to **`-agentlib:hprof=heap=sites, cpu=samples`** and run again. Open the file in a text editor again. Scroll down to the bottom or search for "CPU Samples". Notice that the 17th longest running method is **invert**. Nothing is really jumping out yet as the major performance impact.

7. This is a very slow and painful way to find out what is happening. One helpful thing about the file is that every method invocation is tracked, and every object instance. Lots of information but hard to get a high-level view of. A better way is to use the **JVisualVM**.

8. Change the parameters to **`-agentlib:hprof=format=b`** which will output the file in binary format so that it can be loaded by **JVisualVM**. Open **`JVisualVM`**. It is in the bin folder of the JDK. Once it loads, load the **`java.hprof`** from the project folder. **`Load`** is under the **`File`** menu. Navigate to the project folder and change the filter to **`*.hprof`**. Now we are ready to examine the heap and see what is using the memory.

9. Click on the **`Classes`** tab. Notice that the two most common classes are the **`int[]`** and the **`com.example.mandelbrot.Pixel`**. The reason there are so many **`int[]`** is that they are used to communicate between the threads to coordinate work between **`ForkJoinWorkers`**. While this is interesting and useful to see if there are memory leaks, it doesn't tell us where the application is spending time. We need to get some CPU information.

10. Increase iteration count to at least 4 in **`ImageInvert.java`** line 32. Depending on the speed of your computer you might want to increase it to 10 or even higher so you can take your time in **JVisualVM** during the next steps.

11. Rather than generate the profile data as a separate process to an output file, let's get it dynamically.

    Run the application again, but <u>don't hit enter when prompted</u>. Instead in **`JVisualVM`** there will be a new Java process with the title of the **`com.example.mandelbrot.ImageInvert`**. Open it. Select the **`Monitor`** tab. Notice the live update of the **`CPU, Heap, MetaSpace, Threads,`** and **`Classes`**.

    Select the **`Sampler`** tab. Click the **`Memory`** button and in the window that is waiting for the prompt, hit enter. Notice the live update of the memory. Click on the

Monitor tab. This displays real-time information extracted from the running JVM. Eventually the **com.example.mandelbrot.Pixel** will move to the second spot on the memory space. After the program sits for a few minutes it will drop from the list because there are no memory leaks. All the Pixels allocated get garbage collected.

12. Run again and connect to the new process and this time watch the **CPU Sampler**. Eventually the **com.example.mandelbrot.Pixel.invert** and **com.example.mandelbrot.Pixel.swap** methods will show up as the fourth and fifth longest running methods. Switch to the **Threads** tab and notice that there are a number of **ForkJoinWorker** threads. Click on one of them. The status will show **Parked**. All the **ForkJoinWorker** threads will be parked. Parked is waiting but not eligible for scheduling by the thread scheduler until they are needed.

13. Some anti-virus programs, like Norton or Trend Micro block the use of the profiler connection. If it doesn't, you can select the **Profiler** tab. Use the **CPU** button and attempt a connection to the running JVM. If the connection is rejected, then you can't profile CPU or Memory.

14. Feel free to investigate the tool some more, but for now let's address the issue with **invert** and **swap**.

    Go to Eclipse and project and open the **InvertImage** and find the **invert** and **swap** methods. They were not implemented wrong or even badly, but they are not as performant as desired. There is always a tradeoff between good OO and performance. Since we are doing image manipulation there are several ways to make this faster. The image buffer is a large **ArrayList** of a million pixels. Each line is 1024 wide and 1024 high.

15. Instead of reading each pixel in and manipulating it with an **ArrayList**, it would be much faster to work with it as an **int[]**. Change the **invert** and **swap** method to use an **int[]** for the line instead of using an **ArrayList<Pixel>** for the line. Remove all references in the class to **Pixel**.

    When you get a clean run, run the performance test again. Notice that all the long running passes are gone. The methods **invert** and **swap** no longer show in the list of long methods. They are no longer close to half a second. They are virtually instantaneous. Now the passes are fairly close in performance. It seems that the use of the **Pixel** really slowed down the application. It's important to consider how much a design decision will affect performance. Ease of code maintenance vs. performance is always an issue.

Lab 8: Collections

Overview

In this lab, you will learn about ways to prototype collection usage to find the best performing collection for its intended usage. All collections have a set of performance characteristics that are dependent on the data, how the data will be inserted and how the data will be used. You will load data into a collection and use it in various ways. In addition, the time to perform the operations is going to be logged. You will understand how to select the right collection for some common processes.

