```java
import org.apache.commons.math3.analysis.UnivariateFunction;
import org.apache.commons.math3.exception.NoDataException;
import org.apache.commons.math3.exception.NullArgumentException;
import org.apache.commons.math3.exception.util.LocalizedFormats;
import org.apache.commons.math3.util.MathUtils;

/**
 * Implementierung eines Polynoms in Bézierdarstellung angelehnt an
 * {@code org.apache.commons.math3.analysis.polynomials.PolynomialFunction}.
 */
public class BezierDarstellung implements UnivariateFunction {

        /**
         * The Bezierpoints of the polynomial, ordered by degree -- i.e.,
         * b[0] is {@code b_0} and b[n] is the coefficient of {@code _s^t B_n^n^}
         * where n is the degree of the polynomial.
         */
        private final double[] b;
        /**
         * Die Intervallgrenzen {@code [s, t]} über dem die Basispolynome
         * definiert sind.
         */
        private final double s, t;

        /**
         * Construct a polynomial with the given Bezierpoints. The first element
         * of {@code double[] c} is the Bezierpoint for {@code B_0^(k+1)}.
         * Higher degree Bezierpoints follow in sequence.  The degree of the
         * resulting polynomial is the index of the last element of the array.
         * <p>
         * The constructor makes a copy of the input array and assigns the copy to
         * {@code double[] b}.</p>
         *
         * @param b Bezierpoints.
         * @throws NullArgumentException if {@code c} is {@code null}.
         * @throws NoDataException if {@code c} is empty.
         */
        public BezierDarstellung(double[] b, double s, double t)
                throws NullArgumentException, NoDataException {
            this.s = s;
            this.t= t;
            MathUtils.checkNotNull(b);
            int n = b.length;
            if (n == 0) {
                throw new NoDataException(
                        LocalizedFormats.EMPTY_POLYNOMIALS_COEFFICIENTS_ARRAY);
            }
            this.b = new double[n];
            System.arraycopy(b, 0, this.b, 0, n);
        }

        /**
         * Auswertung des Polynoms in Bézierdarstellung an der Stelle {@code x}.
         * @param x die Stelle an der ausgewertet werden soll.
         * @return Funktionswert an der Stelle {@code x}.
         */
        public double value (double x) {
            return deCasteljau(x, mu(x, s, t), b.length - 1, 0);
        }

        /**
         * Auswertung der {@code m}-ten Ableitung, {@code m = 0, 1, 2} des
```

```
63          * Polynoms in Bézierdarstellung an der Stelle {@code x}.
64          * @param x die Stelle an der ausgewertet werden soll.
65          * @param m Grad der Ableitung, die ausgewertet werden soll.
66          * @return Ableitungsswert an der Stelle {@code x}.
67          */
68         public double derivative (double x, int m) {
69             double tempValue = 0;
70             double mu = mu(x, s, t);
71             switch (m) {
72             case 0:
73                 tempValue = value(x);
74                 break;
75             case 1:
76                 tempValue = (b.length - 1) / (t - s)
77                         * (deCasteljau(x, mu, b.length - 2, 1)
78                                 - deCasteljau(x, mu, b.length - 2, 0));
79                 break;
80             case 2:
81                 tempValue = (b.length - 1) * (b.length - 2) / Math.pow(t - s, 2)
82                         * (deCasteljau(x, mu, b.length - 3, 0)
83                                 - 2 * deCasteljau(x, mu, b.length - 3, 1)
84                                 + deCasteljau(x, mu, b.length - 3, 2));
85                 break;
86             }
87             return tempValue;
88         }
89 
90         /**
91          * Returns a copy of the Bezierpoints.
92          * <p>
93          * Changes made to the returned copy will not affect the Bezierpoints of
94          * the polynomial.</p>
95          *
96          * @return a fresh copy of the Bezierpoints array.
97          */
98         public double[] getBezierpunkte() {
99             return (double[]) b.clone();
100        }
101
102        /**
103         * Berechnet den häufig auftretenden Faktor {@code \mu(x) := (x-s)/(t-s)}.
104         * @param x Wert, für den {@code \mu(x)} berechnet werden soll.
105         * @return {@code \mu(x)}
106         */
107        public static double mu (double x, double s, double t) {
108            return (x-s)/(t-s);
109        }
110
111        /**
112         * Führt den Algorithmus von deCasteljau durch.
113         * @param x Stelle die ausgewertet werden soll.
114         * @param mu Quotient {@code (x-s)/(t-s)} für die Berechnung der Rekursion
115         * mit den Intervallgrenzen {@code [s, t]}.
116         * @param r oberer Index {@code b_i^r}.
117         * @param i unterer Index {@code b_i^r}.
118         * @param b Feld der Bezierpunkte.
119         * @return Für {@code i = 0, r = n} der Funktionswert des Polynoms in
120         * Bezierdarstellung mit Bezierpunkten {@code doube[] b} an der Stelle
121         * {@code x).
122         */
123        public double deCasteljau (double x, double mu, int r, int i) {
124            if (r == 0)
```

```
125                 return b[i];
126             else
127                 return  mu * deCasteljau(x, mu, r - 1, i + 1) + (1 - mu) *
128                         deCasteljau(x, mu, r - 1, i);
129         }
130     }
```