Programming Techniques, 2022-2023

Ludo Bruynseels, Stijn Langendries, Jeroen Wauters, Nianmei Zhou, Koen Pelsmaekers

Lab session 3 : Implement an interface

Goal

The goal of this lab session is to write two implementations of a given stack interface and use this implementation to create a bracket parser class. Along the way you will learn about a couple of new concepts:

- *Packages, which are a way to structure your code*
- *Exception handling*
- *How to write your own test code*

Exercise 1: Implement the stack interface

Start by creating a new IntelliJ project, downloading the file lab3.zip from Toledo and extracting it. The zip contains three folders: application, util and view. Copy each of these folders to your application's src-folder.

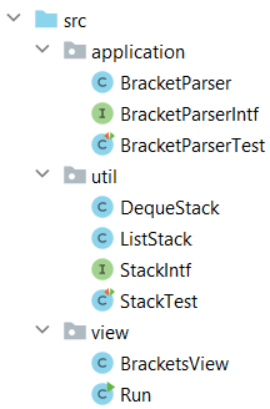
a. Packages

You will notice the folders also show up in IntelliJ. Source code in a Java-project can be given structure by organizing it in separate folders. The larger your project becomes, the more useful this is. There are two things to note:

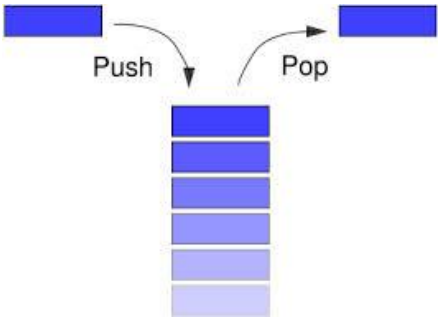
- Simply adding the folders is not enough. You also need to include a package statement in your .java-files. Check the first line of any of the included files.
- Classes from one packages are not visible by default in other packages. They need to be imported, just like classes from an external library would. Check the file `BracketParser` in the package `application`, which includes import statements for `StackIntf` and `DequeStack` from the package `util`.

The three packages in this project are:

- `application`: contains files related to the bracket parser
- `util`: contains files related to the stack structure, which will be used to implement the bracket parser. Stacks have many different applications though, which is why it makes sense to separate these files into a different package.
- `view`: contains an implementation for a GUI (graphical user interface) to test the bracket parser. You will not have to change any files in this package.



b. Stack basics



In this exercise you will implement a stack. This is a so-called LIFO ("last in, first out") data structure where elements are stored in the order they have been put on the stack, and can only be removed in the inverse order they were added. In other words, the last element that was added to the stack, will always be the first one to be removed.

A typical stack contains the following methods:

- `push()`: Puts a new element on the stack
- `pop()`: Removes the top element from the stack and returns it
- `peek()`: returns a reference to the top element of the stack, without removing it
- `isEmpty()`: returns whether the stack is empty
- `clear()`: removes all elements from the stack

If you check the interface StackInf.java, you will notice that it contains these operations. Note that this interface expects the `pop` and `peek` methods to return null if the stack is empty.

Note also the use of generics (the <T> notation), which you will recognize from working with ArrayLists. As you can see here, you can define your own generic classes, where a type T is added when defining a variable of this class type. In other words, in your code you can use T as the type of the elements on the stack.

c. Implement the interface

Your turn. Complete the class `DequeStack`, which provides an implementation for the stack interface[1].

To do this you will of course need to add an attribute to store the stack's data. You could use any type of data structure here, like an ArrayList. There is an easier solution though: Java already contains a predefined class which implements the behavior of a stack: ArrayDeque (in the package java.util). Check the documentation for this class, and note that the five methods described above exist within the class ArrayDeque. Import it and use it as a base around which to implement the interface's methods (hence the class name `DequeStack`). You won't have to write a lot of code.

d. Write appropriate test code

To test whether your implementation is correct, you will write your own test code. Check the class `StackTest`, where an implementation is already provided for testing the `push` and `pop` methods. Add code for testing the `peek`, `isEmpty` and `clear` methods.

The core of the test code are assert methods, which have the following structure (the third parameter is optional):

```
assertEquals( expected value, method call, error message )
```

Each of these methods executes the method call and checks if it returns the expected value. Note that this means you can only assert method calls that return something. If you want to test a method

[1] Remember you can use alt+enter to automatically add every method from the interface (without a useful implementation, of course).

with a void return type, you have to call it outside of an assert and check whether it had the intended behavior using other methods. Check the method `testPush()` for an example.

When writing test code, it is important to keep as many edge cases in mind as possible. For instance, when testing the peek method, make sure you don't forget to test peeking at an empty stack, ideally even twice: before any elements have been added, and after the last element has been popped.

The method `setUp` is executed before each of the test methods, which ensures that both stack attributes are freshly initialized so test methods do not influence each other. This is due to the `@BeforeEach` annotation.

Exercise 2: A second stack implementation

An interface can have multiple implementations. Let's demonstrate this by adding a new class `ListStack`, which also implements the `StackIntf` interface. This time however, use an ArrayList as the base data structure around which to build your stack.

When you're finished, update the test code to work with a `ListStack` instead of a `DequeStack`. You should only need to change two lines of code!

Exercise 3: Adding exception handling

a. A new package

Create a new package `utilExceptions`, and add the file `StackIntfWithExceptions` from Toledo. Copy your classes `DequeStack`, `ListStack` and `StackTest` from the `util` package.

Note that you don't even have to change the names of these classes. That means you now have two classes called `DequeStack`, `ListStack` and `StackTest` in your program! This is because packages define their own namespace. It is typically a very bad idea to have multiple classes with the same name in a project, as this can make your code very confusing to read – so you should avoid this in the future.

b. Exceptions

The new interface `StackIntfWithExceptions` differs from `StackIntf` in one way: the methods `pop` and `peek` no longer return null if the stack is empty, but they throw an exception instead. An exception is an object (it is created with a constructor) that contains information about something which has gone wrong in your program. It is "thrown" out of a method when the error occurs, and subsequently caught by a try-catch block. Throwing exceptions is done as follows:

```
throw new SomeTypeOfException();
```

while processing it is done like this:

```
try
{
    //some code that might throw an exception
}
catch( SomeTypeOfException e )
{
    //do something with the exception here. This can be as simple as printing
    //an error message, but might be much more elaborate.
}
```

Update the classes `DequeStack` and `ListStack` in the package `utilExceptions` to work with the new interface. An implementation of the test class can be found on Toledo.

Exercise 4: Bracket parser

Finally, we will use our stack implementation(s) in a practical application. You will create a bracket parser which checks whether a certain expression contains correctly structured brackets. This means that each type of opening bracket {, [and (needs a corresponding closing bracket, in the correct order. Characters other than brackets are ignored by the parser.

e.g. `({a + b}(c + d)[[(({{}}))]])` : **correct**

`(a { ) }` : **wrong**

Think about your algorithm first: how will you use a stack? Which data will you place on the stack? When will you pop, and what will you do with the element you retrieved? When do you need to check if the stack is empty? And so on.

You can use the view implementation to test any random expression, and there is also a test class to check whether your implementation is correct.

Tip: check out the HashMap data structure, is can be very useful in the implementation of this exercise.