```java
package hyperDap.generator.main;

import java.util.ArrayList;
import java.util.Random;
import java.util.function.DoubleFunction;
import java.util.function.Function;
import hyperDap.base.helpers.Comparator;
import hyperDap.base.types.dataSet.ValueDataSet;

/**
 * This class generates a section of data specific to one function.
 * <p>
 * The intended use is to initialise a {@code GenSegment} with the intended values and then retrieve
 * lists of data points from {@link #generateValues(double, int)} as needed, before leaving the
 * Object to be garbage-collected. calling data generation methods repeatedly with the same
 * parameters will produce the same data points within limits given to any randomness (this will be
 * added later to simulate noise).
 * <p>
 * Each Object represents a function of the format {@code a * Func(x + b) + c}, where {@code Func}
 * is a mathematical function specified by {@code functionEncoding} at construction time. The
 * function is shifted such that it passes through the {@code intercept} at {@code x= -step}, which
 * is expected to be the last value before this function begins. This aligns it with the previous
 * values to transition smoothly.
 * <p>
 * The {@code functionEncding} may specify: <br>
 * {@code constant} : {@code y = c} <br>
 * {@code linear} : {@code y = a * (x + b) + c} <br>
 * {@code square} : {@code y = a * (x + b)^2 +c} <br>
 * {@code cubic} : {@code y = a * (x + b)^3 + c} <br>
 * {@code exp} : {@code y = a * Math.E^(x + b) + c} <br>
 * {@code sine} : {@code y = a * sin(x + b) +c} <br>
 * <p>
 * Here {@code a} translates to the {@code scale} specified at construction, {@code b} to
 * {@code shiftX}, while {@code c} is defined at construction such that the function
 * returns @{@code intercept} for {@code x=-step}.
 *
 * @author soenk
 *
 */
public class GenSegment {

  private double step;
  private double a;
  private double b;
  private double c;
  private Function<Double, Double> func;
  private Random rand = new Random();

  /**
   * The default constructor.
   *
   * @param functionEnccoding A {@link String} endocing of the function to be generated.
   * @param scale Used to scale and make the function more or less 'steep' {@code => a}
   * @param shiftX Shifts the function right or left. Use to fit split up functions together (e.g.
   *        bias) {@code => b}
   * @param intercept Used to ensure the first value is in line with previous values and is assumed
   *        to be the last value before this segment. {@code = f(-step) + c}
   *
   */
  public GenSegment(String functionEnccoding, double scale, double shiftX, double
  intercept,
      double step) throws IllegalArgumentException {
    this.step = step;
    a = scale;
    b = shiftX;
    c = 0;
    this.defineFunction(functionEnccoding);
    c = intercept - f(-step);
    System.out.println(String.format("%s Generating Segment of %s with a= %s, b= %s c= %s",
        GenSegment.class, functionEnccoding, this.a, this.b, this.c));
  }

  /**
   * Classifies the function represented by this Object based on {@code encoding.}
   *
   * @param encoding A {@link String} encoding of the function that is to be modelled.
   */
  private void defineFunction(String encoding) throws IllegalArgumentException {
    encoding = encoding.toLowerCase();
    switch (encoding) {
      case "constant":
        func = x -> 0.0;
        break;
      case "linear":
        func = x -> x;
        break;
      case "square":
        func = x -> Math.pow(x, 2);
        break;
      case "cubic":
        func = x -> Math.pow(x, 3);
        break;
      case "exp":
        func = x -> Math.pow(Math.E, x);
        a = a / 1000;
        break;
      case "sine":
        func = x -> Math.sin(x);
        break;
      default:
        throw new IllegalArgumentException(
            String.format("'%s' is not a valid function encoding!", encoding));
    }
  }

  /**
   * Returns a single value of the function specified in this Object.
   * <p>
   * This is specified as {@code a* Function(x + b) +c}.
   *
   * @param x The {@code xValue} to be fed into the function.
