Lesson 6

* **Interfaces:** Abstract classes are great for providing some base implementations for concrete classes to inherit, but they do have their limitations.
  + The biggest limitation is that you can only inherit from one class
    - Airplan
    - Boat
    - SeaPlane – can’t inherit both
  + We want to use an abstract class when we know how we want some functionality to be implemented and want to leave the rest up to the subclass.
  + Another tool Java provides us with is an **Interface**, this solves the multiple inheritance problem because a class can implement multiple interfaces.
  + Another big difference between an abstract class and an interface is that an interface does not define any method implementations, it only defines the method signatures. You use this when you know what the class implementing the interface will do, but want to leave the actual implementation 100 percent up to them.
  + An interface is a contract. When a class implements an interface, it HAS to create concrete implementations of each method defined in the interface.
  + Let’s look at an example and create a Logger interface and a couple classes that actually implement it.
    - Logger will have the following methods: info, warning, error, fatal
    - ConsoleLogger logs to the console
    - FileLogger logs to a file
    - <http://www.baeldung.com/java-write-to-file>
  + Because an interface is a contract, you know that any class implementing a specific interface must have the methods defined in the interface. That means that if you have a method that takes a type of interface, you can pass any class that implements the interface in as an argument.
    - Just like in the example above with the ConsoleLogger and the FileLogger. Both can go in place of the Logger because we know the methods will exist there.
  + A very common and important rule in the industry is to always code to an interface. This means you create an interface first, then the implementing class.
  + Interfaces we are already familiar with from previous lessons are Collection, Map, Set, and List.
    - ArrayList, Vector, and LinkedList are all classes that implement the List interfact.
    - This means that each of these classes can be used anywhere a List is required.
* Sometimes code doesn’t work the way we think it would, and it can be very difficult to find out where the error is coming from. When this happens, we have to troubleshoot.
  + Initial troubleshooting includes looking through the code and seeing if we can spot the issue ourselves.
  + Then, we might add a few System.out.printlns to check out what the code is doing.
  + If these don’t work we can do something called debugging (we can even skip the sysouts and go straight to debugging for efficiency).
  + Debugging allows us to watch our code execute one line at a time and see all the variable’s values as they change.
  + To start debugging
    - Create a break point – this is a point that execution will pause at when debugging and wait for you to tell it to continue
    - Click the debug button instead of run (select yes when Eclipse prompts you to open the debug perspective)
    - The code will stop executing at your breakpoint and you can view the values of all the variables.
    - You can then continue execution or step through it one line at a time
      * Step over
      * Step into
      * Step out
* As applications become more complex, it becomes difficult to see the impact one change may have on the rest of the code. You may tweak something to make a use case work and unknowingly break 3 other areas of the code that were dependent on the code you tweaked.
  + There is an industry standard practice to help mitigate this issue known as **Unit Testing**.
  + Unit tests are code that run our application code to make sure the output is as expected.
    - For example, if we write a method that multiplies two ints and returns the product, we can write Unit Tests that call the method for us, passing in 2 and 5, and make sure the desired output of 10 is what we get.
    - Then, if someone changes the functionality of this method, and it no longer does what it was supposed to, the unit test will fail and point that programmer to what they broke.
    - This is super valuable, which is why it is an industry standard practice.
    - Writing unit tests can usually take just as long, if not longer than, writing the actual application code. So, it is expensive, but worth it.
  + JUnit is the testing framework we will use to write our Unit Tests.
  + Use the @Test annotation to classify something as a test
  + Write tests for all of your methods, not just happy path, but edge cases as well
  + Call your methods and assert (assertTrue or assertEquals for now) that the return is what is expected.
  + Notice you cannot import from the default package, it is best practice to NEVER use a default, unnamed package.
  + Run as Junit and see if the pass. Change them to see what happens if they fail.
  + These are the basic principles of unit testing, but they can get much more complex depending on the application and how you write code. Writing clean code will help make writing unit tests easier.
  + Another practice that the industry seeks after (though it is not truly implemented in many places because of it’s cost) is Test Driven Development, or TDD. This means writing all the unit tests first, and then writing the code to make the unit tests pass. This way, you define what you know the code should do first, then write the code within those parameters.