CPSC2151 – Lab 4

Generics, Interfaces, Interface Specifications, and default methods

Due: March 1st

This lab is to be done **individually.**

Introduction:

In this lab, you'll be given a video to watch about interface specifications and a set of starter code that contains the following files:

- IStack – A generic stack interface file. All the contracts are written, but some of the methods have no implementation.
- GenericStack – An implementation of the IStack interface. Utilizes an ArrayList data structure.
- StackDriver – A main file for you to use so you can test your implementation.

My recommendation is to watch the recording first. Once you've watched the recording, download the starter code and open it up in IntelliJ. Before you start completing the lab, read the passages on Interface specifications and primary vs secondary methods. The passage on Interface specifications should be a review of the material from the recording, and the passage on primary vs secondary methods should be a review of Wednesday's (2-21) lecture.

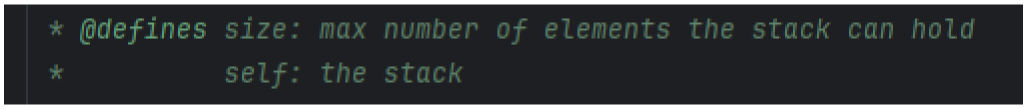
The only file you'll be working on is IStack. You'll need to read the interface specification and the method contracts, so you know what functions to add and how to go about implementing them.

Interface Specifications:

An "Interface Specification" is effectively the "class contract" of an interface file. Remember, interfaces are not classes, and behave very differently from the classes we've become familiar with. As such, we write their "class contract", also known as their interface specification, differently too. An interface specification comes with the typical description and 3 new annotations:

- @defines – Used to "hint at" private data contained within a class.
- @constraints – the "invariant" of the interface. This "invariant" would be true for all possible implementation of the interface, "always".
- @initialization_ensures – what is *going to be true* about any class that implements the interface ***after*** the constructor is called.

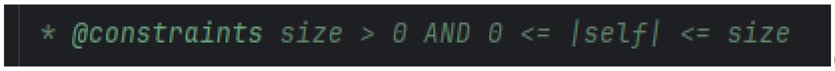
Remember the principles of information hiding, and the role that an interface plays in it. The interface is what we want the client to look at, since it is what allows us to hide the most amount of unnecessary implementation details. The GenericStack class contains 2 variables of data: an ArrayList called "elements" which is the stack's backend data structure, and a private constant int called "capacity" that represents the max number of elements of the stack. Instead of referring directly to these two bits of data, we can use @defines to *hint at them* instead. You'll see in the starter code this line:



```
* @defines size: max number of elements the stack can hold
*          self: the stack
```

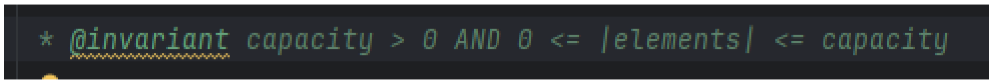
This is our @defines, and *hints at* the fact that any implementation is going to have "some kind private data used to represent the max number of elements". This means, instead of referring directly to the private data, capacity and elements, we can instead refer to this "max number of elements" as "size", and the data structure that makes up the stack as "self". This concept is known as **abstraction** and is one of the most important components of this course. By being **abstract,** which, in this context, just means being vague in our documentation, we aren't limited to any one implementation. In the case of size, this isn't a big deal, since size only makes sense as an int. However, by referring to my stack's data structure as "self", we aren't locking ourselves into GenericStack's implementation. "Self" can either refer to GenericStack's ArrayList (called elements), or if I had a second implementation of my stack that uses a LinkedList backend, it can refer to that instead. This allows me to only write my documentation once. Considering none of us want to write the documentation the first time, this is why abstraction is so important.

