**Osci Drawing**

**Content of the task**

Oscilloscopes are mainly known for being used to visualize the course of electrical voltages. Analog oscilloscopes shoot a beam of electrons at the display screen, which lights up with fluorescence where the electrons strike.The electron beam can be redirected by two electrical signals by means of electromagnets. One signal determines the shift in the x-direction, the other in the y-direction. In the classic application, the x-signal is a sawtooth signal that lets the electron beam move "slowly" from left to right and then jumps back to the left very quickly. Because the second signal determines the y-shift (height on the screen) during this left-right movement, the voltage curve of the second signal becomes visible on the screen.

However, both signals can also be selected arbitrarily and thus the position of the electron beam can be controlled arbitrarily on the screen. However, the same two signals can also be interpreted as tones at the same time, i.e. as a stereo audio signal. For example, the voltage waveform of one signal determines the x-position on the oscilloscope and sounds for the left speaker, while the voltage waveform of the other signal manipulates the y-position of the electron beam and is output as a tone in the right speaker.

A few people have delved into creating stereo signals that both draw interesting images on an oscilloscope and sound musical at the same time. This video gives some interesting insights into it. <https://youtu.be/4gibcRfp4zA>

The main goal of this task is to implement a SignalFactory class that offers some methods that help in creating signals to ultimately display various patterns and effects on the oscilloscope screen.In order to be able to examine the generated signals well, two plotting variants are implemented beforehand. On the one hand, SignalTimePlotter implements a class that plots signals over time (similar to the classic use of an oscilloscope), on the other hand, Osci2DPlotter allows the movement of an oscilloscope's electron beam caused by a stereo signal to be visualized.So that the plotted images can also be viewed, an exporter is implemented beforehand, which can save them as PNG files.

The implemented 2D plotting works great for "static" drawings, but as soon as there is movement it is difficult to see what is happening because all electron beam positions are plotted on the plot. Therefore, an exporter is also implemented that allows stereo signals to be saved as audio files and then displayed on either a real oscilloscope or an oscilloscope emulator.

Have fun!



**General information**

* The classes to be implemented are already given as skeletons, so the tests compile from the start.
* Replace the throw new UnsupportedOperationException() statements with your implementations.
* A failed test means there is a bug in your code. Conversely, a passed test does not necessarily mean that the tested code does not contain any errors.
* All of the classes and methods required in the task must be implemented; additional auxiliary classes and methods may also be written. However, it is not possible to add further abstract methods to abstract classes.

1. **The abstract class Signal**

This task interprets a signal as follows:

* A signal describes one or more value profiles over time.
* The waveform is not saved continuously. Instead, the course is defined by the values (of the double type) that the signal assumes at certain points in time (sample points).
* Sample points are distributed equidistant in time.
* The sample rate indicates how many sample points represent one second of the signal. A higher sample rate allows signals to be described more precisely, but of course requires more storage space.
* We distinguish finite and infinite signals:
* Both are not defined for negative sample point indices.
* A finite signal of size n is defined for sample point indices from 0 to n-1.
* An infinite signal is defined for all sample point indices greater than or equal to 0.
* As already mentioned, a signal can describe several value curves. A flow is called a channel. For a signal with n channels, the channel indices 0 to n-1 are valid.

These properties are represented in the abstract class Signal in the package com.java.jpp.oscidrawing. There can be many different implementations of Signal that describe different waveforms, but the following methods are sufficient to interact with them generically:

* public abstract boolean isInfinite()

Returns whether the signal is finite or infinite. This method must be overridden by the respective implementation.

* public abstract int getSize()

If isInfinite() returns true, the return value of getSize is undefined and should be ignored. However, if isInfinite() is false, then the number of sample points of this finite signal is returned. This method must be overridden by the respective implementation.

* public abstract int getChannelCount()

Returns the number of channels in the signal. This method must be overridden by the respective implementation.

* public abstract int getSampleRate()

Returns the sample rate of the signal. This method must be overridden by the respective implementation.

* public abstract double getValueAtValid(int channel, int index)

Returns the value of the channel channel value history at the sample point index. This method can assume that channel is a valid channel index and index is a valid sample point index. The return value or the behavior of the method is not defined for a call with invalid values. This method must be overridden by the respective implementation.

* public double getDuration()

If isInfinite() returns true, the return value of getDuration is undefined and should be ignored. However, if isInfinite() is false, then this function should return the total duration of the signal. This method is not implementation dependent and can be implemented directly in the abstract class. Calculate the total duration from the number of sample points and the sample rate and return the result.

* public double getValueAt(int channel, int index)

Returns the value of the signal from channel channel at index if the signal is defined there. If channel or index are invalid, 0 should be returned instead. This method is not implementation dependent and can be implemented directly in the abstract class.

1. **Image export**

The ImageExporter class from the com.java.jpp.oscidrawing.io package should only provide a single method:

* public static boolean writeToPNG(String pathWithoutSuffix, BufferedImage img)

This method should append a ".png" to the pathWithoutSuffix and save the img to this file path as a PNG image. If anything goes wrong, e.g. the given path cannot be written to, false should be returned, otherwise true should be returned.

1. **Audio export**

The AudioExporter class from the com.java.jpp.oscidrawing.io package should provide two methods that are used to export signals as audio files.

Your Java code should not write music files directly, e.g. in MP3 or WAV format, but only create binary files that contain all values of a channel of a signal as raw float values.

To be implemented are:

* public static boolean writeChannelToFile(String path, Signal signal, int channel)

The Channel channel of signal is to be exported to the file system to path, where the extension .raw is to be appended to path. If signal is infinite, an IllegalArgumentException should be thrown.

If signal is finite, then convert the double values ​​of the channel's sample points channel to float and write the binary representation to the file in big-endian notation. For example, the DataOutputStream class with the writeFloat method can be used for this. Return false if an error occurs during this process. Otherwise return true.

* public static boolean writeStereoToFiles(String path, Signal signal)

This method is intended to write the two channels of a stereo signal to raw binary files.

