THUY LINH PHAM

CS-2400-01-F24

PROJECT 2

DUE DATE: 10/21/2024

Section 1: ADT Description

Abstract Data Type (ADT): Stack

The StackInterface<T> defines the basic operations of a stack, a Last-In, First-Out (LIFO) data structure. The stack allows adding elements to the top (push), removing elements from the top (pop), and peeking at the top element without removing it (peek). It also provides the function to check if the stack is empty (isEmpty) and to clear all the elements (clear).

Data Structure Used:
For the implementation of the stack, an array-based is used in the class ArrayStack<T>. The array stores elements with a fixed initial capacity. The size of stack is controlled by an integer variable topOfStack, which tracks the index of the last inserted element. This array is initialized with a default or user-defined capacity and when the stack reaches its capacity, it throws a RuntimeException to signal that the stack is full.

- **Push Operation:** Adds an element to the top of the stack by incrementing the topOfStack index and placing the element in the corresponding array position. The operation checks for stack overflow and throws a RuntimeException if the stack is full, since this implementation uses a fixed-size array.
- **Pop Operation:** Removes and returns the top element of the stack, adjusting topOfStack accordingly. The removed element is set to null to free memory. It also throws the EmptyStackException to announce that Stack is empty.
- **Peek Operation:** Returns the top element without removing it, ensuring the stack is not empty. If there is no element in the stack, it will throw EmptyStackException that the Stack is empty.
- **IsEmpty Operation:** Checks whether the stack is empty by evaluating if topOfStack is less than 0.
- **Clear Operation:** Resets the topOfStack index to -1, effectively clearing the stack.
- **Error Handling:** Push operations will throw an exception if the stack is full, and pop/peek operations will throw exceptions if the stack is empty.

Section 2: Testing Methodology

The testing of the stack ADT (ArrayStack) was tested through the Expression class, which uses the stack to convert infix expression to postfix notation and evaluate the postfix expression.

1. **Infix to Postfix Conversion Testing:** I tested multiple versions of expressions, from simple to complex, valid to invalid expressions, with and without parentheses. This ensures that the stack's **push** and **pop** methods are rigorously exercised while handling operators, parentheses, and different precedences. Tests I used:
    - **Basic Arithmetic:** simple expressions like "1 + 2", "2 * 5", etc. to ensure that the conversion logic is correct, and operators are handled properly.

- **Complex Expression:** Expressions like “(1 + 2) * 3”, etc. were used to test how the stack manages parentheses and operator precedence.
- **Edge Cases:** Expressions like “1 +”, etc. were used to verify that errors are caught when an incomplete expression is provided.

2. **Postfix Evaluation Testing:** The evaluation of the postfix expression used the stack’s **push**, **pop**, and **peek** methods intensively. Tests included:
   - **Simple Postfix Expression:** used the simple expressions like “0 + 0”, “1 + 2”, etc. to check that stack correctly evaluates the result
   - **Expressions with Multiple Operators:** used “1 + 2 * 3” to ensure that stack handles operator precedence correctly. The expression tests whether the evaluation processes the operands in the correct order and whether operations like multiplication and division are applied properly.
   - **Edge Cases:** expression like “0 / 0” to test error cases (division by error). This ensures that the stack’s operations are safely guarded against undefined behavior.
3. **Parentheses Handling:** used test cases with unbalanced parentheses like “(1 + 2”, where the **peek** would look for the top parentheses of stack and **pop** on stack was expected to detect the imbalance. This ensures that the stack properly handles opening and closing parentheses, and throws an error when parentheses are unmatched.

Test Cases:

These are some of the specific test cases used to ensure complete coverage of the stack operations:

1. **Basic Valid Expression:**
   - Input: “0 + 0” -> Postfix: 0 0 + -> Result: 0
   - Input: “1 + 2 * 3” -> Postfix: 1 2 3 * + -> Result: 7
   - Input: “10 / 2” -> Postfix: 10 2 / -> Result: 5
2. **Complex Expressions:**
   - Input: “(1 + 2) * 3” -> Postfix: 1 2 + 3 * -> Result: 9
3. **Invalid and Edge Cases:**
   - Input: “1 +”, “a + 1” -> Error: Invalid Expression
   - Input: “0 / 0” -> Error: / by zero
   - Input: “(1 + 2” -> Error: Expression is unbalanced

Why the Test Cases Are Rigorous and Complete

- **Full Coverage of Stack Operations:** Each of the stack's operation – **push**, **pop**, **peek**, **isEmpty**, and **clear** – is tested through the conversion and evaluation processes in various expressions. Both normal and edge cases are included to ensure that these operations behave as expected.
- **Error Handling:** The test cases include both valid and invalid expressions. It specifically tested how the stack handles errors like division by zero, incomplete expressions, expressions with special characters and unmatched parentheses. This ensures that the ArrayStack properly detects and reports errors.
- **Different Types of Expressions:** The tests cover a wide range of arithmetic expressions, including single operators, multiple operators, and expressions with nested parentheses. These ensure that the stack handles complex scenarios and operator precedence correctly.

