Clemens Czermak 0126610 532 e0126610@student.tuwien.ac.at
Maximilian Limbeck-Lilienau 9305920 532 a9305920@unet.univie.ac.at
for 2.0 LU Multimedia 1: Daten und Formate

Visualizing Data Vectors

1. Instructions

We had to implement JAVA classes for visualizing data vectors according to the following pre-defined criteria.

A Java vector of numbers shall be represented visually by a diagram in the form of a canvas object. 5 types of diagrams must be implemented with Java2D:

- bar chart (both horizontal and vertical)
- line chart (both horizontal and vertical)
- scatter diagram
- pie chart
- surface diagram

Furthermore, there should be a class that merges two diagrams into one by scaling them and putting them side by side.

Each diagram should allow the following settings:

- height and breadth in pixels
- customized colors through a vector of color values
- width of bars, lines, points
- color distribution in the surface diagram (black and white, rainbow)

All diagrams shall be derived from the class org.vizir.Diagram, which must feature at least the following methods:

- get/set methods for the properties
- a repaint method to refresh the display

All classes must be documented in English with Javadoc API. JUnit test methods have to be implemented, according to the test environment by JUnit.org .

The demands were all met by our implementation. We shall consider it in detail by presenting the UML model, and then we will detail the implementation of each class, giving some necessary piece of advice to consider for a proper use of those classes.