The application that is provided has everything needed but the collections. You will provide all the behavior for managing creating the collections, and inserting and removing elements from the collections.

1. Create a project called **Collections**. Create a package called **com.example**. Copy the **Record.java** and the **Main.java** from the **Setup/Exercises/Collections** folder into the package you just created. Copy the **csv** file into a **resources** folder under the project.

2. In the **Main.java** the provided code reads from a **csv** file and splits the data into a set of column data. The goal of this step is to store the values as a **Record** object in an **ArrayList**.

   First, you need the **ArrayList**. You will do that by editing the **readToArrayList()** to create a **records ArrayList**.
   To run the program at this point, uncomment the lines with `// <-- for Step 2`.

3. Create a **Record** object and pass it the appropriate values from the **String[]**and add it to the **ArrayList**. (Look at the Record class to see how it expects its variables).

4. Add a line to sort the collection before printing the time taken. To sort using the Collections sort method you will need a **Comparator**. You can use an anonymous class or a Lambda expression. Dates can be compared to each other so use the flight date as the sort criteria. Log the time it took to read the file.

5. Now try to find three specific records, but not by index.
   In the CVS file, they are the records in rows 123456, 234567 and 456789. You will need to provide an implementation for the **findRecord** method as it always returns **null**. One recommendation is a for loop or a more modern implementation using a lamba expression.
   1. **SBA to SFO on 1/29/15 flight 5640 departure was 1229**
   2. **SEA to SMF on1/09/15 flight 742 departure time was 0705**
   3. **IAD to RIC on 1/5/15 flight 3945 departure time was 1750**

See the line that ends on `// <-- For Step 6` and uncomment it.

You can use this template for finding the records:

```
LocalDateTime startFind = LocalDateTime.now();
Record foundRecord = findRecord(...);
if (foundRecord != null) {
      foundRecords.add(foundRecord);
      logger.info("Found record ");
}
foundRecord = findRecord(...);
if (foundRecord != null) {
      foundRecords.add(foundRecord);
      logger.info("Found record ");
}
```

6. Log the time it took to find the three records.

7. Remove them from the collection and time the remove.
   (See the line that ends on `// <-- For Step 7` and uncomment it for the logger.)

8. This shows how long it takes to insert rows into an **ArrayList**, find specific elements in the list and then remove the elements. For timing, we wanted a fairly large list; 400K is a pretty big list. **ArrayList** is pretty memory efficient, and requires little overhead, but to find elements and remove elements requires scanning the entire set.

9. For the next part of the lab we move on to the **readToHashMap().** (You can comment out or delete the part where we sort the collection, as we will not need it further.)

   We are going to do the same thing we just did but use a **ConcurrentSkipListMap** instead. This is a really good collection for finding and removing elements from the list. It is not so good from a memory perspective. It has both a forward and backward reference which improves navigability but requires additional memory. Using the Map will require that a key be derived from the data that is unique. We provided an implementation for you.

10. Create a **ConcurrentSkipListMap** of **Records.**

    For each row create a key (use the **buildKey(String[] columns)**) and put the row with the key in the **ConcurrentSkipListMap.**

11. Now find the same three records from step 5. Instead of the loop, you can use the Map **get** method, passing a key. Log the time to find the records. It will be a significantly faster operation than the loop that was done earlier.

    Tip: The **buildKey(String flightNumber, String origin, String destination, String flightDate, String departureTime, String cancelled)** method will give us the key belonging to the record we want to find.

1. **SBA to SFO on 1/29/15 flight 5640 departure was 1229**
2. **SEA to SMF on1/09/15 flight 742 departure time was 0705**
3. **IAD to RIC on 1/5/15 flight 3945 departure time was 1750**

12. Remove the found objects from the map. Note that map elements are removed by their key not the value. As before, log the time it took to delete the elements. Notice that the time is as close to zero as can be measured. This is a really fast collection.

13. Optional: Show some statistics about the data using lambda expressions:
    - Flights on Monday – Hint: Filter on attribute and get the size of result list
    - Flights on the first of the month – same as above but different attribute
    - Aggregate flights by days of the month – hint **Collectors** methods **groupingBy** and **counting.**

14. Challenge: You will notice that there is a difference in the records read and the results count with **ConcurrentSkipListMap**. This is because there are some duplicate keys generated. How would you catch and handle duplicates so that all records are put in the Map?