Section 3 (Lesson Learned):

- By implementing the stack with arrays, I learned about the process of how the data stored in the array under the stack structure and the importance of memory management (e.g., setting popped elements to null).
- Error Handling: A significant portion of this project involved identifying and handling various edge cases, such as division by zero, unmatched parentheses, and incomplete expressions. This process deepened my understanding of how essential it is to anticipate and account for all possible inputs, especially invalid ones. Learning to include robust error-checking mechanisms is a crucial aspect of software development.
- Efficiency Considerations: Implementing a stack using a fixed-size array required careful consideration of capacity and boundary conditions. This emphasized the trade-offs between simplicity and flexibility. In this case, a fixed-size array was chosen for efficiency, but this limits the stack's maximum capacity. In future implementations, dynamic resizing could offer more flexibility at the cost of increased complexity.
- Testing Strategy: Developing rigorous test cases helped illustrate the importance of thorough testing in ensuring program correctness. Testing is not just testing but also test the invalid input and edge cases, I learned how to design a test suite that covers all the functionality and potential pitfalls of an ADT.
- Documenting both the ADT and its implementation in a clear, concise manner helped solidify my understanding of concepts.
- This project also helps me to understand the use of stack structure. I learned how to call a method helper while inside another method. Finally, I acknowledged that I would have had a better version of ArrayStack by using resizable array and develop the application to process any input with or without the "space" to give the final output with the same result.

Class ArrayStack<T>

java.lang.Object
    ArrayStack<T>

All Implemented Interfaces:

`StackInterface<T>`

```
public class ArrayStack<T>
extends Object
implements StackInterface<T>
```

Name: Pham, Thuy Linh Project: 02 Course: cs-2400-01-f24 Description: This class provides a simple array-based implementation of a stack. It supports basic stack operations like push, pop, peek, and clear, following a Last-In, First-Out (LIFO) structure.

Constructor Summary

Constructors

Constructor	Description
`ArrayStack()`	Default constructor that initializes the stack with a default capacity.
`ArrayStack(int desiredCapacity)`	Constructor that initializes the stack with a specified capacity.

Method Summary

All Methods Instance Methods Concrete Methods

Modifier and Type	Method	Description
`void`	`clear()`	Removes all entries from this stack.
`boolean`	`isEmpty()`	Detects whether this stack is empty.
`T`	`peek()`	Retrieves this stack's top entry.
`T`	`pop()`	Removes and returns this stack's top entry
`void`	`push(T newEntry)`	Adds a new entry to the top pf this stack

Methods inherited from class java.lang.Object

`clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, `wait`, `wait`

Constructor Details

ArrayStack

`public ArrayStack()`

Default constructor that initializes the stack with a default capacity.

ArrayStack

`public ArrayStack(int desiredCapacity)`

Constructor that initializes the stack with a specified capacity.

Parameters:

`desiredCapacity` - The initial capacity of the stack.

Method Details

push

`public void push(T newEntry)`

Adds a new entry to the top pf this stack

Specified by:

`push` in interface `StackInterface<T>`

Parameters:

`newEntry` - An object to be added to the stack

pop

`public T pop()`

Removes and returns this stack's top entry

Specified by:

`pop` in interface `StackInterface<T>`

Returns:

The object at the top of the stack.

Throws:

`RuntimeException` - if the stack is empty before the operation

peek

```
public T peek()
```

Retrieves this stack's top entry.

Specified by:

`peek` in interface `StackInterface<T>`

Returns:

The object at the top of the stack.

Throws:

`RuntimeException` - if the stack is empty

isEmpty

```
public boolean isEmpty()
```

Detects whether this stack is empty.

Specified by:

`isEmpty` in interface `StackInterface<T>`

Returns:

True if the stack is empty

clear

```
public void clear()
```

Removes all entries from this stack.

Specified by:

`clear` in interface `StackInterface<T>`