Lecture 3: class activities

prof. dr. Irma Ravkic

Activity 3.1 Why inheritance and interfaces?

Assume you have a service that allows users to make accounts to which they must first login to. The user has to first create an account with a password (assuming an integer password), and with a preferred notification service which informs the user that something went wrong with their account. Assume that your service initially supports only e-mail notifications and that the user has a maximum of 3 tries for logging, which when reached blocks an account and notifies the user. We will do live coding in the class.

- Think which classes you would need.
- Think which class members, constructors, methods you would need. Think simple.
- Think about exceptions.
- Now that you have some solution think about what you would need to change if you add more notification options: via phone messaging, via Twitter, ...

You will have the final solution on Canvas. This is a very simplified code of what actually happens in 'real-life', and it serves to demonstrate the encapsulation and modularity.

Activity 3.2 More on Interfaces

Test what happens if a Java interface specifiies a particular method signature, and a class that implements the interface provides a different signature for that method? For example, suppose interface *SampleInterface* is defiined as:

```java
public interface SampleInterface {
      public int sampleMethod();
}
```

and the class SampleClass is

```java
public class SampleClass implements SampleInterface {

    public boolean sampleMethod()
    {
        return true;

    }

}
```

Activity 3.3 We implement stack live

We will together write code for implementing Stack with arrays. Create a project/package called Stack/stack, and add your interfaces/classes in the src folder there.

1. We will first see how we can create a naive Stack solution with having multiple Stacks holding different data types. Let us create stacks containing objects of *Double*, *String* and our custom *Complex* class. We will first create interfaces with the signature for *pop()* returning the type the Stack contains. For example, for the Stack containing Doubles, you will have:

   ```
       public Double pop();
   ```

   Now, after you created all of the interfaces/classes your boss tells you that the *pop()* method actually only removes an element, and doesn't actually return it. What changes will you need to make to accomodate this new specification? Does this seem redundant for you? Imagine you need to create a special stack for each data type out there.

   ```
       public Double pop();
   ```

2. Create a driver class to create one of the stacks. Push/pop several elements from it. Well it doesn't do much right? Our methods don't do anything so we need to implement all the stack methods.

3. You notice that you need something to put elements in, or remove elements from. In this exercise we will use arrays to accomplish that. But we will need some extra information/data members to accomplish that. How do you initializ this container?

4. Now that you have three different implementations of Stack let us do some analysis.

5. What happens to the objects when you use pop? Do you actually delete the objects you created?

6. We will write a smarter solution using generics. You can put it in another package called *genericstack*.

Activity 3.4 Exceptional situtations

You have the skeleton of your code for stacks now. However, does it always work? What if you want to remove an element from an empty stack? In this activity we will create some custom made exceptions to deal with these problems.

Activity 3.5 Implement stack with ArrayList

You should have your own Stack implementation with arrays (after the class you can also find it Canvas under *Files* > *Lecture3* folder). For this activity, write your own implementation of Stack but with Java's ArrayList instead of Java arrays. Call your class *ArrayListStack*.

Activity 3.6 Add more to your stack

Add the following methods to your stack implementation with ArrayList clas:

int size() — returns a count of how many items are currently on the stack. Do not add any instance variables to your Stack class in order to implement this method.

String toString() - override this method from Object class. It needs to print stack as it looks like (LIFO). If you did the following:

```
    myStack.push(Double.valueOf(1.5));
    myStack.push(Double.valueOf(1.2));
    myStack.push(Double.valueOf(0.5));
    System.out.println(myStack);
```

your program should print the following:

```
    0.5
    1.2
    1.5
```

Create a test driver for it to show that it work correctly. In order to practice your array related coding skills, code the method by accessing the internal variables of the Stack implementation, not by calling the previously defined public methods of the class. You don't have to create a new project, just update your solution from Activity 1.

Activity 3.7 Application - string reversal

Use your implementation of Stack to test it on the application of reversing a sequence of characters where each element of the stack is a character. You can create this driver class in the package where the Stack implementation is.