If signal is infinity, an IllegalArgumentException should be thrown.

If signal does not have exactly two channels, an IllegalArgumentException should be thrown. Channel 0 of signal should be saved in the file system under path + "left.raw", Channel 1 should be saved under path + "right.raw". When saving a channel, proceed in exactly the same way as in writeChannelToFile.

Return false if an error occurs during this process. Otherwise return true.

Optional: The rest of this section describes how the raw files can be converted into valid audio files for listening or viewing on an oscilloscope (emulator). The signals to be generated later can be tested sufficiently well with the visualizations (next section).

FFmpeg is used to create the audio files. <https://ffmpeg.org/>

To create a mono audio file signal.wav from the raw file signal.raw:



-f f32be Specifies that the raw file contains 32-bit float values with big-endian byte ordering.

-ar 48000 specifies that the sample rate is 48000. This value must be adjusted to the sample rate used.

In order to create a stereo audio file, both mono signals must first be created with the above command. Then they can be merged with the following command.



To visualize a stereo signal, the audio file simply has to be opened and played in the oscilloscope emulator. <https://asdfg.me/osci/>

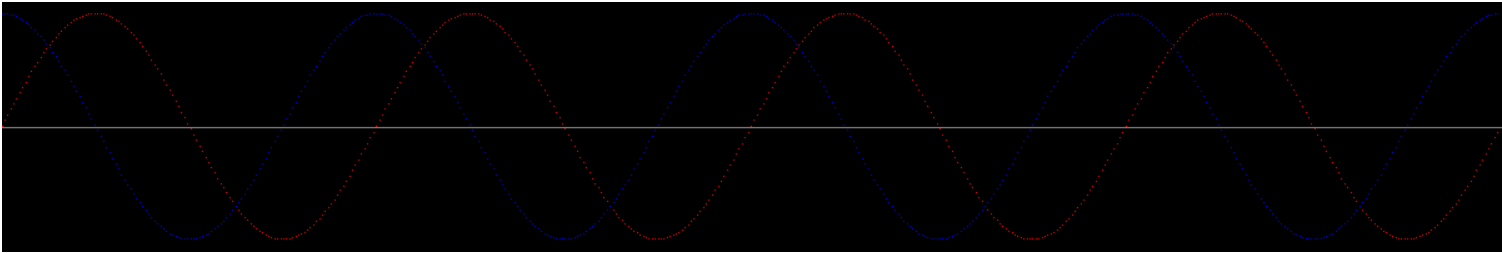
Alternatively, Audacity, an audio editing application with a graphical interface, can be used instead of the FFmpeg commands. <https://www.audacity.de/>

**4. Visualization**

**Plot waveforms over time**

The SignalTimePlotter interface in the com.java.jpp.oscidrawing.visualization package describes an interface that should allow the course of the sample point values of channels of a signal to be visualized.

As an example of such a plot, here is the progression of a sine and a cosine wave:



Write the method



to return an implementation of SignalTimePlotter. Internally, a BufferedImage of size widthxheight is to be saved, on which the signal curves can be drawn using the method.

Initially almost all pixels of the image should have the color bgcol, with the exception of the middle row of pixels, which should be colored in axiscol. If height is odd, the y-coordinate of the middle row is unique. If height is even, the "lower of the two middle rows" should be colored with axiscol (the row with a larger y-coordinate, because the y-axis runs from top to bottom in BufferedImage).

valScale specifies how the y-axis should be scaled. For example, sample points with a value of 0 are drawn in the middle row of the image. Points corresponding to the valScale value are in the top row. Points corresponding to the -valScale value are in the bottom row. Similarly, timeScale describes the scaling of the x-axis. timeScale is therefore the limit of the time interval that is to be displayed on the image. This means that the first sample point (sample index 0, at time 0) is in the first, "leftmost" pixel column. However, on the far right is the sample point that belongs to timeScale. If timeScale is now e.g. 1, exactly one second can be displayed in the image and the value belonging to the last sample point of the first second of the signal is drawn in the rightmost pixel column.

The methods requested by the interface should behave as follows:

* public int sampleIndexToImageXCoord(int sampleIndex, int sampleRate)

Returns the x-coordinate of a sample point on the image.

You can use this map function as a guide. <https://www.arduino.cc/reference/en/language/functions/math/map/> The x here corresponds to the sampleIndex.

in\_min is the smallest sample point index (ie 0).

in\_max is the largest sample point index still visible on the image (sampleRate\*timeScale-1).

out\_min is the smallest x-coordinate (that is, 0).

out\_max is the largest x coordinate (width-1).

Make sure that the calculations are carried out as floating-point number calculations and that only the end result is rounded down to the next smaller int in terms of absolute value.

* public int signalValToImageYCoord(double val)

Returns the y-coordinate of a sample point on the image.

Again, you can use the map function for orientation. <https://www.arduino.cc/reference/en/language/functions/math/map/> The x here corresponds to the val.

in\_min is the sample point value drawn at the top of the image (valScale).

in\_max is the sample point value drawn at the bottom of the image (-valScale).

out\_min is the y-coordinate of the topmost pixel (that is, 0).

out\_max is the y-coordinate of the bottom pixel (height-1).

Make sure that the calculations are carried out as floating-point number calculations and that only the end result is rounded down to the next smaller int in terms of absolute value.

* public void drawSignalAt(Signal signal, int channel, int index, Color col)

This method should draw the sample point with the index index from the channel channel of signal on the image in the color col. To do this, determine the pixel coordinates of the sample point. If these lie within the image, the corresponding pixel should be colored with col. If they are outside the frame, this method should do nothing (nor throw any exceptions).

* public void drawSignal(Signal signal, int channel, Color col)

This method should draw the course of the channel of signal in the color col on the plot.

* public void drawSignal(Signal signal, Color... colors)

This method should draw all channels of signal in the picture. The i-th channel is to be plotted in the color colors[i].

If the number of colors passed does not match the number of channels in signal, an IllegalArgumentException should be thrown.