   * @return The {@code yValue} corresponding to {@code x}.
   */
  private double f(double x) {
    return a * this.func.apply(x + b) + c;
  }

  /**
   * Returns a single value of the function specified by this Object with added noise. Noise here is
   * a value added to {@link #f(double)}, that is randomly taken from a normal distribution around
   * zero and of standard deviation {@code noise}.
   *
   * @param x The {@code xValue} that is passed to {@link #f(double)}
   * @param noise The {@code standard deviation} of the value added (or subtracted).
   * @return {@link #f(double) f(x)} {@code + noise *} {@link Random#nextGaussian()}.
   */
  private double noisyF(double x, double noise) {
    return f(x) + noise * rand.nextGaussian();
  }

  /**
   * Provides a means to reseed the internal instance of {@link Random}.
   * <p>
   * This may not mean that generated data can be used for machine learning, unless it is generated
   * in small enough segments.
   *
   * @param seed Passed to {@link Random#setSeed(long)}
   */
  public void seedRandom(long seed) {
    this.rand.setSeed(seed);
  }

  /**
   * Encapsulation of {@link #generateValues(int, double)} with {@code noise=0}.
   *
   * @param N The number of data points to be generated.
   * @return An {@link ArrayList} of the generated data points.
   */
  public ArrayList<Double> generateValues(int N) {
    return this.generateValues(N, 0.0);
  }

  /**
   * Generate a list of data points of length {@code N}, according to pre-set specifications and
   * with the set amount of noise.
   *
   * @param N The number of data points to be generated.
   * @param noise The noise factor passed to {@link #noisyF(double, double)}
   * @return An {@link ArrayList} of the generated data points.
   */
  public ArrayList<Double> generateValues(int N, double noise) {
    ArrayList<Double> list = new ArrayList<Double>();
    for (Integer i = 0; i < N; i++) {
      list.add(noisyF(i.doubleValue() * step, noise));
    }
    return list;
  }

  /**
   * Generate the specified data points and add them to the end of {@code set}.
   * <p>
   * Calls {@link ValueDataSet#ensureCapacity(int)} before generating data.
   * <p>
   * Encapsulates {@link #addToDoubleDataSet(ValueDataSet, int, double)} with {@code noise=0}.
   *
   * @param set The {@link CalueDataSet} that the data points should be added to.
   * @param step The distance between data points on the x-axis.
   * @param N The number of data points that should be added.
   * @throws IllegalArgumentException If {@link ValueDataSet#getStep()} is not equal to the pre-set
   *          step.
   */
  public void addToDoubleDataSet(ValueDataSet<Double> set, int N) throws IllegalArgumentException {
    this.addToDoubleDataSet(set, N, 0.0);
  }

  /**
   * Generate the specified data points with noise and add them to the end of {@code set}.
   * <p>
   * Calls {@link ValueDataSet#ensureCapacity(int)} before generating data.
   * <p>
   * Noisy values are created using {@link #noisyF(double, double)}.
   *
   * @param set The {@link CalueDataSet} that the data points should be added to.
   * @param N The number of data points that should be added.
   * @param noise The noise factor passed to {@link #noisyF(double, double)}.
   * @throws IllegalArgumentException If {@link ValueDataSet#getStep()} is not equal to the pre-set
   *          step.
   */
  public void addToDoubleDataSet(ValueDataSet<Double> set, int N, double noise)
      throws IllegalArgumentException {
    int size = set.size();
    if (this.step != set.getStep()) {
      throw new IllegalArgumentException(
          String.format("%s. addToDoubleDataSet() does not match preset step! %s!=%s",
              GenSegment.class, this.step, set.getStep()));
    }
    set.ensureCapacity(N + set.size());
    double val;
    for (Integer i = 0; i < N; i++) {
      val = Double.valueOf(noisyF(i.doubleValue() * step, noise));
      set.add(val);
      // if the value is too large or small, consider it invalid
      // if (Comparator.equalApprox(0.0, val, 10000.0) == false) {
      // set.editValidityByIndex((i + size), false);
      // }
    }
  }

  /**
   * Generate the specified data points and add them to the end of {@code set}.
