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

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import hyperDap.base.types.value.ValuePair;

/**
 * Superclass for series of Values that can be mapped from an independent value (or xValue) to a
 * dependent one (a yValue), e.g. a {@link ValuePair}. This class provides an encapsulation of
 * {@link ArrayList}, which contains the independent values, that calculates from the independent
 * value to the correct index in this list, and vice versa.
 * <p>
 * This Collection is used to store independent values that are exactly spaced out by {@link step},
 * or where the exact spacing does not matter. At assignment the independent value is only retained
 * in the index within the list, and resulting rounding errors etc. may lead to discrepancies
 * between the intended value and the actual values of the independent values. An exception to this
 * is the {@link PairDataSet}.
 * <p>
 * Retrieving by the independent variable is performed by calculating the corresponding index using
 * the {@link #base} and {@link #step} fields, such that
 * {@code independentValue= base + index*step}. See {@link #getIndex(double)} for details on
 * calculating the index based on the independent value, and {@link #getIndependentValue(int)} on
 * calculating the value corresponding to an integer.
 * <p>
 * Care should be taken when adding elements by their xValue as in {@link #add(double, Object)} or
 * {@link #add(int, Object)}, as this may require filling the DataSet with many new values
 * increasing its memory use, while not substantially increasing the number of meaningful values
 * recorded. If you end up using this method repeatedly in this manner consider adapting the step or
 * using a HashMap implementation of DataSet.
 * <p>
 * This class implements {@link Collection} but not {@link java.util.List List}, as some of the
 * index based operations here would violate the API of {@link java.util.List List}, while some
 * requirements set out in {@link java.util.List List} would not translate well to DataSet. Note for
 * example {@link #add(int, Object)} here compared to {@link java.util.List List#add(int,Object)}.
 *
 * @author soenk
 *
 * @param <T> The type of values stored in this DataSet
 */
public abstract class DataSet<T> implements Collection<T> {

  protected final double base;
  protected final double step;
  protected ArrayList<T> values;

  /**
   * Default constructor.
   *
   * @param base
   * @param step
   */
  public DataSet(Number base, Number step) {
    this.base = base.doubleValue();
    this.step = step.doubleValue();
```

```
56        values = new ArrayList<T>();
57      }
58
59      // helper category
60      //
        ************************************************************************************
        *
61
62      /**
63       * Calculate the index associated with this independent value.
64       * <p>
65       * Calculated as index=({@code independentValue}- {@link #base} ) / {@link #step},
         then rounding
66       * the result with {@link Math#round(double)} and casting to integer.
67       *
68       * @category helper
69       * @param independentValue
70       * @return
71       */
72      public int getIndex(double independentValue) {
73        independentValue = (independentValue - this.base) / this.step;
74        int index = (int) Math.round(independentValue);
75        return index;
76      } // TODO edge cases
77
78      /**
79       * Calculate the independent value associated with the requested index.
80       * <p>
81       * calculated as return= {@link #base} + {@code index} * {@link #step}.
82       *
83       * @category helper
84       * @param index The requested index.
85       * @return The independent value associated with {@code index}
86       */
87      public double getIndependentValue(int index) {
88        double independentValue = this.base + index * this.step;
89        return independentValue;
90      }
91
92      /**
93       * Returns the largest independent value that still maps to an entry.
94       *
95       * @category helper
96       * @return The independent value associated with the last index of {@link #values}.
97       */
98      public double getMaxIndependentValue() {
99        return this.getIndependentValue(this.values.size() - 1);
100     }
101
102     /**
103      * Used within {@link #add(int, Object)} to initialise elements at intermediate
         indices to a
104      * sensible default value.
105      * <p>
106      * <<<<<<< HEAD Should be overwritten by subclasses but will default to null.
         ======= Should be
107      * overwritten by subclasses. >>>>>>> master
108      *
109      * @category helper
110      * @return A default value for elements of this DataSet.
111      */
112     private T initialisationValue() {
113       return null;
114     }
115
116     /**
117      * Used within {@link #add(int, Object)} to initialise elements at intermediate
         indices to a
118      * sensible value based on the two surrounding values (the currently last and the
         newly added
119      * one).
120      * <p>
121      * <<<<<<< HEAD Should be overwritten by subclasses but will default to null.
```