My other new annotations, @contraints and @initialization_ensures are a bit more self-explanatory. My @contraints is the "invariant" for all possible implementations. It is what must "always" (and remember what quote "always" end-quote means) be true about the state of any implementing object, regardless of what implementation I use. Often, my @constraints are the same as any implementing classes @invariant except the @constraint still can't refer to private data, so it uses the data we hinted at in the @defines instead. @initializtion_ensures is effectively an abstract way of writing the postcondition for all possible implementations' constructors. Look at the similarities:



```
* @constraints size > 0 AND 0 <= |self| <= size
```

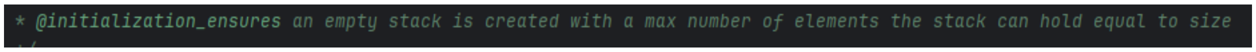
(from IStack)



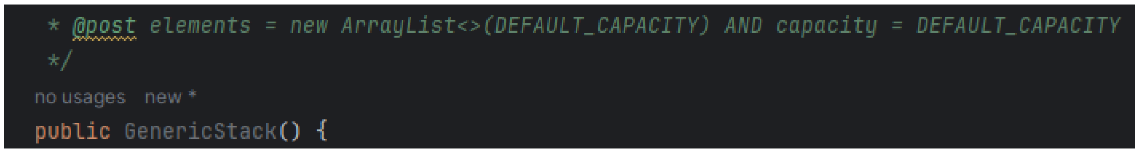
```
* @invariant capacity > 0 AND 0 <= |elements| <= capacity
```

(from GenericStack)

And



```
* @initialization_ensures an empty stack is created with a max number of elements the stack can hold equal to size
```



```
 * @post elements = new ArrayList<>(DEFAULT_CAPACITY) AND capacity = DEFAULT_CAPACITY
 */
no usages   new *
public GenericStack() {
```

Primary vs Secondary Methods:

There are two different ways we can classify the methods we write in an interface file: Primary and Secondary. If you're familiar with abstract classes from a previous course, then what we're working with is the interface equivalent of abstract methods (primary) and non-abstract methods (secondary). The difference between the two is as follows:

- **Primary methods**

    - Contain no implementation in the interface files. Implementation is cone in the classes that implement the interface.
    - There's no keyword to "set a method as primary."
    - Primary methods are used to access private data contained within the class files.
- **Secondary methods**
    - Are implemented as **default** methods in the interface.
    - If the implementation of a **default** method is the same across implementations (class files), then my classes will contain no override/implementation of the default method.
    - Because they're implemented in the interface, secondary methods **cannot access private data on their own,** and must use the primary methods to do so.

IStack contains 6 methods, of which, two of them are primary and four of them are secondary. My primary methods are **getElements()** and **getCapacity(),** while my secondary methods are **push(T), pop(), peek(),** and **getSize()**. Take a look at the contracts for getElements and getCapacity. What do we notice about these methods when we read their contracts? ***They're accessors!*** For both methods, their <u>*only*</u> job is to access the private data structure and max size of an implementing class of IStack. If we decided we wanted to make these methods secondary, which would mean they have to be implemented in the interface, restricting their ability to access private data of an implementing class, what else would they do? Nothing…

Let's look at the secondary methods. How do push and pop do their jobs, which is to add and remove things from the stack, which is private data, if they can't access the private data themselves? ***They call the methods that can!*** Instead of Push calling the GenericStack.elements ArrayList directly so it can call elements.add(), we can use our primary methods whose job it is to access the private data for us. The getElements() and getCapacity() methods can be called to access the private data, which in this case is the elements ArrayList and the capacity int. Once we have access to that private data within the push method, we just perform the typical operations needed to add something to my stack.

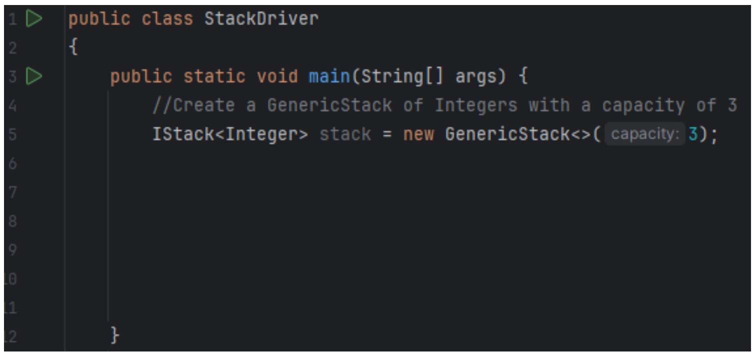
So long as my secondary methods (push, pop, peek, and getSize) never refer to private data directly and only access private data by calling the primary functions, it means that we're keeping these secondary methods **abstract** by allowing them to work for multiple implementations of the same interface. This is the power of **abstraction.** If I had 10 different classes that all implemented the IStack interface (where one uses a linked list backend, one uses an ArrayList, one uses an array, etc.) I only have to write the secondary methods **once.** Once I've written the secondary methods as default functions in the interface, Java will simply *default* back to the implementation in the interface if an implementing class doesn't provide an override for the default method of their own. Push would work the *exact* same way regardless of if the backend of my stack is an ArrayList or a LinkedList. So, instead of writing the implementation twice, let's write it once in the interface as a default method and let Java *default* to using that default implementation. We will (or have, depending on when you start this lab) talk about primary vs secondary methods a lot more in class, so for now, let's go ahead an implement the default methods within IStack.