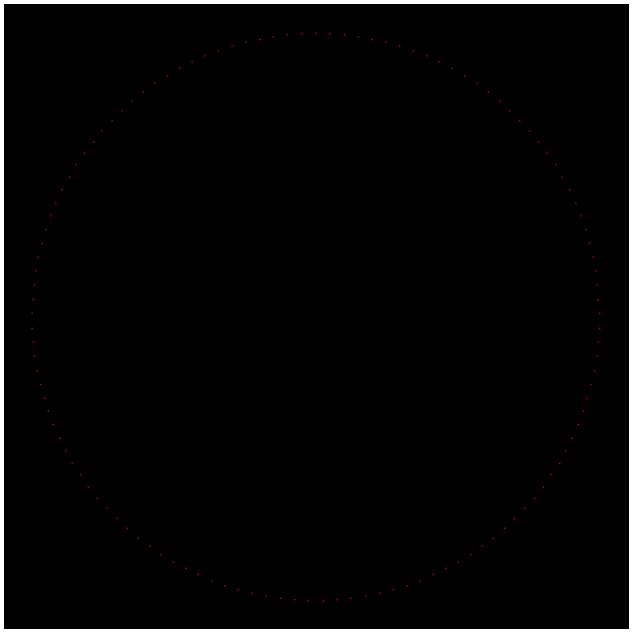
* public BufferedImage getImage()

This method returns the image that was drawn on.

**2D oscilloscope plot of a stereo signal**

The Osci2DPlotter interface in the com.java.jpp.oscidrawing.visualization package describes an interface that should allow visualizing all the positions that the electron beam of an oscilloscope takes when the two channels of the stereo signal are used as x and y alignment .

For example, the sine and cosine waves from the previous section's example result in a circle:



Write the method



to return an implementation of Osci2DPlotter. Internally, a BufferedImage of size sizexsize is to be stored on which to draw.

At the beginning all pixels of the image should have the color bgcol. scale specifies the scaling, which is identical for the x and y axes.

The methods requested by the interface should behave as follows:

* public int signalValToImageXCoord(double val)

Returns the x-coordinate of a sample point on the image.

Again, you can use the map function for orientation. <https://www.arduino.cc/reference/en/language/functions/math/map/> The x here corresponds to the val.

in\_min is the sample point value drawn on the far left of the image (-scale).

in\_max is the sample point value drawn at the far right of the image (scale).

out\_min is the x-coordinate of the leftmost pixel (i.e. 0).

out\_max is the x-coordinate of the rightmost pixel (size-1).

Make sure that the calculations are carried out as floating-point number calculations and that only the end result is rounded down to the next smaller int in terms of absolute value.

* public int signalValToImageYCoord(double val)

Returns the y-coordinate of a sample point on the image.

Again, you can use the map function for orientation. <https://www.arduino.cc/reference/en/language/functions/math/map/> The x here corresponds to the val.

in\_min is the sample point value drawn at the top of the image (scale).

in\_max is the sample point value drawn at the far right of the image (-scale).

out\_min is the y-coordinate of the top pixel (that is, 0).

out\_max is the x-coordinate of the bottom pixel (size-1).

Make sure that the calculations are carried out as floating-point number calculations and that only the end result is rounded down to the next smaller int in terms of absolute value.

* public void drawSignalAt(Signal signal, int index, Color col)

Draws a point for the pair of values ​​that the stereo signal takes on at the sample point index.

If signal does not have exactly two channels, an IllegalArgumentException should be thrown.

Otherwise use the value of the 0th channel to determine the x-coordinate of the pixel to be colored and the value of the 1st channel to calculate the y-coordinate. If the specific pixel is inside the image, it should be changed to the color col in the image. If the pixel is outside, the method should do nothing (not even throw an exception).

* public void drawSignal(Signal signal, Color col)

This method should draw all pairs of values ​​of the stereo signal into the image.

If signal is an infinite signal, an IllegalArgumentException should be thrown.

Otherwise, draw the value pairs at all sample points with the color col in the image as described in drawSignalAt.

* public BufferedImage getImage()

This method returns the image that was drawn on.

**5. Signal generation**

The SignalFactory class in the com.java.jpp.oscidrawing.generation package offers a number of methods for generating signals. It often makes sense to use other factory methods to implement a factory method.

Use the previously implemented visualization tools to examine your signals.

* public static Signal fromValues(double[] signalData, int sampleRate)

This method allows a set of double values, given as an array, to be represented as a signal.

If sampleRate is not positive, an IllegalArgumentException should be thrown.

The signal to be generated should be finite, contain as many sample points as signalData values ​​and have exactly one channel. The sample rate is given by sampleRate. The ith sample point of Channel 0 should correspond to the ith value of signalData.

* public static Signal wave(DoubleUnaryOperator function, double frequency, double duration, int sampleRate)

This method should generate periodic signals with a period length of 2π, for example sine or cosine signals.

If frequency is not positive, an IllegalArgumentException should be thrown.

If duration is not positive, an IllegalArgumentException should be thrown.

If sampleRate is not positive, an IllegalArgumentException should be thrown.

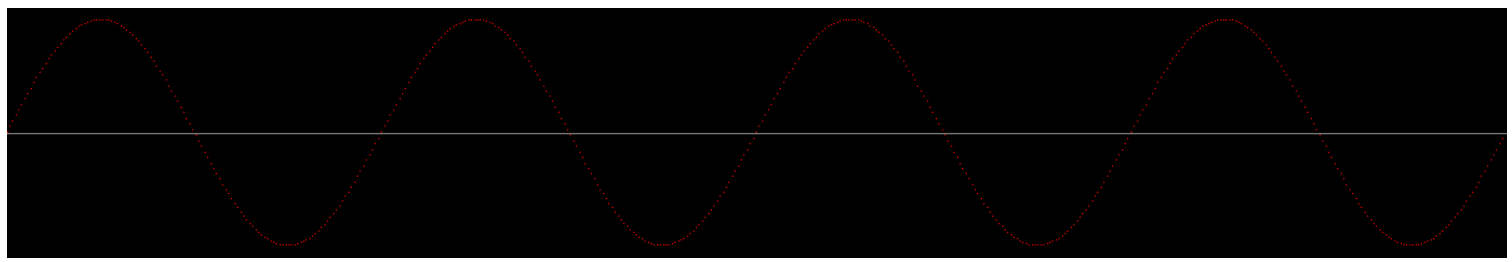
The signal to be generated should be finite, with size sampleRate\*duration, have exactly one channel and have the sample rate sampleRate. To calculate the signal values, proceed as follows:

* Calculate the increment step. This describes how stretched or compressed values ​​must be used in the function method in order to achieve the desired frequency.

