**SUTD 50.001 Introduction to Information Systems and Programming**

**Problem Set 2B**

* **For all questions, please access the vocareum link found at eDimension for the starter code and to submit.**
* **The Vocareum link is for submission only. Please work on the problems in Android studio, and this includes writing code for the test cases.**
* **To prevent hard-coding, test cases used in Vocareum *may* be different from those provided here and will not be given to you.**

1. [20 points] Write a class called Fibonacci. This class is meant to implement a cached version of the recursive algorithm of the Fibonacci series. You will see that this reduces the number of recursive calls.

**Constructor.** The constructor takes in one int parameter max which initializes the number of elements of the int array data. Thus, in the constructor,

* initialize the instance variable max
* Instantiate the array data to have max elements
* data[0] shall be 0 and data[1] shall be 1.

public int getFibonacciNumber(int n).

* Initialize the calls instance variable to 0. This instance variable tracks the number of times getFibRecursive() is called.
* Returns -1 if n is greater than or equal to max, otherwise this method returns the n-th Fibonacci number by returning the result of executing fibRecursive(n).

You can assume that n and max will be less than or equal to 20. Test Cases:

* getFibonacciNumber(0) 🡪 0
* getFibonacciNumber(1) 🡪 1
* getFibonacciNumber(6) 🡪 8

private int fibRecursive(int n). This is the recursive method which implements the cached version of the recursive algorithm of the Fibonacci series. Here’s the pseudocode.

|  |
| --- |
| 1. Increase **calls** by 1 2. If **n** is 0 or 1, return **data[n]**, else    1. If **data[n]** is not zero, return **data[n]**    2. Else, **data[n]** = **fibRecursive(n-1)** + **fibRecursive(n-2)** and return **data[n]** |

Also implement the following getters for testing by the autograder.

public int getCalls() – return the value of the calls instance variable

public int[] getData() – return the data array

1. [20 points] You are given the following interface CustomStack which takes in a type parameter T.

|  |
| --- |
| **public interface** CustomStack<T> {  **void** push(T t);  T pop();  **int** size();  T peek();  **boolean** isEmpty();  } |

Write a concrete class StackImpl<T> which implements CustomStack.

* Write a no-arg constructor to initialize the List<T> myList instance variable with any concrete class that implements the List<T> interface (eg. ArrayList).
* Implement the abstract methods.
  + **push()** adds element **t** to the top of the stack
  + **pop()** removes the element at the top of the stack and returns it. If the stack is has no elements, return null
  + **peek()** returns the element at the top of the stack, but does not remove it. If the stack has no elements, return **null**.
  + **size()** returns the number of elements in the stack.
  + **isEmpty()** returns **true** if the stack has no elements, and **false** otherwise.
* The stack does not need to have a fixed capacity. You can assume that the stack will not be used in such a way that memory is insufficient.

1. [30 points] Write a static method isBalancedBrackets() to take in a String that contains open brackets, close brackets, and characters which are letters of the alphabet. Open brackets are “({[“ and closing brackets are “)}]”. You can assume that this string is at most 20 characters long.

If the brackets in the string are balanced, return **true**, otherwise return **false**.

To solve this, implement a set of static helper methods,

public static boolean isOpenBracket(), isCloseBracket() and isMatchBracket().

You’ll need to use a **Stack** implementation. From the previous question, copy and paste your code in **StackImpl.java**.

Here’s the pseudocode

1. Initialize a stack.
2. For every char c in the string
   1. If c is an opening bracket, push to stack
   2. If c is a closing bracket,
      1. pop a character d from the stack
      2. if both characters are matching open/close brackets, continue, else brackets are not balanced, return false
3. if the stack is empty, return true, else return false otherwise

Test Cases:

public static void main(String[] args) {

System.out.println(isOpenBracket('(')); // true

System.out.println( isOpenBracket('a')); // false

System.out.println(isCloseBracket(']')); // true

System.out.println(isCloseBracket('a')); // false

System.out.println( isMatchBracket('(',')')); // true

System.out.println( isMatchBracket('a','a')); // false

System.out.println(isBalancedBrackets("()[]()")); // true

System.out.println(isBalancedBrackets("([]())")); // true

}

1. [10 points] The following starter code for an abstract class **FixExpression** is given to you.