```
     ======= Should be
122   * overwritten by subclasses. >>>>>>> master
123   *
124   * @category helper
125   * @param value1 The currently last value in {@link #values}
126   * @param value2 The value that should be added at the desired index.
127   * @return A default value for elements of this DataSet.
128   */
129  private T initialisationValue(T value1, T value2) {
130    return null;
131  }
132
133  /**
134   * Helper method to cast from other objects to T without triggering warnings.
135   *
136   * @category helper
137   * @param o The Object of any type that should be cast to type {@code T}.
138   * @return The cast of {@code o} if this is possible.
139   * @throws ClassCastException Thrown if there is no legal cast from {@code o} to
     {@code T}.
140   */
141  @SuppressWarnings({"unchecked", "unused"})
142  private T castToT(Object o) throws ClassCastException {
143    return (T) o;
144  }
145
146  // Writing/setters
147  //
     ************************************************************************************
     *
148
149  /**
150   * Call {@link #add(int, Object)} on the index corresponding to this {@code
     independentValue} wtih
151   * {@code index= (independentValue-base)/step}.
152   *
153   * @category writing
154   * @param independentValue The independent value this value should be associated
     with, used to
155   *        calculate its index.
156   * @param value The value that is to be added.
157   */
158  public void add(double independentValue, T value) {
159    this.add(this.getIndex(independentValue), value);
160  }
161
162  /**
163   * Add a value at a specific index. If this value already exists it is replaced.
     If it does not
164   * the list of values is extended to include the required index, with the values
     between the last
165   * and this new one being initialised to the default value specified by
166   * {@link #initialisationValue(Object, Object).
167   * <p>
168   * This method is more comparable to {@link ArrayList#set(int, Object)} as it
     replaces the element
169   * in question rather than shifting elements to the right.
170   *
171   * @category writing
172   * @param index The index at which the value is to be added.
173   * @param value The value that is to be added.
174   */
175  public void add(int index, T value) {
176    if (index < 0) {
177      throw new IndexOutOfBoundsException();
178    }
179    try {
180      this.values.set(index, value);
181    } catch (IndexOutOfBoundsException e1) {
182      this.values.ensureCapacity(index);
183      int lastIndex = this.values.size() - 1;
184      T initValue;
185      try {
```

```
186           initValue = this.initialisationValue(value, this.values.get(lastIndex));
187         } catch (IndexOutOfBoundsException e2) {
188           initValue = value;
189         }
190         for (int i = lastIndex + 1; i < index; i++) {
191           this.values.add(initValue); // the index of this value will be i
192         }
193         this.values.add(value);
194       }
195     }
196
197     /**
198      * {@inheritDoc}
199      *
200      * @category writing
201      */
202     @Override
203     public boolean add(T e) {
204       return this.values.add(e);
205     }
206
207     /**
208      * Add a new value at the end of the DataSet and return the independent value it
         will be
209      * associated with.
210      *
211      * @category writing
212      * @param value The dependent value to be added.
213      * @return The independent value that will be associated with this value
214      */
215     public double addValue(T value) {
216       this.values.add(value);
217       return this.getIndependentValue(this.values.size() - 1);
218     }
219
220     /**
221      * A shortened version of {@link #add(Object)} which does not calculate the
         associated independent
222      * value. It should complete slightly faster than the aforementioned method.
223      *
224      * @category writing
225      * @param valueThe dependent value to be added.
226      */
227     public void quickAdd(T value) {
228       this.values.add(value);
229     }
230
231     // reading/getters
232     //
         ********************************************************************************
         *
233     /**
234      * Get the {@code base} used to convert between {@code index} and {@code xValue}.
235      *
236      * @return The {@code base} value used by this {@link DataSet}
237      */
238     public double getBase() {
239       return this.base;
240     }
241
242     /**
243      * Get the {@code step} used to convert between {@code index} and {@code xValue}.
244      *
245      * @return The {@code step} value used by this {@link DataSet}
246      */
247     public double getStep() {
248       return this.step;
249     }
250
251     /**
252      * Returns the dependent value from the index corresponding to {@code
         independentValue}.
253      * <p>
```

```
254       * In accordance with DataSet specifications the index that is retrieved is the
          integer cast value
255       * of {@code (independentValue-base)/step}.
256       *
257       * @category reading
258       * @param independentValue The independent value associated with the required value.
259       * @return The dependent Value stored at {@code index=(int)
          (independentValue-base)/step}.
260       */
261      public T get(Number independentValue) {
262        return this.getByIndex(this.getIndex(independentValue.doubleValue()));
263      }
264
265      /**
266       * Returns the dependent value from the index corresponding to {@code
          independentValue}.
267       * <p>
268       * In accordance with DataSet specifications the index that is retrieved is the
          integer cast value
269       * of {@code (independentValue-base)/step}.
270       *
271       * @category reading
272       * @param independentValue Used by {@link #getIndex(double)} to calculate the
          index of the desired
273       *        element.
274       * @return The value corresponding to the calculated index.
275       */
276      public T get(double independentValue) {
277        return this.getByIndex(this.getIndex(independentValue));
278      }
279
280      /**
281       * Get an entry by the index in {@link #values}.
282       *
283       * @category reading
284       * @param index
285       * @return
286       */
287      public T getByIndex(int index) {
288        return this.values.get(index);
289      }
290
291      /**
292       * Check whether this collection contains an entry corresponding to the {@code
          independentValue}.
293       * <p>
294       * More formally, returns {@code true} if {@link #get(double)
          get(independentValue)} would return
295       * an entry and {@code false} if it is out of bounds.
296       * <p>
297       * Entries of {@code null} are considered valid and will return {@code true}.
298       *
299       * @param independentValue The xValue
300       * @return
301       */
302      public boolean hasEntryAt(double independentValue) {
303        try {
304          this.get(independentValue);
305          return true;
306        } catch (IndexOutOfBoundsException e) {
307          return false;
308        }
309      }
310      // other inheritances from Collection
311      //
          ********************************************************************************
          ****
312
313      /**
314       * {@inheritDoc}
315       * <p>
316       * Is applied to {@link #values}.
317       *
```

