# Dev Nexus 2019 – Becoming the first Java 11 Certified Developer

## Lab 1 - \_ and private interface methods

1. Create an interface with method signatures:
   1. **default int getRandomNumberTickets(int max)** – returns a random number of tickets between 1 and max inclusive
   2. **default int getNumberStrikes()** – returns a random number between 0 and 3 inclusive
2. Create a class named **Braves** that implements this interface. Call both methods and print the results.
3. Refactor the interface to use a private method with the following signature and have the two existing interface methods call it
   1. **private int getRandom(int min, int max)**
4. We’ve decided the Braves should get a home field advantages and only be allowed to get 0, 1 or 2 strikes. Overload **getNumberStrikes()** in **Braves** to implement this attempting to call **getRandom()**.
   1. Why doesn’t it work?
   2. How would you recommend fixing it?
5. Try renaming getRandomNumberTickets() to \_ and compile. Note which version of Java disallowed this.
6. Try renaming getRandomNumberTickets() to \_\_ (double underscore). Get the code to compile.
7. Bonus: When are private and private static interface methods useful in the real world?

## Lab 2 – effectively final try with resources and var

1. Create a class called **DevNexus**
2. Create a text file containing the predicated high temperature for the next three days here in Atlanta.
3. Using only features available in 1.7 (no lambdas/streams/anything we talked about in the module), write a method that:
   1. Reads the file into a **List<String>** using a method in **Files**.
   2. Creates variables for and calculate the min, max and average temperature from that list.
   3. Prints the three values
   4. Compile successfully using javac --release 7 DevNexus.java.
4. Copy your code into a class called **DevNexusRefactored**.
   1. Switch to use var in every place it is allowed
   2. Compile and run normally using Java 11
5. Copy your code into a class called **DevNexusStreams**.
   1. Rewrite your logic using Java 8 streams and using var within the lambdas
   2. Compile and run normally using Java 11
6. Bonus: Read <http://openjdk.java.net/projects/amber/LVTIstyle.html>. Which principles or guidelines do you think are most important?

## Lab 3 – Collections and Strings

1. Create a file named **speakers.txt** and fill it with at least three speakers you are excited to see this week in the format, “FirstName,LastName”
2. Using **Files.readAllBytes()**, read the file into a **String**.
3. Using lines(), find the speaker whose first name comes first alphabetically.
4. Using lines(), find the speaker whose last name comes first alphabetically.
5. Output “I love Dev Nexus” 10 times followed by the speaker with the longest name without using a loop.
6. Convert the **String** from the file into a **String[]** without using a loop.
7. Use **Arrays.asList()** to convert the ArrayList into an array. Attempt to add, remove and change an element of the generated list. Which operations succeed? Do any of the operations change the original ArrayList?
8. Use **List.of()** to create a list of your three favorite numbers. Attempt to add, remove and change an element of that list. Do any of them succeed?
9. Bonus: Can you come up with three examples that show why we need each of strip(), stripLeading() and stripTrailing()? What is a unique thing that each of these three methods does that trim() could not do?

## Lab 4 – Single file execution, Deprecation and JShell

1. Write the least code you need to run the command **java JavaDemo.java** and have print “Hello”
2. Write the least code you need to run the command **java JavaDemo.java 1 2 3** and have it print out the sum (6). (Do not use stream/lambdas)
3. Refactor to have a public static method and have the main method call that:
   1. **static long sum(String[] nums)**
4. Deprecate the method for removal as of Java 11.
5. Launch JShell
6. Write a one line command to calculate the sum using streams in JShell.
7. Update the one line command to calculate the sum of all odd numbers less than 10 using streams in JShell.
8. Create a new utility class called **MathUtils** and a method. Deprecate it, but not for removal.
   1. **static long sumOddNumbersLessThanTen(String[] nums)**
9. Have your main method in JavaDemo call the MathUtils version after the existing code.
10. Run **java MathUtils.java**. What error do you get?
11. Run **java JavaDemo.java 5 8 10**. What error do you get?
12. Compile and run these two classes “the long way.”
13. Bonus: Play with JShell to write something fun. What do you like most/least about the interface?

## Lab 5 – Streams

1. Create a class with a method that takes a **String** and returns a **Stream<String>** of 0 elements (if null) and 1 element (if not null)
2. Write a stream that counts down from 10 to 1. Use this stream to print a countdown followed by “blast off”
3. Write a stream that counts odd numbers starting with 1. Print out the 40th element of the stream.
4. Create a file with a few sentences. Read it into a stream and print all the words until you get to one that is six characters or more. (Do not print the six character word.)
5. Using the same file in the previous sentence, print just the first word that is six characters or more. (do not use dropWhile for this)
6. Now print the first word that is six characters or more followed by the next three words of any length.
7. Create an infinite stream of randomly generated positive integers between 1 and 1000. Print all of the even numbers before the first odd number.
8. Create a stream of ten randomly generated numbers from 1 to 100. Print all of the entries after (and including) the first number greater than 50.
9. Bonus: What use cases can you think of for takeWhile/dropWhile? (It’s ok to google for inspiration)

## Lab 6 – Jigsaw and Multi Release Jar

It’s ok to use an IDE or editor for this lab

1. Create a class com.devnexus.ListUtil
2. Create a method that returns a List<Integer> of 10 random numbers. Do not use any Java9+ APIs. Have this method print “Java 8” somewhere inside it.
3. Create another class with the same name for Java 9 that returns an immutable list. Have this method print “Java 9” somewhere inside it.
4. Create yet another class with the same name for Java 11. This time use var somewhere in your implementation. Have this method print “Java 11” somewhere inside it.
5. Package using a multi release jar setting the Manifest value and folders for Java 9 and 10.
6. Write a main method that calls ListUtil in a jar. (remember to use the –cp flag to reference the jar.) Run it as Java 8, 9 and 11. Compare the results.
7. Run java --list-modules. How many are there. Which do you think are most important?
8. Bonus: see if you can package your own module!