It is step = (frequency\*2π)/sampleRate.

* Calculate the ith sample point of channel 0 as function.applyAsDouble(i\*step).

For example, for the call wave(Math::sin, 4, 1, 500) the plot of the signal looks like this:



* public static Signal rampUp(double duration, int sampleRate)

This method should return a signal that has "the course of a ramp", i.e. increases linearly from 0 to 1 over the duration duration.

If duration is not positive, an IllegalArgumentException should be thrown.

If sampleRate is not positive, an IllegalArgumentException should be thrown.

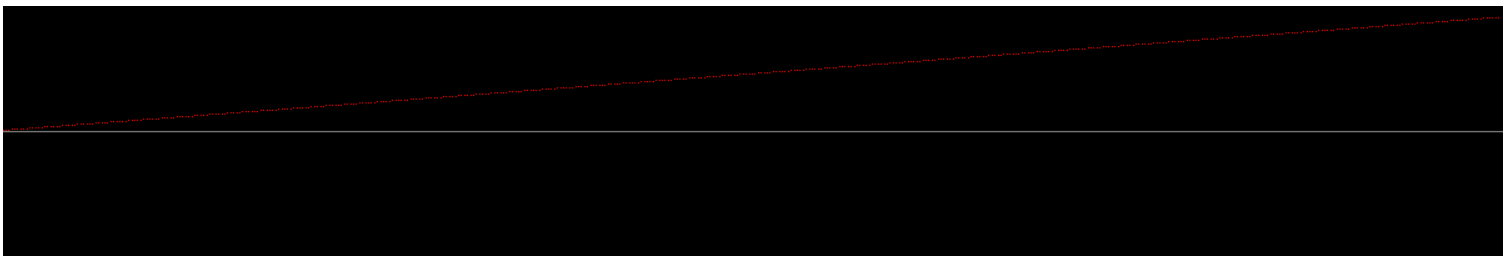
The signal to be generated should be finite, with size sampleRate\*duration, have exactly one channel and have the sample rate sampleRate.

If the magnitude of the signal is less than 2, e.g. because the signal duration is short and the sample rate is low, an IllegalArgumentException should be thrown instead.

To calculate the signal values, proceed as follows:

* In the following, let samples be the number of sample points (i.e. the size of the signal).
* The i-th sample point should have the value i/(samples-1).

For example, for the rampUp(1, 500) call, the plot of the signal looks like this:



* public static Signal combineMonoSignals(List<Signal> signals)

This method is intended to combine multiple mono signals (signals with exactly one channel) into a single signal with multiple channels.

If signals is null, a NullPointerException should be thrown.

If signals is empty, an IllegalArgumentException should be thrown.

If signals contains signals with different sample rates, an IllegalArgumentException should be thrown.

If signals contains a non-mono signal, an IllegalArgumentException should be thrown.

The combined signal should be as large as the shortest mono signal. Consequently, this means that the new signal is infinity only if all signals in signals are infinity. The signal has as many channels as there are signals in Signal objects. The new sample rate is the same as that of the signals from signals. The new value of Channel channel in place index should be the same as the value of the signal from signals with index channel from Channel 0 in place index.

* public static Signal combineMonoSignals(Signal... signals)

This method is intended to work the same as public static Signal combineMonoSignals(List<Signal> signals), except that the mono signals are passed as a vararg argument.

* public static Signal stereoFromMonos(Signal left, Signal right)

This method is intended to combine two mono signals into one stereo signal (signal with two channels). The combination works the same as in combineMonoSignals, but because of the importance of stereo signals for drawing on the oscilloscope, this method is implemented with an explicit name.

* public static Signal extractChannels(Signal source, int... channels)

This method should create a new signal whose channels are composed of the channels of source by the mapping given in channels.

Accordingly, the new signal also inherits infinity, size and sample rate from the original signal. The new signal has "length of channels"-many channels. The new value of Channel channel in place index should be the same as the value of the old signal of Channel channels[channel] in place index.

There are a few errors to watch out for:

If source is null, a NullPointerException should be thrown.

If channels contains values ​​that are not valid channel indices in source, an IllegalArgumentException should be thrown. Example:

Let s3 be a signal with three channels, then gives

* extractChannels(s3, 0) a mono signal that has channel 0 of s3 as channel 0.
* extractChannels(s3, 2) a mono signal that has the channel 2 of s3 as channel 0.
* extractChannels(s3, 2, 1, 0) a signal with three channels, namely the channels of s3, but in reverse order.
* public static Signal circle(double frequency, double duration, int sampleRate)

This method is intended to return a signal that draws a circle on the scope.

If frequency is not positive, an IllegalArgumentException should be thrown.

If duration is not positive, an IllegalArgumentException should be thrown.

If sampleRate is not positive, an IllegalArgumentException should be thrown.

