Eric Olsen

Assumptions:

The purpose of this homework assignment is to design and implement a Java program that generates a polynomial and calculates the result of its total sum. The instructions of this homework specify a few assumptions that we must make. First, we will take an integer array and designate the first 10 indices as the powers of our given polynomial. We are consequently dealing with a polynomial to the tenth degree. Second, we will assign a random integer coefficient between the value of -9 and 9 for each of the coefficients preceding the polynomial variables. The sum of all the polynomial's components is what we will calculate using two methods: calculateBruteForce and calculateHorner. These methods will accept as parameters the integer array and a random integer between 1 and 9 to represent the variable's value. The output of both these methods will be the polynomial's sum. Additionally, we will not utilize the Math class in the Java library when conducting our brute force calculation.

Main Design Decisions:

The calculateBruteForce method consists of two methods which fit into two nested 'for' loops. Rather than describe the code, an abbreviated version of it is depicted below:

```
public int calculateBruteForce(int[] poly, int
variable){
   int solution = 0;
      for (int i = 0; i < poly.length; i++){
         solution += poly[i] * power(variable, i);

      }
      return solution;
}
```

```
private int power(int variable, int power){
   int newValue = 0;
   int updatedVar = variable;
      if (power == 0){
         newValue = 1;
      }
      else {
         for (int i = 0; i < power - 1; i++){

            updatedVar *= variable;
         }
         newValue = updatedVar;
      }
         return newValue;
}
```

At the inner-most section of this method we have a 'for' loop which iterates up to 9 times. This method is best represented as the sum of all nine iterations: $\sum_{k=1}^{n-1} x^k$. Since our iterations grow by increasing powers, this becomes Big-O(c^n). The outer-section of this method iterates for a fixed number of operations and depends on the number of powers in our polynomial. It is best represented by $\sum_{k=0}^{n} k$. This becomes Big-O(n). Our resulting equation is Big-O(n + c^n). As input size grows this will simply become Big-O(c^n). This is a very inefficient algorithm.

Next we have the calculateHorner method which utilizes the concept of synthetic substitution. Here is the java code:

```
private int power(int variable, int power){
                int newValue = 0;
                int updatedVar = variable;
                if (power == 0){
                        newValue = 1;
                }
                else {
                        for (int i = 0; i < power - 1; i++){
                                updatedVar *= variable;
                        }
                        newValue = updatedVar;
                }
                return newValue;          }
```

As specified in the HW1 indications file, we need to take an equation like what we see below:

$P(x) = a_0 * x^0 + a_1*x^1 + \ldots + a_{n-1}*x^{n-1}$

and turn it into:

$P(x) = a_0 + x * (a_1 + x * (a_2 + \ldots x * (a_{n-2} + (x * a_{n-1}))) \ldots)$

In the case of this method we start at a_{n-1} which is the inner-most position of our method. We then sum this value with the product of a_{n-2} and our random variable. We work our way to the outside of the parentheses by running a decrementing 'for' loop from the second to last position of the integer array until it reaches zero. In terms of Big-O analysis, this method can be represented as $\sum_{k=0}^{9} x * a_k$ which becomes Big-O(n *c). This simplifies to Big-O(n) as input sizes grow.

When comparing the Big-O notation of the calculateBruteForce to calculateHorner, we see that the brute force method is Big-O(c^n) and the horner method is Big-O(n). Without looking at any results, we should argue that the horner method requires fewer resources to execute. The results below from test case 1 confirm that the horner method operates at about one third the time as the bruteForce method.

Error Handling:

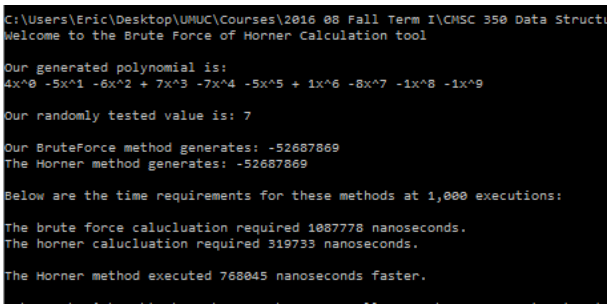
This program does not take any user input, so I omitted any specific exception handling. It is worth noting, however, that the sum of this polynomial with ten terms is very close to exceeding the maximum size of an integer at 2,147,483,647. As a preventative measure, I declared the size of our integer array as a final field in the PolyVal class. I also included 'if' statements inside both calculation methods which closed the program if the integer array exceeded 10 values. See test case 2 for a screen shot where I altered the code and tried to generate an 11th polynomial term.

Test Plan:

The chart below shows the test plan implemented in this program. Test case 1 validated a successful compilation. Test case 2 validated the program's error handling. Tests 3 and 4 further verified that the calculation methods were working properly. As of this project's submission, I can see no errors that will prevent the program from operating properly within the assumptions listed above.

Test Case #	Test Case	Input	Description	Expected Output	Actual Output	Pass / Fail
1	X = 7	A polynomial represented by 10 random coefficients and a variable set to 7	Program should display the polynomial, run both calculations, display the results, execute each of the calculations 1,010 times, and display the difference in time requirements	The appropriate sum of our ten program generated sections of the polynomial	BruteForce: -52,687,869 Horner: -52,687,869 Brute Force time: 1,087,778 nanoseconds Horner time: 319,733nanoseconds	pass
2	X = 1 array size = 11	unchanged	unchanged	Application shut down	Application shut down	pass
3	X = 1	unchanged	unchanged	varies	Brute Force = -1 Horner = -1	pass
4	X = 2	unchanged	unchanged	varies	Brute Force = 1756 Horner = 1756	pass

TestCase 1: successful compilation



Test Case 2: exceeded the size of the integer array