   * <p>
   * This method requires that {@code set} has an assigned {@code convertFromDouble}
   * {@link DoubleFunction Function} assigned.
   * <p>
   * {@link ValueDataSet#ensureCapacity(int)} is called before adding data points.
   *
   * @param set The {@link CalueDataSet} that the data points should be added to.
   * @param N The number of data points that should be added.
   * @throws IllegalArgumentException If {@link ValueDataSet#hasConversionFunction()} returns
   *          {@code false}.<br>
   *          If {@link ValueDataSet#getStep()} is not equal to the pre-set step.
   */
  public void addToDataSet(ValueDataSet<? extends Number> set, int N)
      throws IllegalArgumentException {
    this.addToDataSet(set, N, 0.0);
  }

  /**
   * Generate the specified data points and add them to the end of {@code set}.
   * <p>
   * This method requires that {@code set} has an assigned {@code convertFromDouble}
   * {@link DoubleFunction Function} assigned.
   * <p>
   * {@link ValueDataSet#ensureCapacity(int)} is called before adding data points. Noisy values are
   * created using {@link #noisyF(double, double)}.
   *
   * @param set The {@link CalueDataSet} that the data points should be added to.
   * @param N The number of data points that should be added.
   * @param noise The noise factor passed to {@link #noisyF(double, double)}.
   * @throws IllegalArgumentException If {@link ValueDataSet#hasConversionFunction()} returns
   *          {@code false}.<br>
   *          If {@link ValueDataSet#getStep()} is not equal to the pre-set step.
   */
  public void addToDataSet(ValueDataSet<? extends Number> set, int N, double noise)
      throws IllegalArgumentException {
    int size = set.size();
    if (set.hasConversionFunction() == false) {
      throw new IllegalArgumentException(
          "ValueDataSet must have a convertFromDouble function defined!");
    }
    if (this.step != set.getStep()) {
      throw new IllegalArgumentException(
          String.format("%s. addToDoubleDataSet() does not match preset step! %s!=%s",
              GenSegment.class, this.step, set.getStep()));
    }
    set.ensureCapacity(N + set.size());
```

```java
    double val;
    for (Integer i = 0; i < N; i++) {
      val = noisyF(i.doubleValue() * step, noise);
      set.add(val);
      if (Comparator.equalApprox(0.0, val, 10000) == false) {
        set.editValidityByIndex(i + size, false);
      }
    }
  }

  public void addRandomToDoubleDataSet(ValueDataSet<Double> set, int N)
      throws IllegalArgumentException {
    if (this.step != set.getStep()) {
      throw new IllegalArgumentException(
          String.format("%s. addToDoubleDataSet() does not match preset step! %s!=%s",
              GenSegment.class, this.step, set.getStep()));
    }
    set.ensureCapacity(set.size() + N);
    Integer temp;
    for (int i = 0; i < N; i++) {
      temp = (rand.nextInt(10000) - 5000);
      set.add(temp.doubleValue() / 100);
    }
  }

  public void addRandomToDataSet(ValueDataSet<? extends Number> set, int N)
      throws IllegalArgumentException {
    if (set.hasConversionFunction() == false) {
      throw new IllegalArgumentException(
          "ValueDataSet must have a convertFromDouble function defined!");
    }
    if (this.step != set.getStep()) {
      throw new IllegalArgumentException(
          String.format("%s. addToDoubleDataSet() does not match preset step! %s!=%s",
              GenSegment.class, this.step, set.getStep()));
    }
    set.ensureCapacity(set.size() + N);
    Integer temp;
    for (int i = 0; i < N; i++) {
      temp = (rand.nextInt(10000) - 5000);
      set.add(temp.doubleValue() / 100);
    }

  }

}
```