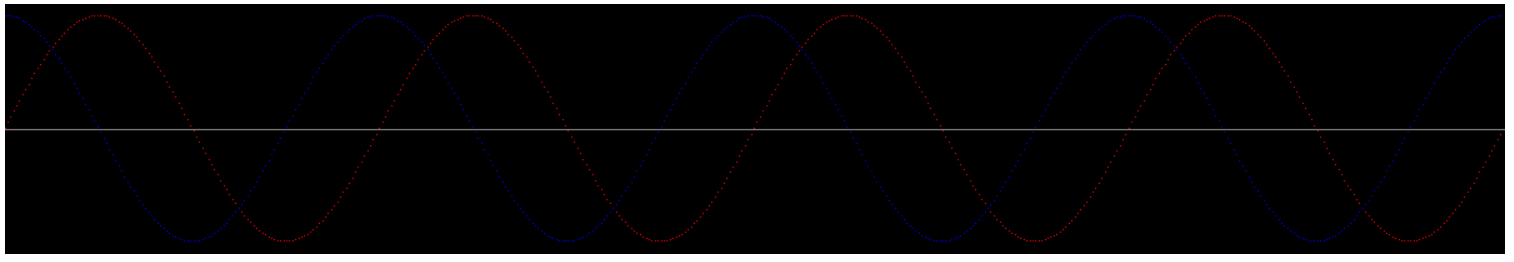
To draw on the oscilloscope, the signal must be stereo. A circle results exactly when one channel contains a sine wave, while the other represents a cosine wave with the same frequency. In principle, the channels can be swapped, but here the sine wave should be in channel 0.

Frequency should be chosen for the frequency of the two waves. This means that the electron beam of the oscilloscope travels the circle frequently per second.

The amplitude of the waves determines the radius of the circle. In this case, the "Default" amplitude of 1 should be used.

Furthermore, probably as a matter of course, the signal should be finite, have a size of duration\*sampleRate and a sample rate of sampleRate.

For example, for the call circle(4, 1, 500), the plot of the signal looks like this:



The red plot shows channel 0 and the blue plot shows channel 1.

* public static Signal cycle(Signal signal)

Now it's finally about generating an infinite signal. This method is intended to return an infinite signal that "loops" the given signal.

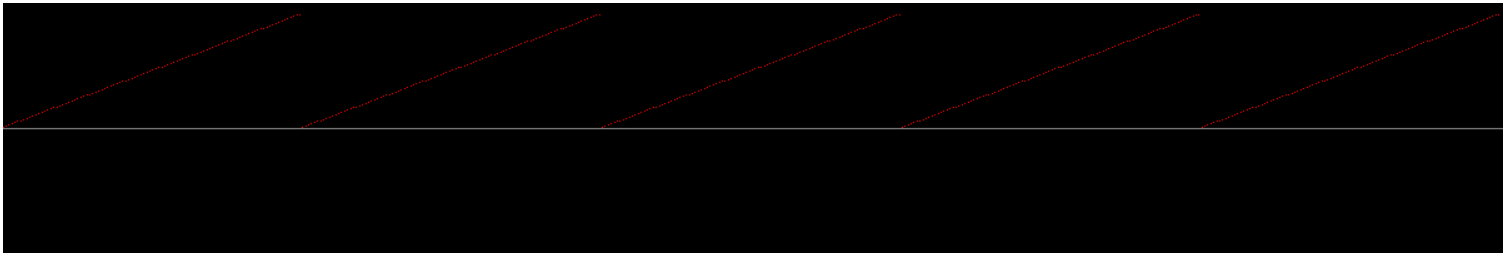
If signal is null, a NullPointerException should be thrown.

The new signal should be infinity. Accordingly, the behavior of getSize is not defined and can be chosen arbitrarily. The number of channels and the sample rate are taken from signal.

signal can also be an infinite signal. In this case the value for the channel channel and sample point index can be derived directly from signal.

If signal is finite and index is greater than or equal to the size of signal, index must be modified by modulo calculation so that it lies between 0(inclusive) and the size of signal(exclusive).

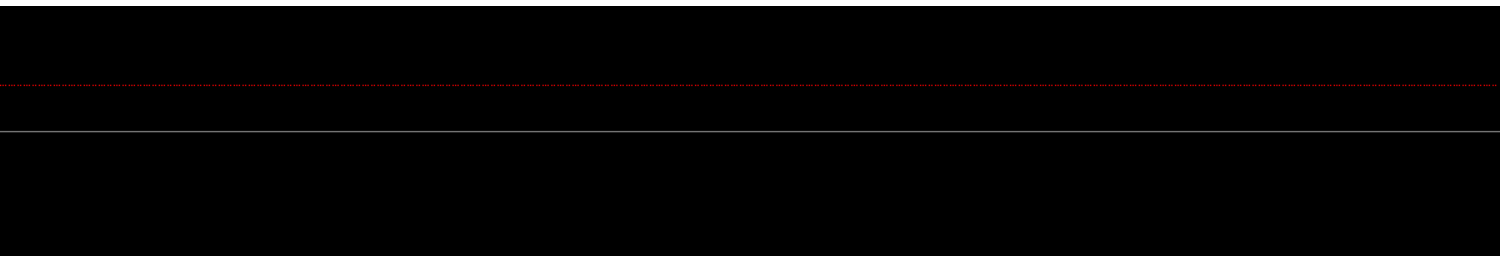
For example, for the call cycle(rampUp(0.2, 500)) the plot of the first second of the signal looks like this:



* public static Signal infiniteFromValue(double value, int sampleRate)

This method is intended to generate an infinite signal that has a channel that has the value value at each sample point.

The new signal is therefore infinite, the behavior of getSize is undefined, the signal has exactly one channel, sampleRate is to be selected as the sample rate and the signal has the value value at every valid point.

Accordingly, a plot of this signal is unspectacular (with value = 0.4):

* public static Signal take(int count, Signal source)

This method is intended to truncate signals to the given length count.

If count is negative, an IllegalArgumentException should be thrown.

The new signal is finite with size count. Number of channels and sample rate are taken from source.

If the size of source is greater than count, the source is truncated. For each channel channel and index, the value can be determined directly from source.

If source has fewer sample points than count, all values ​​for sample point indices greater than or equal to the size of source shall be 0 (zero-pad source to the new size).

* public static Signal drop(int count, Signal source)

This method is intended to discard the beginning of a signal, namely the first count sample points.

If count is negative, an IllegalArgumentException should be thrown.