2. UML model

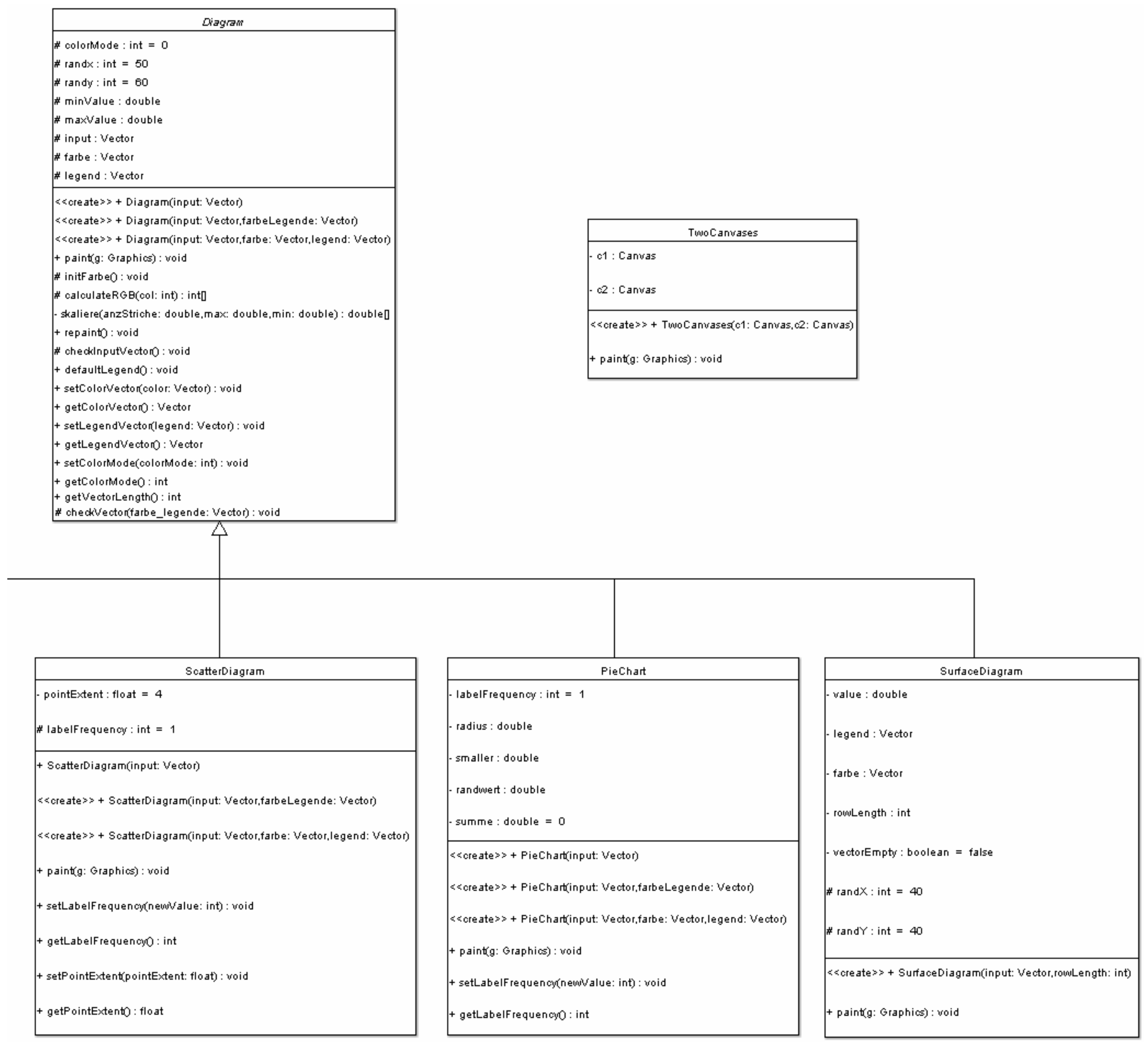
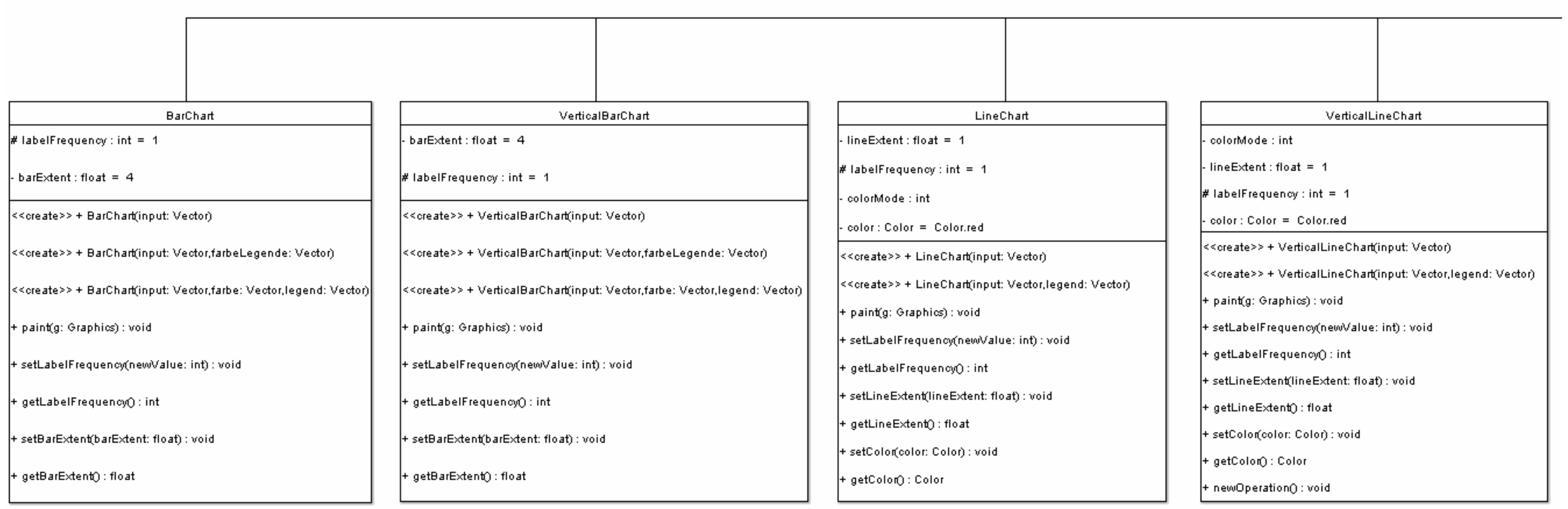