This abstract class is meant to be a superclass of concrete classes that will either represent InFix or PostFix expressions and calculate their result.

* Infix expression: 2+3\*4
* Postfix expression: 234\*+

A string containing the Infix or Postfix expression is passed to the constructor.   
Regardless of whether the expression is a Infix or Postfix expression, the first task is to check that the string contains only valid characters, as defined in the instance variable **validChars**.

You can assume that this String containing the infix/postfix expression **has no spaces**. Operations are constrained to the values 0 to 9 and the four arithmetic operators. Hence there will be no such infix expression or equivalent postfix expression: 10+2 (because the maximum value is 9). Furthermore, all operations will result in an integer result.

* Declare **FixExpression** as an abstract class.
* Complete the method **isValidString()** to return true if expression has valid characters, and false otherwise.
* To test your implementation of **isValidChars(),** declare a concrete class which is a subclasss of **FixExpression** and provide a trivial implementation of the abstract method. See next page.

|  |
| --- |
| public abstract class FixExpression {    private String expression;  private String validChars = "0123456789+-\*/";   FixExpression(String expression){  this.expression = expression;  }   public boolean isValidString(){  return true;  }   public String getExpression() {  return expression;  }   public String getValidChars() {  return validChars;  }   public abstract int evaluateExpression(); } |

Testing your abstract class methods:

|  |
| --- |
| public class TestingFixExpression {   public static void main(String[] args) {   FixExpression fixExpression1 = new TrivialImplExpression("1+2");  System.*out*.println(fixExpression1.isValidString()); *//true* FixExpression fixExpression2 = new TrivialImplExpression("abc");  System.*out*.println(fixExpression2.isValidString()); *//false* } }  class TrivialImplExpression extends FixExpression{   TrivialImplExpression(String string){  super(string);  }    // provide a trivial implementation of this abstract method  @Override  public int evaluateExpression() {  return 0;   } } |

1. [20 points] Write a concrete class **PostFixExpression** that is a child class of **FixExpression.** This means you have to implement **evaluateExpression()** to evaluate a postfix expression using your concrete stack class developed in Question 2.

|  |
| --- |
| public class PostfixExpression extends FixExpression {   PostfixExpression(String expression){  super(expression);  }   @Override  public int evaluateExpression(){  return 0;  } } |

* Make **PostfixExpression** a child class of **FixExpression**. Write the constructor accordingly if it is not given to you already.
* Implement **evaluateExpression().** If expression contains invalid characters, return **Integer.MIN\_VALUE** otherwise, you may assume that expression contains a valid postfix expression with the assumptions stated in the previous question. Then, implement **evaluateExpression()** according to the algorithm below.

|  |
| --- |
| 1. Initialize a **stack**. 2. For each char **c** in **expression**,    1. If **c** is an operand, get its int value and push it to **stack**    2. Else **c** is an operator and do the following,       1. pop the **stack** and assign the result to **p1**       2. pop the **stack** and assign the result to **p2**       3. **p3** is the result of the operation **p2** **c** **p1**.       4. push **p3** to the **stack** 3. pop the **stack** and return the result |

* You may also need the following private methods. It is up to you how to declare these methods. The autograder does not test your private methods.
  + isOperator() – returns true if a char is an operator, and false otherwise.
  + isOperand() – returns true if a char is an operand, and false otherwise.
  + getValue() – given operands **p1**, **p2** and operator **c**, return the result of the operation between them

Test Cases

|  |
| --- |
| public class TestPostfixExpression {   public static void main(String[] args) {   FixExpression f1 = new PostfixExpression("12+");  System.*out*.println(f1.evaluateExpression()); *//3* FixExpression f2 = new PostfixExpression("234\*+");  System.*out*.println(f2.evaluateExpression()); *//14* FixExpression f3 = new PostfixExpression("1");  System.*out*.println(f3.evaluateExpression()); *// 1* } } |

Note: writing a similar class for Infix Expression is left to you for your own exploration.