Lab 9: Internationalization

Overview

In this lab, you will do learn how easy it is to support internationalization of applications. You will use **ResourceBundles** to manage the international content of strings. You will also use **Locales** to provide content for numbers, dates and times that are internationalized.

1. Create a Main class with a main method in a project called `Internationalization` and a package called `com.example.i18n.`

2. In the main method print `"The current date is: " +` the current date to the console. The date can be retrieved using `LocalDate.now()`.

3. Add another line to print `"The current time is: " +` the current time to the console. The current time can be retrieved using `LocalTime.now()`.

4. Add another line to print `"The amount is: " +` an amount that is greater than 10 million with at least two decimal places. An example would be 12345678.99, which the solution uses.

5. Run the application. There should be three lines of text resembling:
```
The current date is: 2016-01-11
The current time is: 18:29:11.272
The amount is: 1.234567899E7
```

6. Instead of outputting a hard-coded string, create a file named `messages.properties` in the `src` folder and add three name value pairs. Use the names `DATE_MESSAGE`, `TIME_MESSAGE` and `BALANCE_MESSAGE.` For the values use the text you output to the console ("The current date is", etc.).

   Add those three as constants to the `Main` class as **Strings** with the same value as their names.

7. To lookup the values added to the properties file, use the `ResourceBundle.getBundle()` method using the name of the properties file (without the **.properties** part) and the current **locale**, which can be retrieve using a `Locale` factory method. Use the `getString` method on the resource bundle to get the value that should be output as the message part of the display to the console.

8. To format the date, Java 7 added a `DateTimeFormatter` set of classes that can be retrieved as constants from the class. `ISO_LOCAL_DATE` and `ISO_LOCAL_TIME` can be used for the date and time respectively. `DateTimeFormatter` objects have `ofLocalizedDate` and `ofLocalizedTime` methods to localize strings with a

specific format like **`FormatStyle.MEDIUM`**. They return a **`DateTimeFormatter`** instance that can be used in conjunction with their **`withLocale()`** to internationalize to a specific locale. Give that a try for both the date and the time.

9. Run the application and you should get a slightly different output than before. The time doesn't have milliseconds and the date has a month day format.

10. To internationalize the amount value being displayed use the **`NumberFormat.getCurrencyInstance()`** and run again.

11. Now to really have some fun, create four different locales. One for **en_US**, one for **en_UK**, one for **es_ES**, and one for **de_DE** and run the program for each of the locales and see if there are any differences.

12. The strings didn't change. Now let's internationalize the strings. Copy the **`messages.properties`** to **`messages_es.properties`** and **`messages_de.properties`**. Use these strings as internationalized strings. Then run the application and the strings will now change based on the language.

```
ES:
DATE_MESSAGE=la fecha actual es
TIME_MESSAGE=la hora actual es
BALANCE_MESSAGE=la cantidad es

DE:
DATE_MESSAGE=Das aktuelle Datum ist
TIME_MESSAGE=Das aktuelle Zeit ist
BALANCE_MESSAGE=Das betrag ist
```

Lab 10: Secure a Web Application

Overview

In this exercise, you will secure a web application using both declarative security and programmatic security. You will implement the declarative security using the **web.xml** file of the web application. You will test this functionality using the Apache Tomcat server version 7. We provide the sample **index.html** and Eclipse will be used to prepare the rest. The second phase of securing the application will be done in conjunction with the SSL lecture.

PART ONE

1. In Eclipse, create a new Dynamic Web Project named **SecureApp** and on the Web Module page, check the box to create the **web.xml** deployment descriptor.

2. Open the web.xml file and add a **security-constraint** section.

   A. Within the **security-constraint** section, add a **web-resource-collection** and an **auth-constraint** section.
   B. Within the **web-resource-collection,** add a **web-resource-name** and a **url-pattern**.
   C. Set the **web-resource-name** to **static** and the **url-pattern** to **/*** . The HTML pages will be restricted to the **role-name** in the **auth-constraint** which you can set to **user**.

3. Add another section below the **security-constraint** of **login-config** in the same **web.xml**. Add **auth-method** tag with a content of **BASIC** and a **realm-name** tag with a content of **Test Realm**.

   Below the **login-config** add another section **security-role** and insert a tag **role-name** and put **user** as the content.