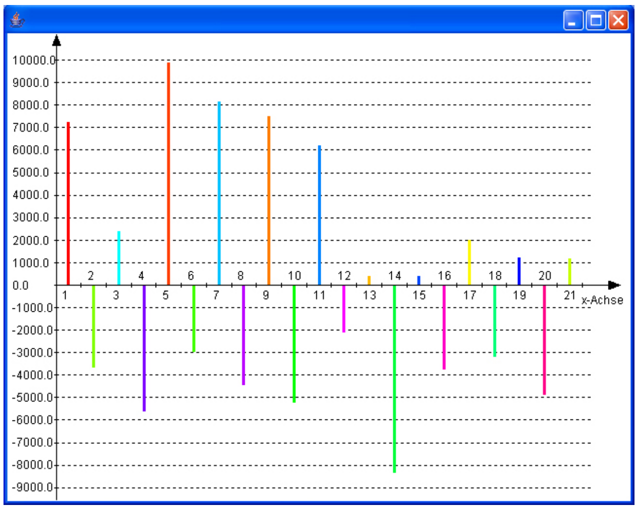
Diagram
# colorMode : int = 0
# randx : int = 50
# randy : int = 60
# minValue : double
# maxValue : double
# input : Vector
# farbe : Vector
# legend : Vector
<<create>> + Diagram(input: Vector)
<<create>> + Diagram(input: Vector,farbeLegende: Vector)
<<create>> + Diagram(input: Vector,farbe: Vector,legend: Vector)
+ paint(g: Graphics) : void
# initFarbe() : void
# calculateRGB(col: int) : int[]
- skaliere(anzStriche: double,max: double,min: double) : double[]
+ repaint() : void
# checkInputVector() : void
+ defaultLegend() : void
+ setColorVector(color: Vector) : void
+ getColorVector() : Vector
+ setLegendVector(legend: Vector) : void
+ getLegendVector() : Vector
+ setColorMode(colorMode: int) : void
+ getColorMode() : int
+ getVectorLength() : int
# checkVector(farbe_legende: Vector) : void
TwoCanvases
- c1 : Canvas
- c2 : Canvas
<<create>> + TwoCanvases(c1: Canvas,c2: Canvas)
+ paint(g: Graphics) : void
ScatterDiagram
- pointExtent : float = 4
# labelFrequency : int = 1
+ ScatterDiagram(input: Vector)
<<create>> + ScatterDiagram(input: Vector,farbeLegende: Vector)
<<create>> + ScatterDiagram(input: Vector,farbe: Vector,legend: Vector)
+ paint(g: Graphics) : void
+ setLabelFrequency(newValue: int) : void
+ getLabelFrequency() : int
+ setPointExtent(pointExtent: float) : void
+ getPointExtent() : float
PieChart
- labelFrequency : int = 1
- radius : double
- smaller : double
- randwert : double
- summe : double = 0
<<create>> + PieChart(input: Vector)
<<create>> + PieChart(input: Vector,farbeLegende: Vector)
<<create>> + PieChart(input: Vector,farbe: Vector,legend: Vector)
+ paint(g: Graphics) : void
+ setLabelFrequency(newValue: int) : void
+ getLabelFrequency() : int
SurfaceDiagram
- value : double
- legend : Vector
- farbe : Vector
- rowLength : int
- vectorEmpty : boolean = false
# randX : int = 40
# randY : int = 40
<<create>> + SurfaceDiagram(input: Vector,rowLength: int)
+ paint(g: Graphics) : void


