```java
package hyperDap.base.types.dataSet;

import java.util.ArrayList;
import java.util.List;
import java.util.function.DoubleFunction;
import hyperDap.base.helpers.Comparator;
import hyperDap.base.helpers.Tangenter;
import hyperDap.base.types.value.ValuePair;

/**
 * An implementation of {@link DataSet} that holds {@link Number} Objects as the dependentValue.
 * <p>
 * This implementation and its subclasses are collections of values, as opposed to
 * {@link NestedDataSet} subclasses, and feature methods to retrieve and manipulate these values
 * efficiently.
 *
 * @author soenk
 *
 * @param <T>
 */
public class ValueDataSet<T extends Number> extends ValidityDataSet<T> {

  ArrayList<T> values;
  /**
   * A record of how precise this DataSet is. Used to assess whether two yValues are equal or not.
   */
  protected final double yPrecision;

  protected ArrayList<Integer> derivDepths;

  protected DoubleFunction<T> fromDouble;

  public ValueDataSet(Number base, Number step, Number yPrecision) {
    super(base, step);
    this.values = new ArrayList<T>();
    this.yPrecision = yPrecision.doubleValue();
    this.derivDepths = new ArrayList<Integer>();
  }

  /**
   * A Constructor that provides a conversion function, that is used in {@link #add(double)} to
   * convert to {@code T}.
   * <p>
   * This function must be provided when
   *
   * @param base
   * @param step
   * @param yPrecision
   * @param convertFromDouble
   */
  public ValueDataSet(Number base, Number step, Number yPrecision,
      DoubleFunction<T> convertFromDouble) {
    super(base, step);
    this.values = new ArrayList<T>();
    this.yPrecision = yPrecision.doubleValue();
    this.derivDepths = new ArrayList<Integer>();
    this.fromDouble = convertFromDouble;
  }

  // conversion function
  //
  ************************************************************************************************
  ***********************************

  /**
   * Assigns the fromDouble {@link Double Function Function} if not already assigned.
   * <p>
   * In the interest of consistency this function should only be assigned once.
   *
```

```
68       * @param convertFromDouble A {@link DoubleFunction} to convert to the {@code
         DataSet's} type
69       *        {@code T}
70       * @throws Exception When the function is already assigned.
71       */
72      public void addConversionFunction(DoubleFunction<T> convertFromDouble) throws
         Exception {
73        if (this.fromDouble == null) {
74          this.fromDouble = convertFromDouble;
75        } else {
76          throw new Exception("Conversion Function has already been assigned!");
77        }
78      }
79
80      /**
81       * Check whether the conversion function used in {@link #add(double)} has been
         assigned.
82       *
83       * @return {@code true} if the function is defined, {@code false} otherwise.
84       */
85      public boolean hasConversionFunction() {
86        if (this.fromDouble == null) {
87          return false;
88        }
89        return true;
90      }
91
92      // helpers
93      //
         ************************************************************************************
         ****
94
95      public void calcDerivDepths() {
96        this.derivDepths = Tangenter.calcDerivDepth(this);
97      }
98
99      // write
100     //
         ************************************************************************************
         ****
101
102     /**
103      * Encapsulation of {@link #add(Object)} that converts from {@link Double} to
         {@code T} if the
104      * required conversion function has been defined.
105      *
106      * @param value
107      * @return
108      */
109     public boolean add(double value) throws NullPointerException {
110       try {
111         return this.add(this.fromDouble.apply(value));
112       } catch (NullPointerException e) {
113         throw new NullPointerException(
114             String.format("No conversion function has  been assigned to convert from
                 double to T"));
115       }
116     }
117
118     /**
119      * Encapsulation of {@link #add(double, Object)} using {@link ValuePair} input.
120      *
121      * @param valuePair Data that is to be unboxed to add an entry.
122      */
123     public void add(ValuePair<T> valuePair) {
124       double xValue = valuePair.getX().doubleValue();
125       T yValue = valuePair.getY();
126       this.add(xValue, yValue);
127     }
128
129     /**
130      * Allows editing the {@code derivDepth} for specific values.
131      * <p>
```

```
132       * Intended only for use within {@link Tangenter#calcDerivDepth(ValueDataSet)}.
133       *
134       * @param index The index of the value.
135       * @param depth The {@code derivDepth} that is to be set.
136       * @throws IndexOutOfBoundsException If there is no such value in the internal
          {@link ArrayList}
137       *         of {@code derivDepths}.
138       */
139      public void setDerivDepth(int index, int depth) throws IndexOutOfBoundsException {
140        // TODO not complete?
141        this.derivDepths.set(index, depth);
142      }
143
144      /**
145       * Concatenates {@code depths} to the end of the internal {@link ArrayList} of
146       * {@code derivDepths}.
147       *
148       * @param depths
149       */
150      public void addToDerivDepth(List<Integer> depths) {
151        this.derivDepths.addAll(depths);
152      }
153
154      // getters
155      //
          ***************************************************************************
          ***
156
157      /**
158       * Returns the {@code yPrecision} set at construction time. This value defines the
          precision used
159       * when comparing {@code yValues} and making use of
160       * {@link Comparator#equalApprox(double, double, double)}, e.g. in
161       * {@link #contains(double, double)}.
162       *
163       * @return The set precision for {@code yValues}.
164       */
165      public double getPrecision() {
166        return this.yPrecision;
167      }
168
169      /**
170       * Returns the depths to which a trace by trace derivative for this value is not
          zero.
171       *
172       * @param index
173       * @return The number of derivatives that are not zero, can be {@link
          Integer#MAX_VALUE} to
174       *         represent infinity and negative when this value could not be calculated
          normally.