If source is infinite, the resulting signal is still infinite, and the behavior of getSize is undefined. If source is finitely large, source's size is reduced by count, or possibly to 0 (a signal of size 0 isn't very useful, but it's legal).

Number of channels and sample rate are taken from source. The value of Channel channel at sample point index should be the same as source of Channel channel at sample point index+count.

* public static Signal transform(DoubleUnaryOperator function, Signal source)

A DoubleUnaryOperator is an interface that has a method



demands.

The given implementation of this interface function shall be used to transform each signal value of source.

If function is null, a NullPointerException should be thrown.

If source is null, a NullPointerException should be thrown.

(In)infinity, size, number of channels and sample rate should be identical to those of source. The value of Channel channel at sample point index shall be the value of source of Channel channel at sample point index inserted in applyAsDouble of function.

* public static Signal scale(double amplitude, Signal source)

This method should return a signal that stretches or compresses each signal value from source by amplitude. If source is null, a NullPointerException should be thrown.

(In)infinity, size, number of channels and sample rate should be identical to those of source. The value of Channel channel at sample point index shall be the value of source of Channel channel at sample point index multiplied by amplitude.

* public static Signal reverse(Signal source)

This method should return a signal containing the sample points from source is in reverse order.

If source is null, a NullPointerException should be thrown.

If source is infinite, an IllegalArgumentException should be thrown.

Consequently, the new signal should also be finite. Size, number of channels and sample rate should be identical to those of source. The value of Channel channel at sample point index shall be the value of source of Channel channel at sample point ("Size of source"-1-index).

* public static Signal rampDown(double duration, int sampleRate)

This method should behave like rampUp, with the difference that the values ​​in the only channel should decrease linearly from 1 to 0.

* public static Signal merge(BiFunction<Double, Double, Double> function, Signal s1, Signal s2)

A BiFunction<Double, Double, Double> is an interface that the method



demands.

The given implementation of the interface function should be used to combine the signal values ​​of the signals s1 and s2.

If s1 is null, a NullPointerException should be thrown.

If s2 is null, a NullPointerException should be thrown.

If function is null, a NullPointerException should be thrown.

If s1 and s2 have different sample rates, an IllegalArgumentException should be thrown.

If s1 and s2 have different channel numbers, an IllegalArgumentException should be thrown.

If the signals are of different sizes, the smaller size is chosen as the new size for the new signal. Consequently, the resulting signal is infinity only if both s1 and s2 are infinity. The number of channels and the sample rate should be taken from s1 and s2.

The new signal should have the result of function in channel channel at sample point index, if the values ​​of s1 and s2 are inserted from channel channel and sample point index, respectively (value of s1 as first argument.)

* public static Signal add(Signal s1, Signal s2)

This method shall merge the signals s1 and s2 by adding the signal values.

If s1 is null, a NullPointerException should be thrown.

If s2 is null, a NullPointerException should be thrown.

If s1 and s2 have different sample rates, an IllegalArgumentException should be thrown.

If s1 and s2 have different channel numbers, an IllegalArgumentException should be thrown.

If the signals are of different sizes, the smaller size is chosen as the new size for the new signal. Consequently, the summed signal is infinity only if both s1 and s2 are infinity. The number of channels and the sample rate should be taken from s1 and s2.

* public static Signal mult(Signal s1, Signal s2)

This method shall merge the signals s1 and s2 by multiplying the signal values.

If s1 is null, a NullPointerException should be thrown.

If s2 is null, a NullPointerException should be thrown.

If s1 and s2 have different sample rates, an IllegalArgumentException should be thrown.

If s1 and s2 have different channel numbers, an IllegalArgumentException should be thrown.

If the signals are of different sizes, the smaller size is chosen as the new size for the new signal. Consequently, the summed signal is infinity only if both s1 and s2 are infinity. The number of channels and the sample rate should be taken from s1 and s2.

* public static Signal append(List<Signal> signals)

This method is intended to merge a list of signals into one signal by concatenating the signals in a given order. If signals is null, a NullPointerException should be thrown.

If signals is empty, an IllegalArgumentException should be thrown.

If a signal from signals that is not the last signal is infinity, an IllegalArgumentException should be thrown.

If the signals in signals do not all have the same sample rates, an IllegalArgumentException should be thrown.

If the signals in signals do not all have the same number of channels, an IllegalArgumentException should be thrown.

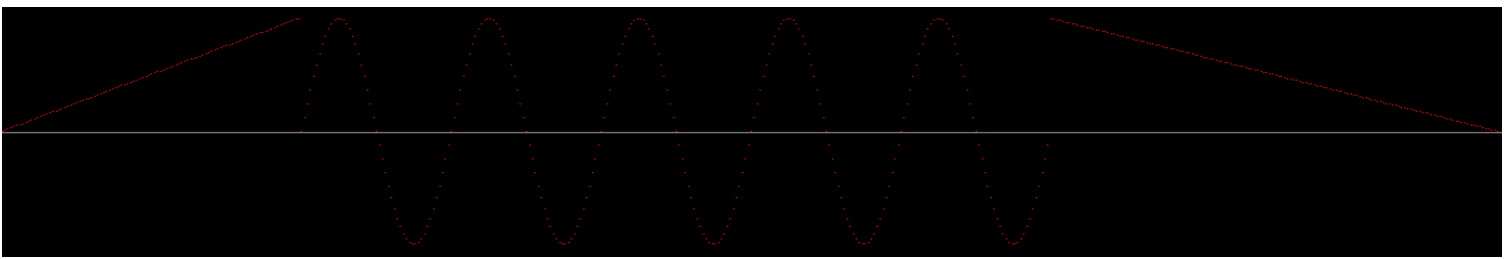
The new signal is said to be infinity if and only if the last signal in signals is infinity. If not, the signal is finite, with magnitude being the sum of the magnitudes of the signals from signals.

The number of channels and the sample rate should be accepted.