BarChart
# labelFrequency : int = 1
- barExtent : float = 4
<<create>> + BarChart(input: Vector)
<<create>> + BarChart(input: Vector,farbeLegende: Vector)
<<create>> + BarChart(input: Vector,farbe: Vector,legend: Vector)
+ paint(g: Graphics) : void
+ setLabelFrequency(newValue: int) : void
+ getLabelFrequency() : int
+ setBarExtent(barExtent: float) : void
+ getBarExtent() : float
VerticalBarChart
- barExtent : float = 4
# labelFrequency : int = 1
<<create>> + VerticalBarChart(input: Vector)
<<create>> + VerticalBarChart(input: Vector,farbeLegende: Vector)
<<create>> + VerticalBarChart(input: Vector,farbe: Vector,legend: Vector)
+ paint(g: Graphics) : void
+ setLabelFrequency(newValue: int) : void
+ getLabelFrequency() : int
+ setBarExtent(barExtent: float) : void
+ getBarExtent() : float
LineChart
- lineExtent : float = 1
# labelFrequency : int = 1
- colorMode : int
- color : Color = Color.red
<<create>> + LineChart(input: Vector)
<<create>> + LineChart(input: Vector,legend: Vector)
+ paint(g: Graphics) : void
+ setLabelFrequency(newValue: int) : void
+ getLabelFrequency() : int
+ setLineExtent(lineExtent: float) : void
+ getLineExtent() : float
+ setColor(color: Color) : void
+ getColor() : Color
VerticalLineChart
- colorMode : int
- lineExtent : float = 1
# labelFrequency : int = 1
- color : Color = Color.red
<<create>> + VerticalLineChart(input: Vector)
<<create>> + VerticalLineChart(input: Vector,legend: Vector)
+ paint(g: Graphics) : void
+ setLabelFrequency(newValue: int) : void
+ getLabelFrequency() : int
+ setLineExtent(lineExtent: float) : void
+ getLineExtent() : float
+ setColor(color: Color) : void
+ getColor() : Color
+ newOperation() : void

3. Implementation

We shall try and describe our implementation in detail within a reasonable scope. If you seek a shorter manual or a method summary, please refer to the Javadoc documentation (*index.html* in the folder *doc*).

Each diagram class (***BarChart***, ***VerticalBarChart***, ***LineChart***, ***VerticalLineChart***, ***ScatterDiagram***, ***PieChart*** and ***SurfaceDiagram***) is derived from our abstract class ***Diagram***, itself extending ***Canvas*** and therefore ***Component***. Those public classes will graphically display a set of input numbers, accepting **positive and negative double values**. Negative values will be displayed on the *BarChart*, *VerticalBarChart*, *LineChart*, *VerticalLineChart* and *ScatterDiagram* below the x-axis; on the *PieChart* negative values will be taken as their absolute value; on the *SurfaceDiagram* a color is assigned to each value from the lowest to the highest (e.g. -5.23 is lower than -2, which is lower than 4.00).



BarChart with positive and negative values

The diagrams benefit from visual features such as an automatically generated reference scale with dotted lines (*BarChart, VerticalBarChart, LineChart, VerticalLineChart, ScatterDiagram*) or a color legend (*PieChart*, *SurfaceDiagram*).

You can create a diagram (apart from the surface diagram) in three ways:

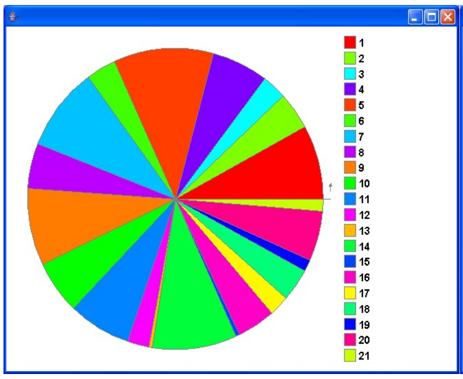
1. By passing the constructor a **Java Vector of Double objects**, each corresponding to a value then visualized by the diagram. The default legend will be the position of each double value in the input vector (1,2,3, ...). We will discuss the default color mode later.
   E.g.: *public BarChart(Vector input)*
2. By passing the constructor as first argument a **Vector of Double objects**, and as second argument **either a Vector of only Color objects or a Vector of only String objects** for the legend. *LineChart* and *VerticalLineChart* will only accept a legend vector.
   E.g.: *public PieChart(Vector input, Vector farbeLegende)*
3. By passing the constructor as first argument a **Vector of Double objects**, as second argument a **Vector of Color objects** and as third argument a **Vector of String objects** for the legend. *LineChart* and *VerticalLineChart* do not implement this constructor.
   E.g.: *public ScatterDiagram(Vector input, Vector farbe, Vector legend)*

