**Lab 6 – Data Element Class**

**Lab Objectives**

* Be able to declare a new class
* Be able to write a constructor
* Be able to write instance methods that return a value
* Be able to write instance methods that take arguments
* Be able to instantiate an object
* Be able to use calls to instance methods to access and change the state of an object

**Introduction**

In Lab 2 we implemented a function that returned a value. In this lab we will create several functions with methods for each, and allow the user to compute the limit from the left of each function. You will create a class called “Function” that will be run by a “driver” class called “EstimateLimit”, which is provided.

Your class will have the following fields:

* **FUNCTION\_CHOICE** – this is an integer constant that references the function the user selects
* **description** – this is a string that describes the chosen function.

The class will have the following methods:

* **getFunctionChoice** – this is an accessor the returns the index of the chosen function
* **setDescription** – this sets the description field with a string that describes the chosen function.
* **getDescription** – this is an accessor that takes an integer indexing one of the functions and returns the string describing the function. This method is used before the user has chosen a function, to list each of the descriptions
* **toString** – this is a method that is already defined for any object in Java. We are “overriding” it to have it return the string which describes the chosen function.
* **computeFunctionValue** – this takes a value for x and computes the value of the function f(x).

We will also need a constructor method that will be used to create an instance of a

Function.

* **Function** – this takes the index of the chosen function, sets the constant that indexes the function and sets the description. It returns the instance (i.e., the object) that will be used.

These ideas can be brought together to form a UML (Unified Modeling Language)

diagram for this class as shown below.

* Function (fnChoice:int)
* getFunctionChoice():int
* setDescription()
* getDescription (fnChoice:int):String
* toString():String
* computeFunctionValue(x:double):double

Methods

Function

Class name

Fields

+ public

* private
* FUNCTION\_CHOICE: double
* description:String

Return type

**Task #1 Creating a New Class**

1. In a new file, create a class definition called Function.
2. Put a program header (comments/documentation) at the top of the file in Javadoc format:

/\*\*

\* The purpose of this class is …

\* Your name and today’s date

\*/

1. Declare the fields listed in the UML diagram.
2. Write a comment for each field indicating what it represents.
3. Save this file as Function.java.
4. Compile and debug. Do not run.

**Task #2 Writing a Constructor**

1. Create a constructor method that has one parameter, the index of the selected function.
2. In the constructor, assign the value taken in from the parameters to the corresponding constant field, and call the mutator (“setter”) called setDescription to set the description. Be sure the constant is set before calling the setter. (You will need to create a “stub” of the method setDescription in order to compile – in Eclipse just hover over the error message and choose the “create method” option)
3. Write comments describing the purpose of the constructor above the method header.
4. Compile and debug. Do not run.

**Task #3 Methods**

1. Complete the mutator (“setter”) method called setDescription to set the value of the field called “description” to the string corresponding to the index supplied to the method. Use standard in-line notation for the functions (for example, use sqrt(x) for , use x^2 for , and use a/b for . Use parenthesis to show order of evaluation.) The functions you will be using are:
2. Define accessor (“getter”) methods called getFunctionChoice and getDescription that return the values of the corresponding fields. The method getDescription takes an integer and returns the description corresponding to the integer. Define the toString() method, which returns the value of the “description” field.
3. Define a method called computeFunctionValue that takes a double x as its parameter and calculates the value of f(x) according to the previously-selected function.
4. Write javadoc comments above each method header.
5. Compile and debug. Do not run.

**Task #4 Running the application**

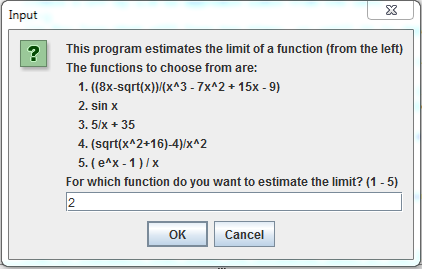
1. You can only execute (run) a program that has a main method, so there is a driver program that is already written to test out your Function class. Copy the file EstimateLimit.java (see code listing 4.1) from Blackboard. Make sure it is in the same directory as Function.java.
2. Compile and run EstimateLimit and follow the prompts.
3. If your output compiles and matches the output below, Function.java is complete and correct. You will not need to modify it further for this lab.

**Task #5 Study the EstimateLimit class**

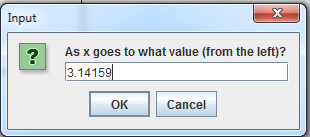
1. You should study how the EstimateLimit class estimates the limits of the functions.
2. After prompting the user to enter the applicable values, the main method instantiates the object ***fn*** by calling Function fn = new Function(fnVersion); This is the syntax for creating any new objects from a class.
3. Then the main method calls estimateLimit with its parameter x being the value the user said to be the limiting argument (as x’ goes from the left closer to the supplied value of x).
4. After decrementing the supplied value of x by 1.0 (x--), a for loop is started which loops 10 times, or until it is determined that the limit diverges. Each loop moves x’ closer to the limiting value by adding 9/(10^i) where i is the loop counter. So as i increments, 10^i increases by a power of ten, resulting in the series . So at each loop, x’ gets closer to x.
5. At each loop, f(x’) is computed, and the value of f(x’) is remembered.
6. After the first 5 iterations, if f(x”) is larger than 1.0 away from f(x’), it is estimated that the function is diverging, and the result is returned that the limit does not exist.
7. Otherwise, after 10 iterations, the value of is estimated by rounding f(x’) to two decimal places.
8. Note also how the main method forces the user to enter valid function choices, by looping until the selection is between 1 and 5.