```
318       * @category fromCollection
319       */
320      @Override
321      public int size() {
322        return this.values.size();
323      }
324
325      /**
326       * {@inheritDoc}
327       * <p>
328       * Is applied to {@link #values}.
329       *
330       * @category fromCollection
331       */
332      @Override
333      public void clear() {
334        this.values.clear();
335      }
336
337      /**
338       * Ensures that this {@code DataSet} can hold at least as many values as specified.
339       * <p>
340       * Compare to {@link ArrayList#ensureCapacity(int)}.
341       *
342       * @param capacity
343       */
344      public void ensureCapacity(int capacity) {
345        this.values.ensureCapacity(capacity);
346      }
347
348      /**
349       * {@inheritDoc}
350       * <p>
351       * Is applied to {@link #values}.
352       *
353       * @category fromCollection
354       */
355      @Override
356      public boolean isEmpty() {
357        return this.values.isEmpty();
358      }
359
360      /**
361       * {@inheritDoc}
362       * <p>
363       * Is applied to {@link #values}.
364       *
365       * @category fromCollection
366       */
367      @Override
368      public Iterator<T> iterator() {
369        return this.values.iterator();
370      }
371
372      /**
373       * {@inheritDoc}
374       * <p>
375       * Is applied to {@link #values}.
376       *
377       * @category fromCollection
378       */
379      @Override
380      public boolean contains(Object o) {
381        return this.values.contains(o);
382      }
383
384      /**
385       * {@inheritDoc}
386       * <p>
387       * Is applied to {@link #values}.
388       *
389       * @category fromCollection
390       */
```

```
391     @Override
392     public boolean containsAll(Collection<?> c) {
393       return this.values.containsAll(c);
394     }
395
396     /**
397      * {@inheritDoc}
398      * <p>
399      * Is applied to {@link #values}.
400      *
401      * @category fromCollection
402      */
403     @Override
404     public Object[] toArray() {
405       return this.values.toArray();
406     }
407
408     /**
409      * {@inheritDoc}
410      * <p>
411      * Is applied to {@link #values}.
412      *
413      * @category fromCollection
414      */
415     @Override
416     @SuppressWarnings("unchecked")
417     public Object[] toArray(Object[] a) {
418       return this.values.toArray(a);
419     }
420
421     /**
422      * Replaces the specified Object with a default value, if it exists.
423      *
424      * @category fromCollection
425      * @param o The Object to be removed
426      * @return {@code true} if this DataSet has changed as a result of this operation.
427      */
428     @Override
429     public boolean remove(Object o) {
430       int index = this.values.indexOf(o);
431       if (index == -1) {
432         return false;
433       }
434       this.values.remove(index);
435       this.values.add(index, this.initialisationValue());
436       return true;
437     }
438
439     /**
440      * {@inheritDoc}
441      * <p>
442      * Is applied to {@link #values}.
443      *
444      * @category fromCollection
445      */
446     @Override
447     public boolean addAll(Collection<? extends T> c) {
448       return this.values.addAll(c);
449     }
450
451     /**
452      * Replace all Objects in {@code c} with default values by calling {@link
         #remove(Object)} with
453      * them.
454      *
455      * @category fromCollection
456      * @param The Objects that are to be removed.
457      * @return {@code true} if this DataSet has changed as a result of this operation.
458      */
459     @Override
460     public boolean removeAll(Collection<?> c) {
461       boolean ret = false;
462       for (Object o : c) {
```

```
463         if (this.remove(o) == true) {
464           ret = true;
465         }
466       }
467       return ret;
468     }
469 
470     /**
471      * Not implemented from {@link Collection}.
472      *
473      * @category fromCollection
474      */
475     @Override
476     public boolean retainAll(Collection<?> c) throws UnsupportedOperationException {
477       throw new UnsupportedOperationException();
478     }
479 
480   }
481 
```