МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное
образовательное учреждение высшего образования
«Самарский национальный исследовательский университет
имени академика С.П. Королева»
(Самарский университет)

ОТЧЕТ ПО
ЛАБОРАТОРНОЙ РАБОТЕ № 2

«Разработка набора классов для работы с функциями одной переменной, заданными в табличной форме»

Выполнила: Элибекян
Анжелина, группа
6203-010302D

Задание №1

Изначально я создала пакет “functions”, в котором разместила все классы программы.

Задание №2

Я реализовала класс `FunctionPoint` для представления точки функции с координатами **(**x, y**).** В классе я предусмотрела три конструктора и методы доступа к полям:

1 конструктор **–** принимает значение параметров для создания точки.

2 конструктор **–** копирования для возвращения копии уже существующей точки.

3 конструктор **–** создает точку в начале координат **(0, 0).**

Также я реализовала геттеры и сеттеры в классе FunctionPoint для обеспечения доступа к приватным полям класса.

```
// Конструктор с параметрами
   public FunctionPoint(double x, double y){
      this.x = x;
      this.y = y;
   }
   // Конструктор копирования
   public FunctionPoint(FunctionPoint point) {
      this.x = point.x;
      this.y = point.y;
   }

   // Конструктор по умолчанию
   public FunctionPoint() {
      this.x = 0;
      this.y = 0;
   }

   // Геттеры и сеттеры
   public double getX() { return x; }
   public void setX(double x) { this.x = x; }
   public double getY() { return y; }
   public void setY(double y) { this.y = y; }
}
```

Задание №3

Далее я создала класс “TabulatedFunction” и реализовала два конструктора. В первом конструкторе я создаю функцию с заданным количеством точек, равномерно распределяя их по интервалу от “leftX” до “rightX” с нулевыми значениями “y”. Для этого я рассчитываю шаг между точками и в цикле создаю каждую точку с соответствующими координатами. Для хранения точек я использую массив FunctionPoint**[]** и отдельное поле pointsCount

```
public class TabulatedFunction {
   private FunctionPoint[] arrPoints;
   private int pointsCount;

   // Конструктор 1: создает функцию с равномерным распределением точек
   public TabulatedFunction(double leftX, double rightX, int pointsCount){
      this.pointsCount = pointsCount;
      this.arrPoints = new FunctionPoint[pointsCount];
      double x_step = (rightX - leftX) / (pointsCount - 1);
      for (int i = 0; i < pointsCount; i++) {
         double x = leftX + i * x_step;
         arrPoints[i] = new FunctionPoint(x, 0);
      }
   }
```

Второй конструктор позволяет создать функцию с предопределенными значениями “y”, переданными в виде массива. Я использую длину этого массива для определения количества точек, сохраняю равномерное распределение по “x”, но устанавливаю y-координаты из переданных значений. Оба конструктора обеспечивают строгую упорядоченность точек по возрастанию “x”.

```
 // Конструктор 2: создает функцию с заданными значениями y
   public TabulatedFunction(double leftX, double rightX, double[] values) {
      this.pointsCount = values.length;
      this.arrPoints = new FunctionPoint[pointsCount];
      double x_step = (rightX - leftX) / (pointsCount - 1);
      for (int i = 0; i < pointsCount; i++) {
         double x = leftX + i * x_step;
         arrPoints[i] = new FunctionPoint(x, values[i]);
      }
   }
}
```

Задание №4

В классе "TabulatedFunction" для определения границ области определения я описываю методы getLeftDomainBorder() и getRightDomainBorder(), которые возвращают x–координаты первой и последней точки. В методе getFunctionValue(double x) я реализую линейную интерполяцию. Сначала проверяю, находится ли точка "x" в пределах области определения функции – если нет, возвращаю Double.NaN. Затем я ищу интервал между двумя точками, в который попадает "x", и вычисляю значение "y" через уравнение прямой между соседними точками(интерполяция).

```
public double getLeftDomainBorder() {
   return arrPoints[0].getX();
}

public double getRightDomainBorder() {
   return arrPoints[pointsCount - 1].getX();
}

public double getFunctionValue(double x) {
   if (x < getLeftDomainBorder() || x > getRightDomainBorder()) {
      return Double.NaN;
   }

   for (int i = 0; i < pointsCount - 1; i++) {
      double x1 = arrPoints[i].getX();
      double x2 = arrPoints[i + 1].getX();
      if (x >= x1 && x <= x2) {
         double y1 = arrPoints[i].getY();
         double y2 = arrPoints[i + 1].getY();
         return y1 + (y2 - y1) * (x - x1) / (x2 - x1);
      }
```

Задание №5

Для реализации методов для работы с точками табулированной функции я описываю методы в классе "TabulatedFunction":
getPointsCount() – возвращает текущее количество точек.
getPoint(int index) – возвращает копию точки из массива(используя конструктор копирования). Тем самым обеспечивая инкапсуляцию.

При реализации методов setPoint(int index, FunctionPoint point) и setPointX(int index, double x) добавляю проверки на сохранения порядка точек. Проверяю, что новая x–координата точки больше x–координаты предыдущей точки и меньше x–координаты следующей точки.

```
// если точка не первая, проверяем что ее x больше x предыдущей точки
```

```
    if (index > 0 && point.getX() <= arrPoints[index - 1].getX()) {
        return;
    }

    // если точка не последняя, проверяем что ее x меньше x следующей точки
    if (index < pointsCount - 1 && point.getX() >= arrPoints[index + 1].getX()) {
        return;
    }
```

Для методов работы с y-координатами я не стала добавлять проверки, т.к. изменение y-координаты не влияет на порядок точек.
Также я использовала геттеры и сеттеры объектов FunctionPoint, а не прямой доступ к полям, чтобы сохранить принцип инкапсуляции.