The ***SurfaceDiagram*** represents an exception. It has only one constructor:
public SurfaceDiagram(Vector input, int rowLength)
the first argument being the **Java Vector of Double objects,** and the second argument an **integer** indicating the length of each horizontal row of elements. For example, a surface diagram with an input vector containing 9 Double objects and an integer *rowLength* of 4 will result in a colored surface matrix of 3x4 cells (3 rows, 4 columns); the first 4 values will be shown in the first row, from left to right, element 5 through 8 in the second row, element number 9 in the third row on the left. The 3 last cells in the third row will remain void (or barred if the surface diagram is in black and white).

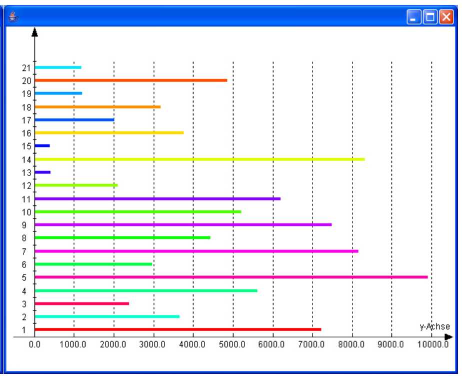
Except for the surface diagram, each input value has its corresponding color and legend value. If you decide to construct a diagram with only double values, the color and legend vectors will be initialized to their default values. Furthermore, you can change the **color mode** (*public int getColorMode()* , *public void setColorMode(int colorMode)*), thus re-initializing the color vector, or even pass your own color or legend vectors to the diagram:
public Vector getLegendVector() and *public void setLegendVector(Vector legend)*
public Vector getColorVector() and *public void setColorVector(Vector color)*
Naturally, you can go back to the default enumeration of elements in the legend through the method *public void defaultLegend()*.
You can also set the frequency of legend tags to be displayed in all diagrams except the *SurfaceDiagram*. If you set the label frequency to 5, every fifth element in the legend vector will be displayed on the *Canvas*: *public int getLabelFrequency()* and
public void setLabelFrequency(int newValue).

Our class Diagram provides four different color modes you can set according to the type of your diagram.
A ***ColorMode*** can not be set for ***LineChart*** and ***VerticalLineChart***: they display only one color. You can change this color with the methods *public Color getColor()* and
public void setColor(Color color) . The default color is *Color.red*
For ***BarChart***, ***VerticalBarChart***, ***ScatterDiagram*** and ***PieChart***:

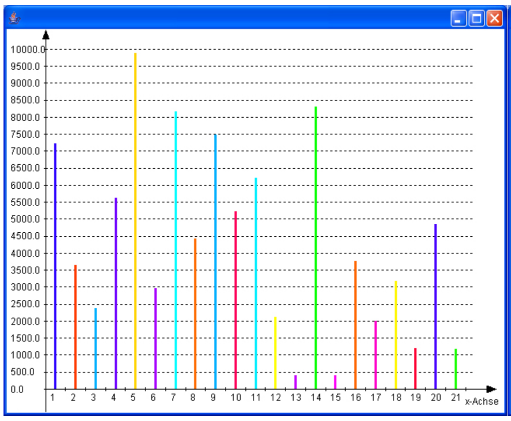
- colorMode 0 (the default color mode) displays four equidistant colors of the rainbow, and then varies them. Any two colors in the color vector cannot be identical.
- colorMode 1 displays two complementary colors of the rainbow, and then varies them. Any two colors in the color vector cannot be identical.
- colorMode 2 displays random colors of the rainbow. Two adjacent colors can not be too similar.
- colorMode 3 displays colors of the rainbow, from red through yellow green blue violet to red again. Any two colors in the color vector cannot be identical.