175       * @throws IndexOutOfBoundsException When there is no such value.
176       */
177      public int getDerivDepthsByIndex(int index) throws IndexOutOfBoundsException {
178        if (index < 0 || index >= this.size()) {
179          throw new IndexOutOfBoundsException();
180        }
181        if (index >= this.derivDepths.size()) {
182          this.calcDerivDepths();
183        }
184        return this.derivDepths.get(index);
185      }
186
187      /**
188       * Returns the depths to which a trace by trace derivative for this value is not
          zero.
189       *
190       * @param xValue
191       * @return The number of derivatives that are not zero, can be {@link
          Integer#MAX_VALUE} to
192       *         represent infinity and negative when this value could not be calculated
          normally.
193       * @throws IndexOutOfBoundsException When there is no such value.
194       */
```

```
195     public int getDerivDepth(double xValue) throws IndexOutOfBoundsException {
196       return this.getDerivDepthsByIndex(this.getIndex(xValue));
197     }
198
199     /**
200      * {@link Number} encapsulation of {@link #getDerivDepth(double)}.
201      *
202      * @param xValue
203      * @return The number of derivatives that are not zero, can be {@link
         Integer#MAX_VALUE} to
204      *         represent infinity and negative when this value could not be calculated
         normally.
205      * @throws IndexOutOfBoundsException When there is no such value.
206      */
207     public int getDerivDepth(Number xValue) throws IndexOutOfBoundsException {
208       return this.getDerivDepth(xValue.doubleValue());
209     }
210
211     // contains
212     //
         ********************************************************************************
         ****
213
214     /**
215      * Test if this DataSet contains the specified value at this index.
216      * <p>
217      * <<<<<<< HEAD Only checks for exactly this entry, for checking for a range of
         values around a
218      * specific independent value use {@link #contains(double, double)}. ======= Only
         checks for
219      * exactly this entry, for checking for a value close to this one see
220      * {@link #contains(double, double, double, double)}. >>>>>>> master
221      * <p>
222      * If the index is out of bounds false is returned.
223      *
224      * @param index The index where this value is expected.
225      * @param value The value that should be contained.
226      * @return {@code true} if this @{@code yValue} is stored under this {@code index}.
227      */
228     public boolean contains(int index, Number value) {
229       try {
230         return Comparator.equalApprox(value.doubleValue(),
           this.getByIndex(index).doubleValue(),
231             this.yPrecision);
232       } catch (IndexOutOfBoundsException e) {
233         return false;
234       }
235     }
236
237     /**
238      * Check whether the requested x-y values are represented in this DataSet, given
         the desired
239      * precisions.
240      *
241      * @param xValue
242      * @param yValue
243      * @param xPrecision
244      * @param yPrecision
245      * @return {@code true} if this @{@code yValue} is stored under this {@code xValue}.
246      */
247     public boolean contains(double xValue, double yValue, double xPrecision, double
         yPrecision) {
248       int index = this.getIndex(xValue);
249       double indexValue = this.getIndependentValue(index);
250       if (Comparator.equalApprox(xValue, indexValue, xPrecision) == false) {
251         return false;
252       }
253       List<Integer> list = new ArrayList<Integer>(); // a list of the indices that
           should be checked
254       list.add(index);
255       int i = index - 1;
256       while (Comparator.equalApprox(xValue, this.getIndependentValue(i), xPrecision)) {
257         list.add(i);
```

```
258         i--;
259       }
260       i = index + 1;
261       while (Comparator.equalApprox(xValue, this.getIndependentValue(i), xPrecision)) {
262         list.add(i);
263         i++;
264       }
265       for (int j : list) {
266         try {
267           if (Comparator.equalApprox(yValue, this.getByIndex(j).doubleValue(),
              yPrecision) == true) {
268             return true;
269           }
270         } catch (IndexOutOfBoundsException e) {
271
272         }
273       }
274       return false;
275     }
276
277     /**
278      * Check whether the requested x-y values are contained within this DataSet,
         within the default
279      * precisions defined by the DataSet (yPrecision and step).
280      *
281      * @param xValue The independent value defining the index/indices to be checked.
282      * @param yValue The dependent value that should be contained at the checked
         indices.
283      * @param yPrecision The precision within which the yValue will be considered
         equal to that
284      *        contained at the checked indices.
285      * @return {@code true} if this @{@code yValue} is stored under this {@code xValue}.
286      */
287     public boolean contains(double xValue, double yValue) {
288       return this.contains(xValue, yValue, 0.5 * this.step, this.yPrecision);
289     }
290
291     /**
292      * Check whether this {@link ValuePair} is represented in this DataSet.
293      * <p>
294      * Makes use of {@link #contains(double, double)};
295      *
296      * @param valuePair A pair of the independent and dependent values to be checked.
297      * @return {@code true} if this pair is conatained, {@code false} otherwise.
298      */
299     public boolean contains(ValuePair<? extends Number> valuePair) {
300       return this.contains(valuePair.getX().doubleValue(),
          valuePair.getY().doubleValue());
301     }
302
303     // other
304     //
          ********************************************************************************
          ****
305
306     /**
307      * {@inheritDoc}
308      */
309     @Override
310     public void clear() {
311       super.clear();
312       this.derivDepths.clear();
313     }
314
315     /**
316      * {@inheritDoc}
317      */
318     @Override
319     public void ensureCapacity(int capacity) {
320       super.ensureCapacity(capacity);
321       this.derivDepths.ensureCapacity(capacity);
322     }
323
```

```
324
325     }
326
```