Задание №6

В данном задании я описываю методы :
deletePoint**(**int index**) –**метод удаления точки. Добавляю проверку на валидность точки, т.е. если индекс выходит за границы существующих точек, завершаем работу без изменений. Также используем метод System.arraycopy**()**, благодаря которому эффективно перемещаю блок данных, начиная со следующей после удаляемой точки, на одну позицию влево. Это позволяет избежать создания нового массива при каждой операции удаления. После сдвига я устанавливаю последний элемент в null, чтобы избежать утечек памяти, и у меньшая счетчик точек.

Самой сложной задачей для меня оказалась реализация метода addPoint**(**FunctionPoint point**)**, т.к. нужно было не просто добавить точку, но и сохранить упорядоченность массива по x**–**координатам. Сначала последовательно перебираю точки пока не найду первую точку с x**–**координатой больше или равной добавляемой, чтобы найти правильную позицию для вставки точки. Также добавляю проверку на дубликаты **–** я сравниваю x**–**координату новой точки с существующими, используя машинный эпсилон **(1e–10)** , что является хорошим решением при сравнение вещественных чисел, из**–**за особенностей внутреннего представления чисел типа double в компьютере. В конечном итоге, если точка с такой x**–**координатой существует, я не добавляю новую точку, дабы сохранить уникальность точек по x. Еще я реализовала механизм динамического расширения массива, т.е. если массив заполнен, я создаю новый массив вдвое большего размера и копирую в него все существующие точки.
Чтобы вставить новую точку, я снова использую System.arraycopy**()** для сдвига части массива вправо, освобождая позицию для новой точки. Вставляю копию переданной точки и увеличиваю счетчик точек.

```
public void deletePoint(int index) {
```

```
    if (index < 0 || index >= pointsCount){
        return;
    }
    int elementsToMove = pointsCount - index - 1;
    if (elementsToMove > 0) {
        System.arraycopy(arrPoints, index + 1, arrPoints, index, elementsToMove);
    }
    arrPoints[pointsCount - 1] = null;
    pointsCount--;
}

public void addPoint(FunctionPoint point) {
    int insertIndex = 0;
    while (insertIndex < pointsCount && arrPoints[insertIndex].getX() < point.getX()) {
        insertIndex++;
    }

    if (insertIndex < pointsCount && Math.abs(arrPoints[insertIndex].getX() - point.getX())
< 1e-10) {
        return;
    }

    if (pointsCount >= arrPoints.length) {
        FunctionPoint[] newPoints = new FunctionPoint[arrPoints.length * 2];
        System.arraycopy(arrPoints, 0, newPoints, 0, pointsCount);
        arrPoints = newPoints;
    }

    System.arraycopy(arrPoints, insertIndex, arrPoints, insertIndex + 1, pointsCount -
insertIndex);
    arrPoints[insertIndex] = new FunctionPoint(point);
    pointsCount++;
}
```

Задание №7

В классе Main тестирую все созданные классы. Я начала с создания табулированной функции y=x^2 на интервале **[0, 4]** с **5** точками. Вручную устанавливаю значения "y" как **(**x*x**)**. Сразу же вывожу все начальные точки, чтобы убедиться в правильности созданной точки. Также я протестировала корректность интерполяции в различных точках, включая граничные случаи и значения вне области определения. После проверки операций модификации: добавление новых точек, удаление и изменение существующих, я убедилась в сохранении упорядоченности данных. Финальное тестирование подтвердило корректность реализации всех

алгоритмов. Ниже прикрепляю результат:

```
"C:\Program Files\Java\jdk-24\bin\java.exe" "
=== ТЕСТИРОВАНИЕ TABULATEDFUNCTION ===

1. Создание функции y = x^2
Начальные точки функции:
(0,00, 0,00)
(1,00, 1,00)
(2,00, 4,00)
(3,00, 9,00)
(4,00, 16,00)

2. Интерполяция на исходной функции
f(0,50) = 0,50
f(1,20) = 1,60
f(2,50) = 6,50
f(3,20) = 10,40
f(6,80) = NaN

3. Область определения
Левая граница: 0,00
Правая граница: 4,00
Проверка точек вне области определения:
f(-1,00) = не определено
f(10,00) = не определено

4. Добавление точек
После добавления двух точек:
(0,00, 0,00)
(1,00, 1,00)
(1,50, 2,25)
(2,00, 4,00)
(3,00, 9,00)
```

```
(3,00, 9,00)
(3,50, 12,25)
(4,00, 16,00)

5. Удаление точки
После удаления точки с индексом 3:
(0,00, 0,00)
(1,00, 1,00)
(1,50, 2,25)
(3,00, 9,00)
(3,50, 12,25)
(4,00, 16,00)

6. Изменение точки
После изменения точки с индексом 2:
(0,00, 0,00)
(1,00, 1,00)
(1,50, 2,25)
(3,00, 9,00)
(3,50, 12,25)
(4,00, 16,00)

7. Финальная интерполяция
f(0,50) = 0,50
f(1,20) = 1,50
f(2,50) = 6,75
f(3,20) = 10,30
f(6,80) = NaN

8. Итоговое состояние функции
(0,00, 0,00)
(1,00, 1,00)
```

```
8. Итоговое состояние функции
(0,00, 0,00)
(1,00, 1,00)
(1,50, 2,25)
(3,00, 9,00)
(3,50, 12,25)
(4,00, 16,00)
```