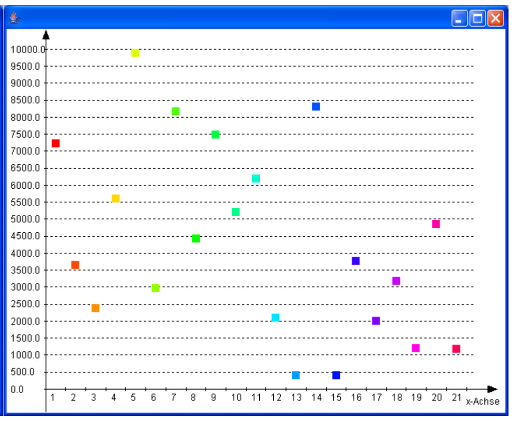
PieChart with ColorMode 0



VerticalBarChart w. ColorMode 1



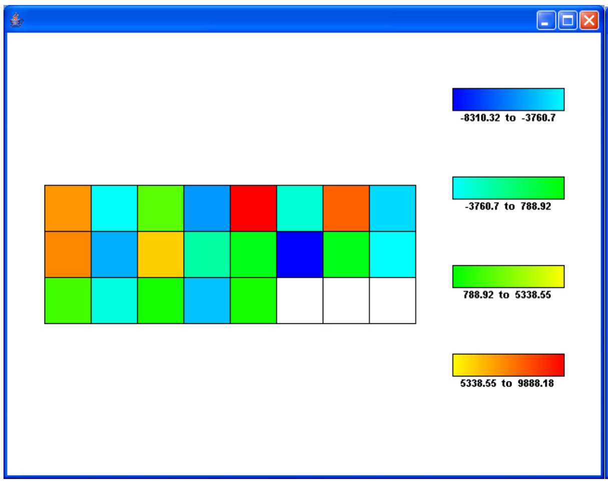
BarChart with ColorMode 2



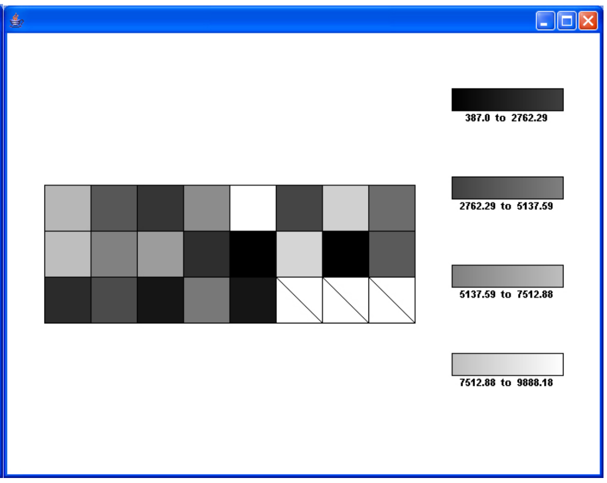
ScatterDiagram w. ColorMode 3

A ***SurfaceDiagram*** can be displayed in either **color** or **black and white**:

- colorMode 0 (the default color mode) sets the lowest value to blue and the highest to red.
- colorMode 1 sets the lowest value to black and the highest to white.



SurfaceDiagram with ColorMode 0



SurfaceDiagram with ColorMode 1

Two possibilities arose for the handling of the **consistency of the three vectors** (input color and legend). The first possibility would have been to allow any kind of object in a vector of any length to be passed to the diagram, implementing only valid values. While testing our program, we found this possibility to be very confusing, as it wasn't always evident in a diagram with various possible color modes, if a desired change really took place. On one hand, an error in a visual diagram wouldn't represent any notable threat to your program or your computer, but on the other hand, it isn't as apparent as we would like it to be: the human eye with all its weaknesses has to act as an arbitrator. We thought the visual aspect of a diagram shouldn't dilute its scientific value and its necessary preciseness. The second possibility we chose to follow was to allow only valid vectors to be passed to the diagram. Error messages would inform you about a possible ambiguity. Although it may sound a bit harsh, this kind of hygiene helped us a lot in avoiding inadvertencies we would probably not have been able to spot in a drawn diagram.