TODO:

Using what you learned about interface specifications and primary vs secondary methods, your job is to implement the secondary methods of IStack.

The only 4 secondary methods we have are Push, Pop, Peek, and getSize. Remember, the goal is to implement them as **default methods.** This means that they cannot refer directly to the private data themselves and must rely on the primary methods to access the private data for them.

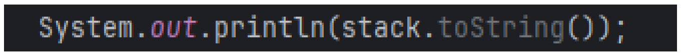
GenericStack is implemented for you, which means all you need to do is implement a driver so that you can test your code. Look at the code provided in StackDriver.java:



```
1   public class StackDriver
2   {
3       public static void main(String[] args) {
4           //Create a GenericStack of Integers with a capacity of 3
5           IStack<Integer> stack = new GenericStack<>( capacity: 3);
6
7
8
9
0
1
2       }
```

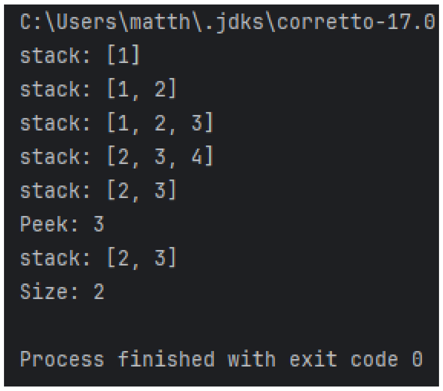
You should recognize what is happening on line five from Monday's (2-19) lecture. This is "Coding to the interface" so I am going to make my **declared type an IStack** and my **dynamic type a GenericStack**. If my default methods were done correctly, I should be able to call my secondary functions of push, pop, peek, and getSize on stack, despite it being an IStack (interface) type and GenericStack having no implementation of its own for those 4 methods. Writing drivers to test code is what we call **"Primitive testing"** and is something we'll talk more about in the testing unit after we finish interfaces.

The only thing else you need to do aside from implementing the 4 default secondary methods in IStack is to create a driver so you can perform some primitive testing on your implementation. What does this entail? Just call your secondary methods on stack, and then call



```
System.out.println(stack.toString());
```

after each secondary method call. By doing this, you should be able to produce an output like this:



```
C:\Users\matth\.jdks\corretto-17.0
stack: [1]
stack: [1, 2]
stack: [1, 2, 3]
stack: [2, 3, 4]
stack: [2, 3]
Peek: 3
stack: [2, 3]
Size: 2

Process finished with exit code 0
```

In this output, you see that I called stack.push and immediately called System.out.println right after three different times to make sure that each call to push worked as expected and added 1, 2, and 3 to my stack. Then, I do the same thing once more, which should put the stack over capacity, causing it to remove the 1 (as the longest-standing item in the stack) so there's room for 4. I then call pop and System.out.println to make sure that 4 gets removed properly. I do the same with peek (calling System.out.println to make sure it just returns 3 and doesn't remove it) and then I check that the size is 2 using getSize.

This is primitive testing in a nutshell. When we get to the testing unit, we'll talk about how this isn't how we properly test code. However, **it is at a *minimum* what you should be doing to any code you write for this class before you submit it**.

Once you've done your primitive testing, take a screenshot of the output so it can be submitted.

Submission:

You need to submit a **zip** file that contains 3 things:

- IStack.java
- StackDriver.java
- The screenshot you took of the output of your driver running (as a JPG or PNG)

Failing to submit anything other than the three files listed above (nothing more, nothing less) will at a minimum net you a 50. **If your submission is missing any of the 3 above files, it's a zero**.

I don't care what the name of the screenshot is (be an adult) and I don't care what the name of the zip file is either (once again, be an adult).

Once you have your **zip** file ready, submit it to Gradescope.