Let s\_i with i = 0,1,2,...,n be the size of the signal at position i The value of Channel channel at position index should be:

* If index<s\_0, then value of first signal from channel channel instead of index
* Else, if index<s\_0+s\_1, then value of second signal from channel channel instead of index-s\_0
* Else, if index<s\_0+s\_1+s\_2, then value of third signal from channel channel in place index-s\_0-s\_1
* ...
* Else, the value of the last signal from Channel channel in place index-s\_0-s\_1-...-s\_(n-1)

For example, for the call append(List.of(rampUp(0.2, 500), wave(Math::sin, 10, 0.5, 500), rampDown(0.3, 500))) the plot of the signal looks like this:



* public static Signal append(Signal... signals)

This method is intended to work the same as public static Signal append(List<Signal> signals), except that the signals are passed as a vararg argument.

* public static Signal translate(List<Double> distances, Signal signal)

This method should shift the signal by a fixed offset given by distances. Each entry in distances describes the displacement in a channel. If signal is null, a NullPointerException should be thrown.

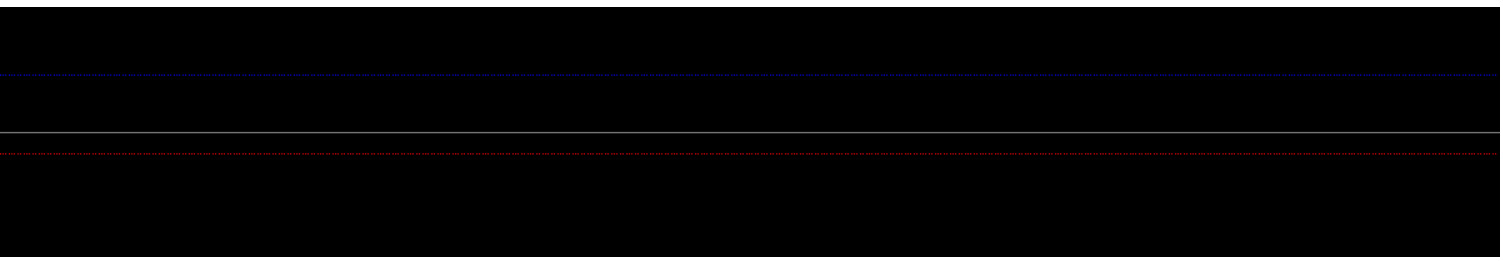
If distances is null, a NullPointerException should be thrown.

If distances does not have as many elements as signal Channels, an IllegalArgumentException should be thrown.

The new signal should inherit finiteness, size, number of channels and sample rate from signal.

The value of Channel channel at sample point index should be equal to the value of signal of Channel channel at sample point index added with the value given in the channel-th place in distances.

For example, for the translate(List.of(-0.2, 0.5), stereoFromMonos(infiniteFromValue(0, 500), infiniteFromValue(0, 500))) call, the plot of the first second of the signal looks like this:



The red plot shows channel 0 and the blue plot shows channel 1.

The last method to be implemented in SignalFactory is to generate a stereo signal that moves the scope's electron beam along a path at constant speed. To do this, two helper classes are to be implemented first.

**The Point class**

The Point class from the com.java.jpp.oscidrawing.generation.pathutils package is used to manage points in two-dimensional space. The following methods are to be implemented:

* public Point(double x, double y)

Creates a new Point object.

* public double getX()

Returns the x value obtained in the constructor.

* public double getY()

Returns the y value obtained in the constructor.

* public double distanceTo(Point p)

Returns the Euclidean distance to point p.

* public String to String()

Returns a textual representation of the point in the following form:



where <x> and <y> should be replaced with the appropriate attribute values.

* public Point interpolateTo(Point p, double factor)

Let xd be the vector pointing from this point to the point p. The returned point should be "this point"+factor\*xd. In particular, the following then applies:

If factor is 0, the returned point should have the same coordinates as this point.

If factor is 1, the returned point should have the same coordinates as point p.

**The Line class**

The Line class from the com.java.jpp.oscidrawing.generation.pathutils package is used to manage lines between two points in two-dimensional space. The following methods are to be implemented:

* public Line(Point p1, Point p2)

Creates a new Line object.

* public double getStart()

Returns the p1 value obtained in the constructor.

* public double getEnd()

Returns the p2 value obtained in the constructor.

* public doublelength()

Returns the length of the line.

* public Point getPointAt(double percentage)

Returns the point that is a percentage of the way from the start point to the end point. In particular, the following then applies:

If percentage is 0, the returned point should have the same coordinates as the starting point.

If percentage is 1, the returned point should have the same coordinates as the end point.

public String to String()

Returns a textual representation of the line in the form:



where <p1> and <p2> should be replaced with the appropriate attribute values.

With these tools, the last method of the SignalFactory can now be implemented:

* public static Signal fromPath(List<Point> points, double frequency, int sampleRate)

This method is intended to return a stereo signal that sweeps the oscilloscope's electron beam along the points contained in points . The speed should be kept as constant as possible. frequency specifies how many times per second the signal could be repeated. However, the returned signal should travel along the path only once.

If frequency is not positive, an IllegalArgumentException should be thrown.

If sampleRate is not positive, an IllegalArgumentException should be thrown.

If points is empty or contains only a single point, an IllegalArgumentException should be thrown.

Proceed as follows to generate a corresponding signal:

* Calculate the duration as the reciprocal of the frequency.
* Create lines containing all the lines of the path (this list is then 1 less than points).
* Create lineLengths, which stores the length of the line for each line from lines.
* Calculate pathLength as the sum of all line lengths.
* Create a list normalizedLineLengths that stores the length of the line divided by pathLength for each line of lines.
* Create a pointsPerLine list that stores for each line from lines how many sample points this line should have.