4. The sections in the web.xml will allow access to the index.html page but not to the servlets we are going to define later.

5. To test this, add a server. Open Window->Preferences. Under **Server** find the **Runtime Environments** and **Add** a **Tomcat 7.0 Server**. Click the **Next** button and set the location to **C:\Program Files\apache-tomcat-7.0.64** for Windows or **/usr/local/apache-tomcat-7.0.64** for Mac. Click **Finish**.

   Switch to the **Servers** view. Right-click and select **New -> Server** to add a new server. Look under **Apache** and find the **Tomcat 7.0 Server**. Click **Finish**.

6. In the left tree of the main screen under **Servers** you will see **Tomcat 7.0 Server** and there will be some xml files under that. Open the **tomcat-users.xml** and add a **role** with a **role-name** of **user**. Add **user** to the roles of all the users and **tomcat** to the **tomcat user**. Save the file and close it.

7. Import the provided **Setup/Exercises/SecureApp index.html** file into the Web Content folder. It contains the following markup:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Security test page</title>
</head>
<body>
Click the link below to test permissions
<br><a href="HelloServlet">Hello - tomcat allowed</a>
<br><a href="Hello2Servlet">Hello 2 - role1 allowed</a>
</body>
</html>
```

8. Right click on the **index.html** and **Run As -> Run on Server**.

9. You will be prompted to log in. Use **tomcat** as the **user** name and **password**. The links will not work, but the page will load.

10. Create two servlets, **HelloServlet** and **Hello2Servlet** in a package **com.example**. Eclipse has a wizard the creates the servlet using a **HTTPServlet** annotation. The servlets will need an **@ServletSecurity** annotation.

    ```
    @ServletSecurity(rolesAllowed = {"tomcat"}))
    ```

    A. Each security constraint will use an **@HTTPConstraint** with a property of **rolesAllowed** with a value of an array of Strings.
    B. Make **HelloServlet** allow "**user**" and **HelloServlet2** allow "**role1**". Add code to display the **request UserPrincipal** name so that we can verify which user is accessing the page.
    C. Run index.html to test the security.

    If you get Errors like **`The import javax.servlet cannot be resolved,`** go to Properties → Project Facets. Click on Runtimes and select the Apache Tomcat v7.0 server

    STOP HERE. Part II of this exercise will not be done until after the **SSL** unit.

PART TWO

1. To enforce transport security or SSL/TLS support, Tomcat needs to be told to enable it in addition to the application requiring it. There should be a **keystore.jks** from the **JarSigning** lab in the **JarSigning** project. If not, the **Setup** folder has a **keystore.jks** that can be used.

   Copy the **keystore.jks** to the folder with the **tomcat-users.xml** file.

2. Open the **server.xml** file and search for a **Connector** with port **8443**. It will be commented out. Uncomment it and add the following attributes. You can use Eclipse code assist.
   keystoreFile="**c:/setup/ Exercises /SecureApp/keystore.jks”** (adjust to your installation path)
   protocol="**org.apache.coyote.http11.Http11Protocol” keyAlias**="**server-alias**" **keystorePass**="**changeit**" **SSLEnabled**=”**true” secure**=”**true” sslProtocol**=”**TLS”**

3. The server will need to be restarted. If everything is correct it will start. Otherwise correct the errors and try again. Typical errors are missing quotes, not matching quotes or the property name being misspelled.

   When the server starts run the index.html on the server again. This time in the web browser, change the http:// to **https://** and the port from 8080 to **8443** to use SSL instead.

4. The prior step did not force the use of SSL on the servlets or the static content. To force the servlets to be only accessible using SSL, add the property **transportGuarantee** to the **@HTTPConstraint** and set it to **TransportGuarantee .CONFIDENTIAL.** This will redirect to https if the client is not already using https.

Running the Solution

1. If you would like to run the solution project, here are the steps. After importing the project, you will need to add a server in the Server tab of the JEE perspective or Web perspective. Tomcat is already installed, simply point to the existing installed location which should be **C:\Program Files\apache-tomcat-7.0.64** or similar.

2. Change the project properties for Project Facets and Server to reference the server just added. Using the **server.xml** and the **tomcat-users.xml** from the Exercises/SecureApp folder as a model, change the setting for the tomcat server just added. The items to note are the **users** and **roles** for the tomcat-users.xml, and the **Connection** with **port=”8443”** for the server.xml.