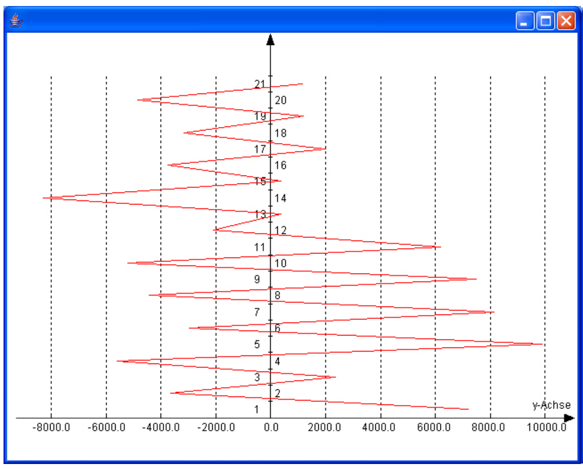
As a necessary matter of security, all vectors i.e. the **input-vector**, the **color-vector** and the **legend-vector, must have the same length** and contain only appropriate objects:

- **Double objects** for the input vector.
- **Color objects** for the color vector.
- **String objects** for the legend vector.

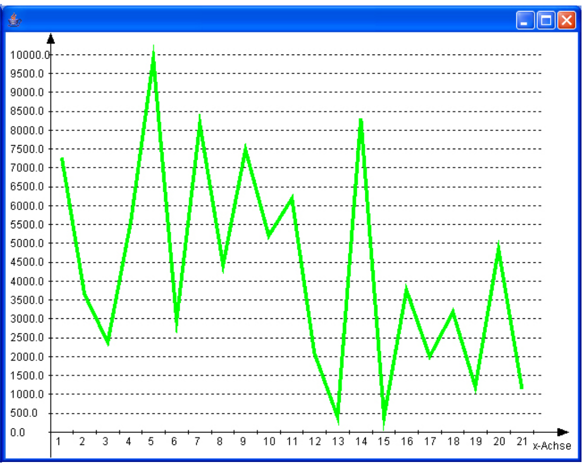
Otherwise a **runtime exception** will be **thrown** with a message describing the cause of the error. Now you have to choose how to handle this very strict application. We suggest three possible approaches:

1. You don't give a damn about any error message. You want the canvas not to react to a non-valid command. You should **try** and **catch** a **RuntimeException** every time you create a diagram (with more than one input vector) or pass it a color or legend vector.
2. You accept our help in case of a non-valid command, but don't want your program to terminate. You should **try** and **catch** a **RuntimeException** every time you create a diagram or pass it a color or legend vector and **retrieve** the exception **message** with the *getMessage()* method inherited from *Throwable*.
3. You want your program to terminate in case of a non-valid vector command. Don't catch any Exception. Any exception message will appear on your console display anyway.

By default a diagram *Canvas* will have a horizontal extent of 640 and a vertical extent of 480 pixels. You can naturally change this **size** by means of the *setSize(int x, int y)* method inherited from the class *Component*. Sizes inferior to (200,200) are not sensible, as the lines, legends and graphs won't be drawn correctly.
The **width of bars lines, points** can be set by the user. Negative values will not be accepted.
BarChart, *VerticalBarChart* (default *barExtent*: 4):
public float getBarExtent() and *public void setBarExtent(float barExtent)*
LineChart, *VerticalLineChart* (default *lineExtent*: 1):
public float getLineExtent() and *public void setLineExtent(float lineExtent)*
ScatterDiagram (default *pointExtent*: 4):
public float getPointExtent() and *public void setPointExtent(float pointExtent)*



VerticalLineChart with its default color and LineExtent



LineChart set to Color.green with LineExtent of 4 pixels

A class ***TwoCanvases*** creates a new canvas by appending one old canvas to the left of another old canvas. Both will keep their former dimensions.
public TwoCanvases(Canvas c1, Canvas c2)

All classes have been tested from Java 1.3 trough Java 1.5.0.
JUnit test methods were implemented to test the exact position of some given pixel, although visual tests of the generated canvases prevailed.