**Expected Output**:

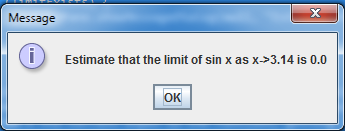
At startup –



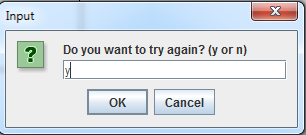
The next screen is:



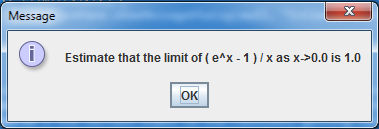
The limit is estimated as:



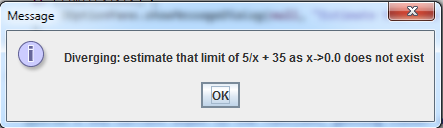
The driver loops using a do-while construct as long as the user wants to try again:



You should test all five functions with various values of x. Function 5 ((e^x-1)/x) is an interesting one:



If the limit calculation diverges, the program displays the following:



**Code Listing 5.1 – EstimateLimit.java**

**import** javax.swing.JOptionPane;

/\*\*

\* Driver to estimate the limit of one of several functions

\* as x goes to a specified value from the left

\* **@author** ralexander

\*

\*/

**public** **class** EstimateLimit {

**private** **static** Function *fn*;

**public** **static** **void** main(String[] args) {

//declare local variables

**int** fnVersion = 0;

**double** x = 0.0;

String input;

**boolean** again = **false**;

//do loop that repeats if user wants to estimate another limit

**do** {

//displays the options and prompts the user to choose one

//this should not be hardwired, but call the getDescription(i) methods of the Function class

input = JOptionPane.*showInputDialog*(**null**,"This program estimates the limit of a function (from the left)\n"+

"The functions to choose from are:\n"+

" 1. "+Function.*getDescription*(1)+"\n"+

" 2. "+Function.*getDescription*(2)+"\n"+

" 3. "+Function.*getDescription*(3)+"\n"+

" 4. "+Function.*getDescription*(4)+"\n"+

" 5. "+Function.*getDescription*(5)+"\n"+

"For which function do you want to estimate the limit? (1 - 5)");

//parses the string into an integer.

//Does not catch a non-integer entry - will cause an unhandled exception

fnVersion = Integer.*parseInt*(input);

//while loop to ensure choices are between 1 and 5

**while**(fnVersion<1 || fnVersion>5) {

input = JOptionPane.*showInputDialog*(**null**,"Input must be 1, 2, 3, 4, or 5. Try again\n");

fnVersion = Integer.*parseInt*(input);

}

//create the chosen function instance

*fn* = **new** Function(fnVersion);

//user enters the value that x goes to

input = JOptionPane.*showInputDialog*(**null**,"As x goes to what value (from the left)?");

//parses the string result into a double

x = Double.*parseDouble*(input);

//call the method to estimate the limit

*estimateLimit*(x);

//prompt the user to ask if he/she wants to estimate another limit

String againString = JOptionPane.*showInputDialog*(**null**,"Do you want to try again? (y or n)");

//if the user entered y, set the boolean again to true

**if** (againString.equals("y")) {

again=**true**;

}

//if the user entered anything except y, set the boolean again to false

**else** {

again=**false**;

System.***out***.println("Goodbye");

}

}

**while**(again);

}

/\*\*

\* the method estimateLimit computes the approximate value of the chosen function ten times, coming closer

\* and closer to f(x) as x gets closer and closer to the desired value from the left

\* **@param** functionChoice one of five integers that represents the five functions

\* **@param** x the user's input to the limit of the function

\*/

**private** **static** **void** estimateLimit(**double** x) {

//false if the function diverges, true if the limit exists

**boolean** limitExists = **true**;

//whether or not the user wants to repeat the process

**boolean** again=**true**;

//to test if the function is diverging, we need to know the previous value of the function

**double** previousResult=0;

//the current value of the function

**double** result=0;;

System.***out***.println("Estimating the limit of "+*fn*+" as x goes to "+x+" from the left");

//System.out.println("Estimating the limit of f"+fn.getFunctionChoice()+"(x) as x goes to "+x+" from the left");

//back x off by 1.0 to approach limit from the left

x--;

//a for loop that will loop ten times, or until it is determined that the function is diverging

**for** (**int** i=1; (i<10 && limitExists); i++) {

//for each iteration of the for loop, calculate the value of the function with a closer and closer value of the input

result = *calcFunction*(x+=(9/Math.*pow*(10,i)));

//an if statement that tests if the previous result is more than 1.0 away from the current result

//the if statement ensures that at least five iterations are done before testing for divergence

**if** ((i > 5)&&Math.*abs*(previousResult-result)>1.0) {

JOptionPane.*showMessageDialog*(**null**, "Diverging: estimate that limit of "+*fn*+" as x->"+(Math.*round*(x\*100))/100.0+" does not exist");

limitExists = **false**;

}

**else** {

System.***out***.println("for x = "+x+", "+*fn*+" = "+result);

previousResult = result;

}

}

//an if statement that shows the estimated limit if the limit exists, rounded to two decimal places

**if** (limitExists) {

JOptionPane.*showMessageDialog*(**null**, "Estimate that the limit of "+*fn*+" as x->"+(Math.*round*(x\*100))/100.0+" is "+(Math.*round*(result\*100))/100.0);

}

}

/\*\*

\* calcFunction calls the appropriate function calculation

\* **@param** functionChoice the function the user has selected

\* **@param** x the current input to the function, getting closer and closer to the desired

\* value with each call to this function

\* **@return** a double which is the value of the function at x

\*/

**private** **static** **double** calcFunction(**double** x) {

**return** *fn*.computeFunctionValue(x);

}

}