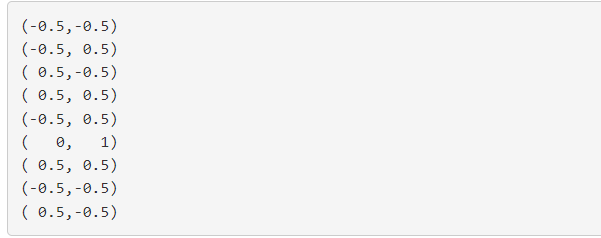
Let l be a Line and nLen the normalized length of l. Then the duration lDur that the electron beam spends on this line can be calculated, i.e. duration\*nLen. Furthermore, lDur\*sampleRate can be used to calculate how many sample points the line should have. However, the result is a floating point value. Round this down and save the result as an integer in the pointsPerLine list.

* Create a list of interpolatedPoints containing all sample points stored as Point. Start with an empty list. For each line from lines, do:
* Let numPoints be the number of sample points in the line.
* Create a list indices containing the integers from 0(inclusive) to numPoints(exclusive).
* Create a list of doubles named lineProgress containing the elements from indices divided by numPoints.
* Create a list called interpolatedPointsOfLine that contains the points that getPointAt returns on the line, using the elements of lineProgress.
* Add all elements of interpolatedPointsOfLine to the end of interpolatedPoints.

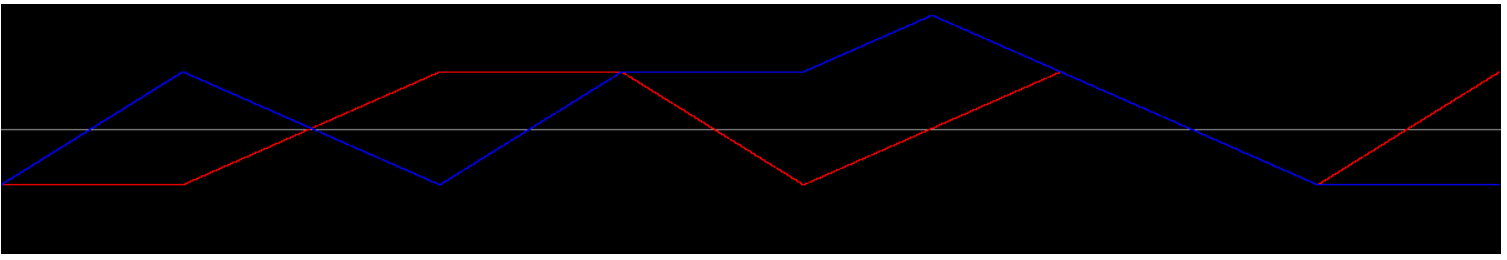
Now interpolatedPoints contains all information necessary for the signal to be generated.

Return a stereo signal whose channel 0 contains the x-coordinates of the points and whose channel 1 contains the y-coordinates of the points.

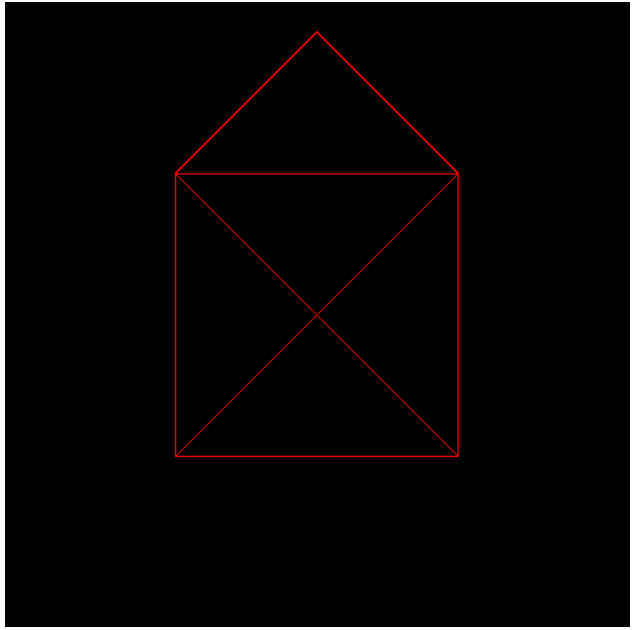
For example, the points result in



the following signal (channel 0 -> red, channel 1 -> blue):



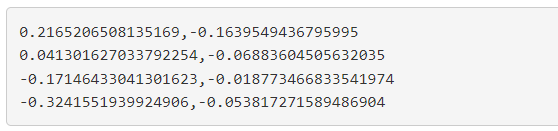
The following picture then emerges on the oscilloscope:



**Path import**

Since it is relatively inconvenient to create a list of points in the code in order to generate signals from them, an importer should be written that can read points from a text format.

The format for points is very simple and looks something like this:



Each line describes a point. First, a floating-point number specifies the x-coordinate of the point, followed by a comma, before another floating-point number specifies the y-coordinate of the point.

Implement these two methods in the PathImporter class in the com.java.jpp.oscidrawing.io package:

* public static Optional<List<Point>> fromString(List<String> lines)

This method receives a list of strings, each of which should represent a line of a text file.

If all lines conform to the format just described, a list of the items described by the lines, wrapped in an optional, should be returned.

If one of the lines does not correspond to the format described and therefore cannot be translated into a point, an empty optional should be returned instead.

* public static Optional<List<Point>> fromFile(String path)

This method gets a path to a file with path. The content of this file is to be read in and each line is to be translated into a point.

If anything goes wrong, e.g. the file doesn't exist, can't be read, or the content doesn't match the required format, an empty Optional should be returned.

Otherwise, an Optional-wrapped list of Points should be returned that match the contents of the file.

In the resources folder you will find the path file samplePath.txt, which you can use to test your importer.

**Optional: Generate your own signal**

The myCoolSignal method still exists in the SignalFactory. This method must be present for the tests to compile, but it is sufficient to leave the return null default implementation unchanged.

Anyone who had fun creating signals and would like to use the factory methods in a meaningful way is invited to create and give back a creative signal here.

The "submissions" will be evaluated at the end of the internship.

If you read files while doing this, place them in the resources folder and read them in as described here <https://mkyong.com/java/java-read-a-file-from-resources-folder/> (mark resources as Resources Root).